Contents

Chapter 1

Abstract Factory Pattern

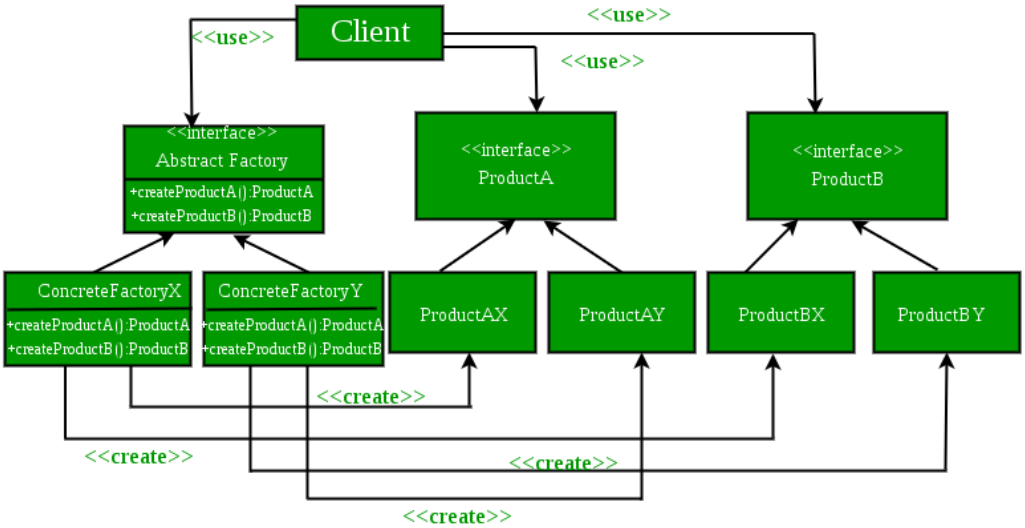
Abstract Factory Pattern - GeeksforGeeks

Introduction

Abstract Factory design pattern is one of the Creational pattern. Abstract Factory pattern is almost similar to Factory Pattern is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories.

Abstract factory pattern implementation provides us a framework that allows us to create objects that follow a general pattern. So at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type.

Let see the GOFs representation of Abstract Factory Pattern :



UML class diagram example for the Abstract Factory Design Pattern.

- **AbstractFactory** : Declares an interface for operations that create abstract product

objects.
- **ConcreteFactory** : Implements the operations declared in the AbstractFactory to create concrete product objects.
- **Product** : Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.
- **Client** : Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Factory provides interfaces for creating families of related or dependent objects without specifying their concrete classes.

Client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the family of objects.
The client does not know or care which concrete objects it gets from each of these concrete factories since it uses only the generic interfaces of their products.

So with this idea of Abstract Factory pattern, we will now try to create a design that will facilitate the creation of related objects.

Implementation

Let's take an example, Suppose we want to build a global car factory. If it was factory design pattern, then it was suitable for a single location. But for this pattern, we need multiple locations and some critical design changes.

We need car factories in each location like IndiaCarFactory, USACarFactory and Default-CarFactory. Now, our application should be smart enough to identify the location where it is being used, so we should be able to use appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling wrong factory for a particular location.

Here we need another layer of abstraction which will identify the location and internally use correct car factory implementation without even giving a single hint to user. This is exactly the problem, which abstract factory pattern is used to solve.

```
 // Java Program to demonstrate the
// working of Abstract Factory Pattern

enum CarType
{
    MICRO, MINI, LUXURY
}

abstract class Car
{
    Car(CarType model, Location location)
    {
        this.model = model;
        this.location = location;
    }
```

```
    abstract void construct();

    CarType model = null;
    Location location = null;

    CarType getModel()
    {
        return model;
    }

    void setModel(CarType model)
    {
        this.model = model;
    }

    Location getLocation()
    {
        return location;
    }

    void setLocation(Location location)
    {
        this.location = location;
    }

    @Override
    public String toString()
    {
        return "CarModel - "+model + " located in "+location;
    }
}

class LuxuryCar extends Car
{
    LuxuryCar(Location location)
    {
        super(CarType.LUXURY, location);
        construct();
    }
    @Override
    protected void construct()
    {
        System.out.println("Connecting to luxury car");
    }
}

class MicroCar extends Car
{
```

```
    MicroCar(Location location)
    {
        super(CarType.MICRO, location);
        construct();
    }
    @Override
    protected void construct()
    {
        System.out.println("Connecting to Micro Car ");
    }
}

class MiniCar extends Car
{
    MiniCar(Location location)
    {
        super(CarType.MINI,location );
        construct();
    }

    @Override
    void construct()
    {
        System.out.println("Connecting to Mini car");
    }
}

enum Location
{
  DEFAULT, USA, INDIA
}

class INDIACarFactory
{
    static Car buildCar(CarType model)
    {
        Car car = null;
        switch (model)
        {
            case MICRO:
                car = new MicroCar(Location.INDIA);
                break;

            case MINI:
                car = new MiniCar(Location.INDIA);
                break;

            case LUXURY:
```

```
                car = new LuxuryCar(Location.INDIA);
                break;

                default:
                break;

        }
        return car;
    }
}

class DefaultCarFactory
{
    public static Car buildCar(CarType model)
    {
        Car car = null;
        switch (model)
        {
            case MICRO:
                car = new MicroCar(Location.DEFAULT);
                break;

            case MINI:
                car = new MiniCar(Location.DEFAULT);
                break;

            case LUXURY:
                car = new LuxuryCar(Location.DEFAULT);
                break;

                default:
                break;

        }
        return car;
    }
}


class USACarFactory
{
    public static Car buildCar(CarType model)
    {
        Car car = null;
        switch (model)
        {
            case MICRO:
                car = new MicroCar(Location.USA);
```

```
                break;

            case MINI:
                car = new MiniCar(Location.USA);
                break;

            case LUXURY:
                car = new LuxuryCar(Location.USA);
                break;

                default:
                break;

        }
        return car;
    }
}



class CarFactory
{
    private CarFactory()
    {

    }
    public static Car buildCar(CarType type)
    {
        Car car = null;
        // We can add any GPS Function here which
        // read location property somewhere from configuration
        // and use location specific car factory
        // Currently I'm just using INDIA as Location
        Location location = Location.INDIA;

        switch(location)
        {
            case USA:
                car = USACarFactory.buildCar(type);
                break;

            case INDIA:
                car = INDIACarFactory.buildCar(type);
                break;

            default:
                car = DefaultCarFactory.buildCar(type);
```

```
        }

        return car;

    }
}

class AbstractDesign
{
    public static void main(String[] args)
    {
        System.out.println(CarFactory.buildCar(CarType.MICRO));
        System.out.println(CarFactory.buildCar(CarType.MINI));
        System.out.println(CarFactory.buildCar(CarType.LUXURY));
    }
}
```

Output :

```
Connecting to Micro Car
CarModel - MICRO located in INDIA
Connecting to Mini car
CarModel - MINI located in INDIA
Connecting to luxury car
CarModel - LUXURY located in INDIA
```

Difference

- The main difference between a "factory method" and an "abstract factory" is that the factory method is a single method, and an abstract factory is an object.
- The factory method is just a method, it can be overridden in a subclass, whereas the abstract factory is an object that has multiple factory methods on it.
- The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

Advantages

This pattern is particularly useful when the client doesn't know exactly what type to create.

- **Isolation of concrete classes:** The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.

- **Exchanging Product Families easily:** The class of a concrete factory appears only once in an application, that is where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use various product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once.
- **Promoting consistency among products:** When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.n.

Disadvantages

- **Difficult to support new kind of products:** Extending abstract factories to produce new kinds of Products isn't easy. That's because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses.

NOTE :
Somewhat the above example is also based on How the Cabs like uber and ola functions on the large scale.

Source

https://www.geeksforgeeks.org/abstract-factory-pattern/

Chapter 2

Adapter Pattern

Adapter Pattern - GeeksforGeeks

This pattern is easy to understand as the real world is full of adapters. For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.

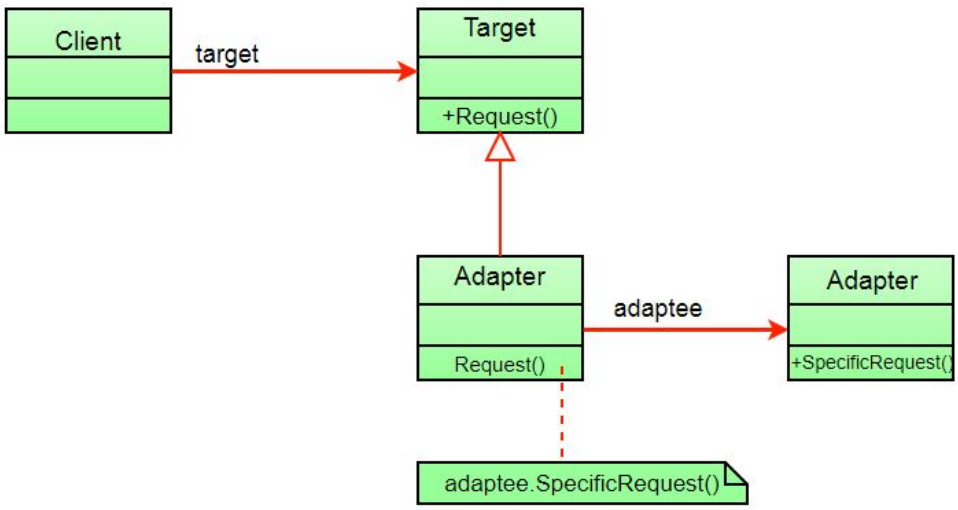
To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter's presence.

Definition:

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

Class Diagram:



The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adaptee.

Example:

Suppose you have a Bird class with fly() , and makeSound()methods. And also a ToyDuck class with squeak() method. Let's assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can't use them directly. So we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.

Below is Java implementation of it.

```
 // Java implementation of Adapter pattern

interface Bird
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interface
    public void fly();
    public void makeSound();
}

class Sparrow implements Bird
{
    // a concrete implementation of bird
    public void fly()
    {
        System.out.println("Flying");
    }
```

```
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}

interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}

class PlasticToyDuck implements ToyDuck
{
    public void squeak()
    {
        System.out.println("Squeak");
    }
}

class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }

    public void squeak()
    {
        // translate the methods appropriately
        bird.makeSound();
    }
}

class Main
{
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        PlasticToyDuck toyDuck = new PlasticToyDuck();
```

```
        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // bird behaving like a toy duck
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```

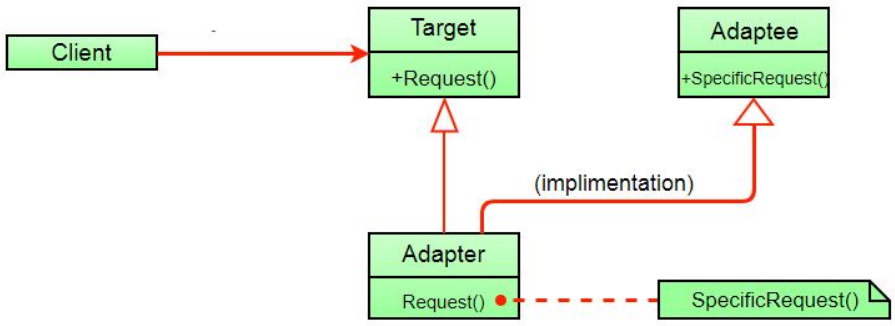
Output:

```
Sparrow...
Flying
Chirp Chirp
ToyDuck...
Squeak
BirdAdapter...
Chirp Chirp
```

Object Adapter Vs Class Adapter

The adapter pattern we have implemented above is called Object Adapter Pattern because the adapter holds an instance of adaptee. There is also another type called Class Adapter Pattern which use inheritance instead of composition but you require multiple inheritance to implement it.

Class diagram of Class Adapter Pattern:



Here instead of having an adaptee object inside adapter (composition) to make use of its functionality adapter inherits the adaptee.

Since multiple inheritance is not supported by many languages including java and is associated with many problems we have not shown implementation using class adapter pattern.

Advantages:

- Helps achieve reusability and flexibility.
- Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

Disadvantages:

- All requests are forwarded, so there is a slight increase in the overhead.
- Sometimes many adaptations are required along an adapter chain to reach the type which is required.

References:
Head First Design Patterns (Book)

Source

https://www.geeksforgeeks.org/adapter-pattern/

Chapter 3

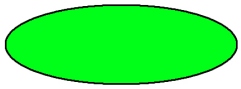
An introduction to Flowcharts

An introduction to Flowcharts - GeeksforGeeks

What is a Flowchart?
Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.
The process of drawing a flowchart for an algorithm is known as "flowcharting".

Basic Symbols used in Flowchart Designs

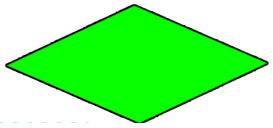
1. **Terminal:** The oval symbol indicates Start, Stop and Halt in a program's logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



2. **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.

3. **Processing:** A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.

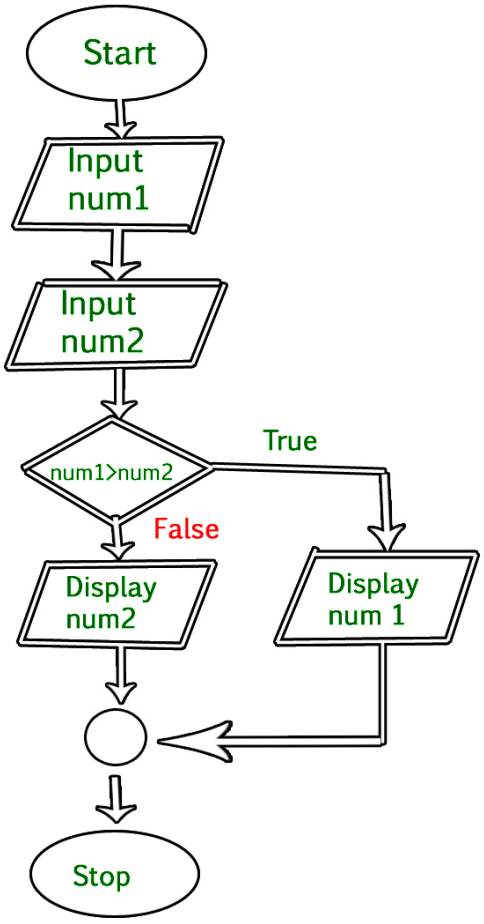
4. **Decision** Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



5. **Connectors:** Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.

6. **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.

Example : Draw a flowchart to input two numbers from user and display the largest of two numbers



C++

```
 // C++ program to find largest of two numbers
#include <iostream>
using namespace std;
int main()
{
    int num1, num2, largest;

    /*Input two numbers*/
    cout << "Enter two numbers:\n";
    cin >> num1;
    cin >> num2;

    /*check if a is greater than b*/
    if (num1 > num2)
        largest = num1;
```

```
    else
        largest = num2;

    /*Print the largest number*/
    cout << largest;

    return 0;
}
```

Java

```
 // Java program to find largest of two numbers
import java.util.Scanner;
public class largest {
    public static void main(String args[])
    {
        int num1, num2, max;

        /*Input two numbers*/
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter two numbers:");

        num1 = sc.nextInt();
        num2 = sc.nextInt();

        /*check whether a is greater than b or not*/
        if (num1 > num2)
            max = num1;
        else
            max = num2;

        /*Print the largest number*/
        System.out.println(max);
    }
}
```

Output

```
Enter two numbers:
10 30

30
```

References:
Computer Fundamentals by Pradeep K. Sinha and Priti Sinha

Source

https://www.geeksforgeeks.org/an-introduction-to-flowcharts/

Chapter 4

Bridge Design Pattern

Bridge Design Pattern - GeeksforGeeks

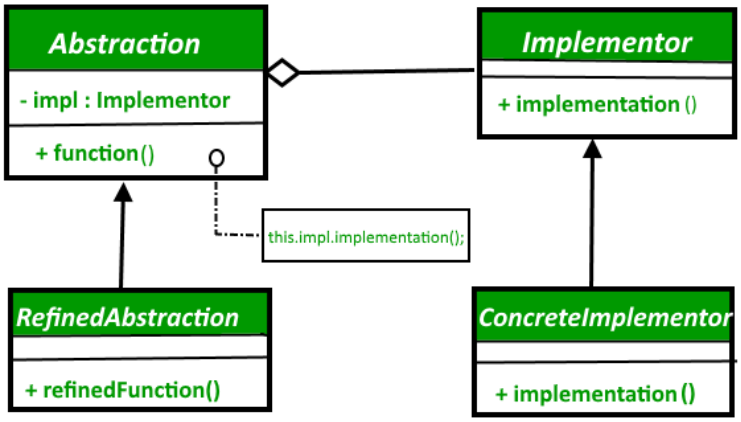
The Bridge design pattern allows you to separate the abstraction from the implementation.It is a structural design pattern.
There are 2 parts in Bridge design pattern :

1. Abstraction
2. Implementation

This is a design mechanism that encapsulates an implementation class inside of an interface class.

- The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.
- The abstraction is an interface or abstract class and the implementor is also an interface or abstract class.
- The abstraction contains a reference to the implementor. Children of the abstraction are referred to as refined abstractions, and children of the implementor are concrete implementors. Since we can change the reference to the implementor in the abstraction, we are able to change the abstraction's implementor at run-time. Changes to the implementor do not affect client code.
- It increases the loose coupling between class abstraction and it's implementation.

UML Diagram of Bridge Design Pattern



Elements of Bridge Design Pattern

- **Abstraction** – core of the bridge design pattern and defines the crux. Contains a reference to the implementer.
- **Refined Abstraction** – Extends the abstraction takes the finer detail one level below. Hides the finer elements from implemetors.
- **Implementer** – It defines the interface for implementation classes. This interface does not need to correspond directly to abstraction interface and can be very different. Abstraction imp provides an implementation in terms of operations provided by Implementer interface.
- **Concrete Implementation** – Implements the above implementer by providing concrete implementation.

Lets see an Example of Bridge Design Pattern :

```
 // Java code to demonstrate
// bridge design 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()
    {
        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();
    }
}
```

Output :

```
Car Produced And Assembled.
Bike Produced And Assembled.
```

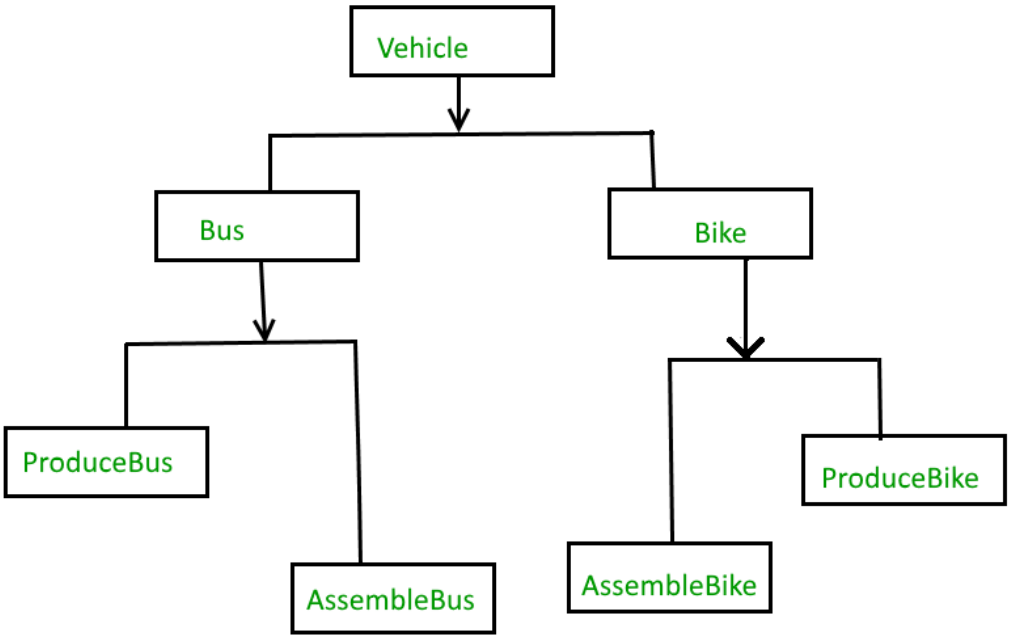
Here we're producing and assembling the two different vehicles using Bridge design pattern.

When we need bridge design pattern

The Bridge pattern is an application of the old advice, "prefer composition over inheritance". It becomes handy when you must subclass different times in ways that are orthogonal with one another.

For Example, the above example can also be done something like this :

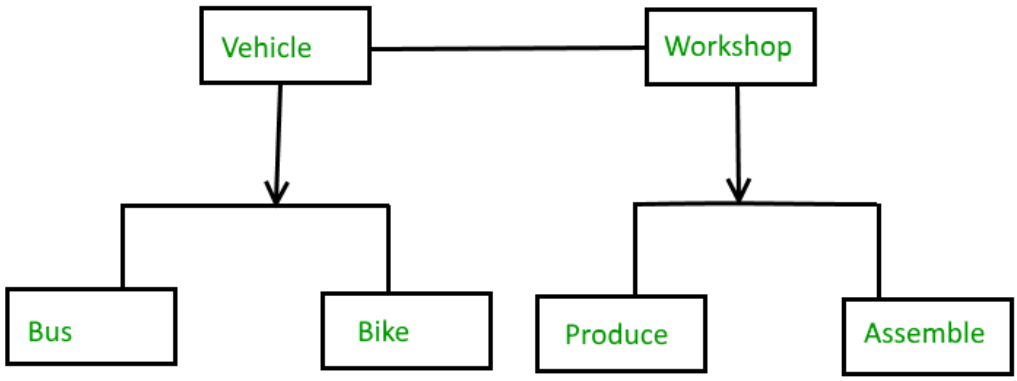
Without Bridge Design Pattern



But the above solution has a problem. If you want to change 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 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.



Advantages

1. Bridge pattern decouple an abstraction from its implementation so that the two can vary independently.
2. It is used mainly for implementing platform independence feature.
3. It adds one more method level redirection to achieve the objective.

4. Publish abstraction interface in a separate inheritance hierarchy, and put the implementation in its own inheritance hierarchy.
5. Use bridge pattern to run-time binding of the implementation.
6. Use bridge pattern to map orthogonal class hierarchies
7. Bridge is designed up-front to let the abstraction and the implementation vary independently.

Improved By : sunny94

Source

https://www.geeksforgeeks.org/bridge-design-pattern/

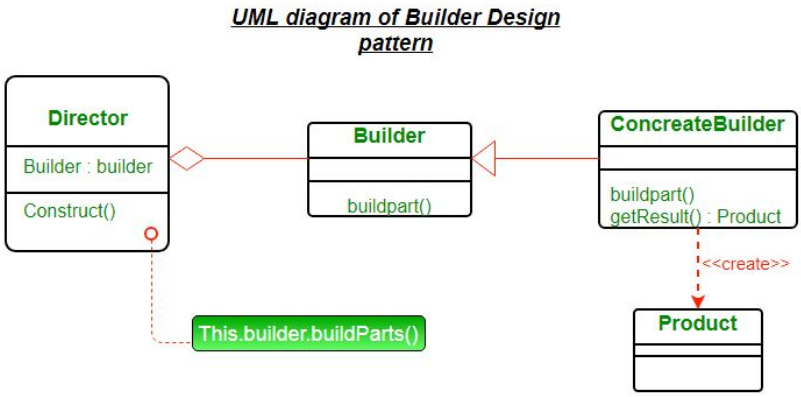
Chapter 5

Builder Design Pattern

Builder Design Pattern - GeeksforGeeks

Builder pattern aims to "Separate the construction of a complex object from its representation so that the same construction process can create different representations." It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

UML Diagram of Builder Design Pattern



Source : Wikipedia

- **Product** – The product class defines the type of the complex object that is to be generated by the builder pattern.
- **Builder** – This abstract base class defines all of the steps that must be taken in order to correctly create a product. Each step is generally abstract as the actual functionality of the builder is carried out in the concrete subclasses. The GetProduct method is used to return the final product. The builder class is often replaced with a simple interface.

- **ConcreteBuilder** – There may be any number of concrete builder classes inheriting from Builder. These classes contain the functionality to create a particular complex product.
- **Director** – The director class controls the algorithm that generates the final product object. A director object is instantiated and its Construct method is called. The method includes a parameter to capture the specific concrete builder object that is to be used to generate the product. The director then calls methods of the concrete builder in the correct order to generate the product object. On completion of the process, the GetProduct method of the builder object can be used to return the product.

Lets see an Example of Builder Design Pattern :

Consider a construction of a home. Home is the final end product (object) that is to be returned as the output of the construction process. It will have many steps like basement construction, wall construction and so on roof construction. Finally the whole home object is returned. Here using the same process you can build houses with different properties.

```
 interface HousePlan
{
    public void setBasement(String basement);

    public void setStructure(String structure);

    public void setRoof(String roof);

    public void setInterior(String interior);
}

class House implements HousePlan
{

    private String basement;
    private String structure;
    private String roof;
    private String interior;

    public void setBasement(String basement)
    {
        this.basement = basement;
    }

    public void setStructure(String structure)
    {
        this.structure = structure;
    }

    public void setRoof(String roof)
    {
```

```
        this.roof = roof;
    }

    public void setInterior(String interior)
    {
        this.interior = interior;
    }

}


interface HouseBuilder
{

    public void buildBasement();

    public void buildStructure();

    public void bulidRoof();

    public void buildInterior();

    public House getHouse();
}

class IglooHouseBuilder implements HouseBuilder
{
    private House house;

    public IglooHouseBuilder()
    {
        this.house = new House();
    }

    public void buildBasement()
    {
        house.setBasement("Ice Bars");
    }

    public void buildStructure()
    {
        house.setStructure("Ice Blocks");
    }

    public void buildInterior()
    {
        house.setInterior("Ice Carvings");
    }
```

```
    public void bulidRoof()
    {
        house.setRoof("Ice Dome");
    }

    public House getHouse()
    {
        return this.house;
    }
}

class TipiHouseBuilder implements HouseBuilder
{
    private House house;

    public TipiHouseBuilder()
    {
        this.house = new House();
    }

    public void buildBasement()
    {
        house.setBasement("Wooden Poles");
    }

    public void buildStructure()
    {
        house.setStructure("Wood and Ice");
    }

    public void buildInterior()
    {
        house.setInterior("Fire Wood");
    }

    public void bulidRoof()
    {
        house.setRoof("Wood, caribou and seal skins");
    }

    public House getHouse()
    {
        return this.house;
    }

}
```

```
class CivilEngineer
{

    private HouseBuilder houseBuilder;

    public CivilEngineer(HouseBuilder houseBuilder)
    {
        this.houseBuilder = houseBuilder;
    }

    public House getHouse()
    {
        return this.houseBuilder.getHouse();
    }

    public void constructHouse()
    {
        this.houseBuilder.buildBasement();
        this.houseBuilder.buildStructure();
        this.houseBuilder.bulidRoof();
        this.houseBuilder.buildInterior();
    }
}

class Builder
{
    public static void main(String[] args)
    {
        HouseBuilder iglooBuilder = new IglooHouseBuilder();
        CivilEngineer engineer = new CivilEngineer(iglooBuilder);

        engineer.constructHouse();

        House house = engineer.getHouse();

        System.out.println("Builder constructed: "+ house);
    }
}
```

Output :

```
Builder constructed: House@6d06d69c
```

Advantages of Builder Design Pattern

- The parameters to the constructor are reduced and are provided in highly readable method calls.

- Builder design pattern also helps in minimizing the number of parameters in constructor and thus there is no need to pass in null for optional parameters to the constructor.
- Object is always instantiated in a complete state
- Immutable objects can be build without much complex logic in object building process.

Disadvantages of Builder Design Pattern

- The number of lines of code increase at least to double in builder pattern, but the effort pays off in terms of design flexibility and much more readable code.
- Requires creating a separate ConcreteBuilder for each different type of Product.

Source

https://www.geeksforgeeks.org/builder-design-pattern/

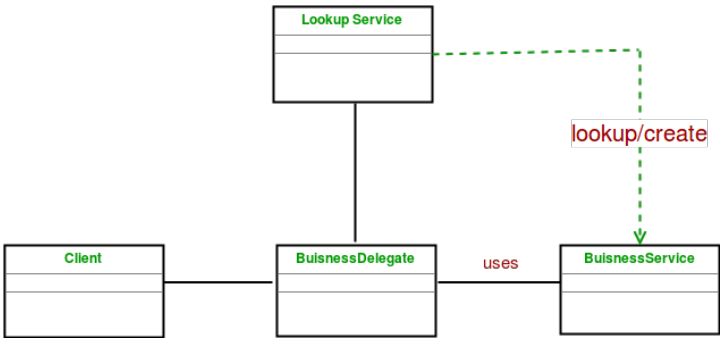
Chapter 6

Business Delegate Pattern

Business Delegate Pattern - GeeksforGeeks

The Business Delegate acts as a client-side business abstraction, it provides an abstraction for, and thus hides, the implementation of the business services. It reduces the coupling between presentation-tier clients and the system's Business services.

UML Diagram Business Delegate Pattern



Design components

- **Business Delegate :** A single entry point class for client entities to provide access to Business Service methods.
- **LookUp Service :** Lookup service object is responsible to get relative business implementation and provide business object access to business delegate object.
- **Business Service :** Business Service interface. Concrete classes implement this business service to provide actual business implementation logic.

Let's see an example of Business Delegate Pattern.

```
 interface BusinessService
{
```

```
    public void doProcessing();
}

class OneService implements BusinessService
{
    public void doProcessing()
    {
        System.out.println("Processed Service One");
    }
}

class TwoService implements BusinessService
{
    public void doProcessing()
    {
        System.out.println("Processed Service Two");
    }
}

class BusinessLookUp
{
    public BusinessService getBusinessService(String serviceType)
    {
        if(serviceType.equalsIgnoreCase("One"))
        {
            return new OneService();
        }
        else
        {
            return new TwoService();
        }
   }
}

class BusinessDelegate
{
    private BusinessLookUp lookupService = new BusinessLookUp();
    private BusinessService businessService;
    private String serviceType;

    public void setServiceType(String serviceType)
    {
        this.serviceType = serviceType;
    }

    public void doTask()
    {
        businessService = lookupService.getBusinessService(serviceType);
```

```
        businessService.doProcessing();
    }
}

class Client
{
    BusinessDelegate businessService;

    public Client(BusinessDelegate businessService)
    {
        this.businessService  = businessService;
    }

    public void doTask()
    {
        businessService.doTask();
    }
}

class BusinessDelegatePattern
{
    public static void main(String[] args)
    {
        BusinessDelegate businessDelegate = new BusinessDelegate();
        businessDelegate.setServiceType("One");

        Client client = new Client(businessDelegate);
        client.doTask();

        businessDelegate.setServiceType("Two");
        client.doTask();
    }
}
```

Output:

```
Processed Service One
Processed Service Two
```

Advantages :

- Business Delegate reduces coupling between presentation-tier clients and Business services.
- The Business Delegate hides the underlying implementation details of the Business service.

Disadvantages :

- Maintenance due the extra layer that increases the number of classes in the application.

Source

https://www.geeksforgeeks.org/business-delegate-pattern/

Chapter 7

Chain of Responsibility Design Pattern

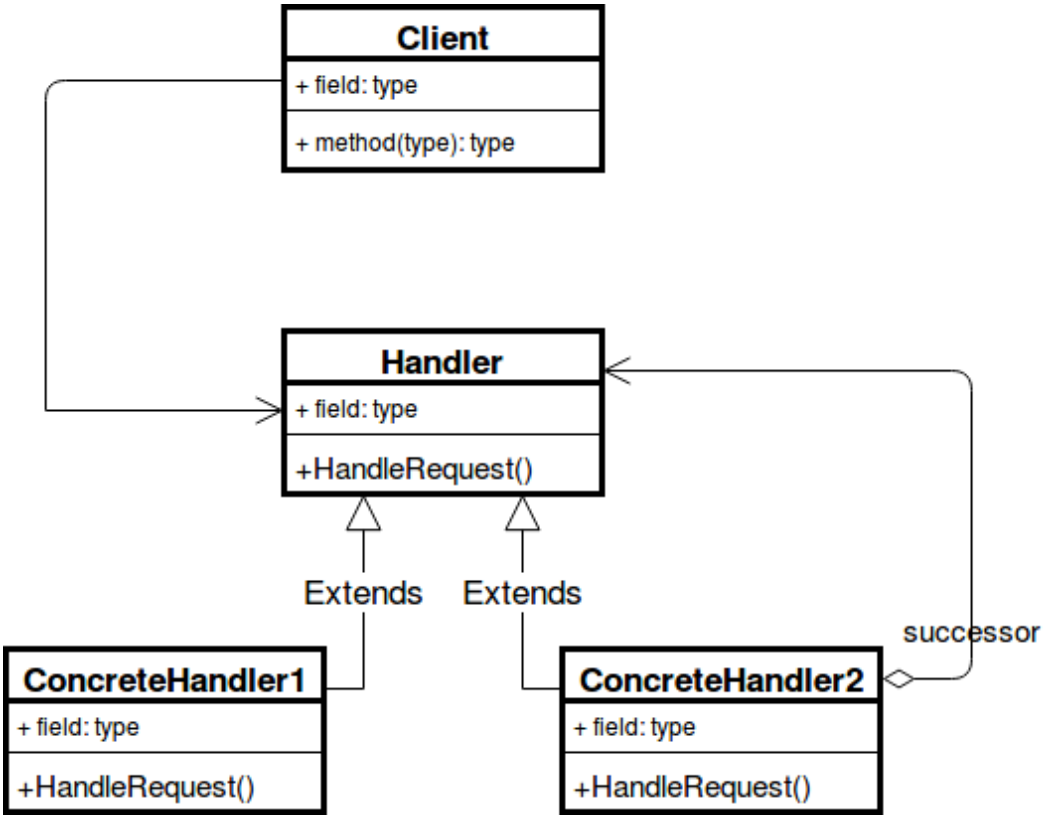
Chain of Responsibility Design Pattern - GeeksforGeeks

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

Where and When Chain of Responsibility pattern is applicable :

- When you want to decouple a request's sender and receiver
- Multiple objects, determined at runtime, are candidates to handle a request
- When you don't want to specify handlers explicitly in your code
- When you want to issue a request to one of several objects without specifying the receiver explicitly.

This pattern is recommended when multiple objects can handle a request and the handler doesn't have to be a specific object. Also, handler is determined at runtime. Please note that that a request not handled at all by any handler is a valid use case.



- **Handler :** This can be an interface which will primarily recieve the request and dispatches the request to chain of handlers. It has reference of only first handler in the chain and does not know anything about rest of the handlers.
- **Concrete handlers :** These are actual handlers of the request chained in some sequential order.
- **Client :** Originator of request and this will access the handler to handle it.

How to send a request in the application using the Chain of Responsibility

The Client in need of a request to be handled sends it to the chain of handlers which are classes that extend the Handler class.
Each of the handlers in the chain takes its turn at trying to handle the request it receives from the client.
If ConcreteHandler1 can handle it, then the request is handled, if not it is sent to the handler ConcreteHandler2, the next one in the chain.

Lets see an Example of Chain of Responsibility Design Pattern:

```
 interface Chain
{
    public abstract void setNext(Chain nextInChain);
    public abstract void process(Number request);
```

```
}

class Number
{
    private int number;

    public Number(int number)
    {
        this.number = number;
    }

    public int getNumber()
    {
        return number;
    }

}

class NegativeProcessor implements Chain
{
    private Chain nextInChain;

    public void setNext(Chain c)
    {
        nextInChain = c;
    }

    public void process(Number request)
    {
        if (request.getNumber() < 0)
        {
            System.out.println("NegativeProcessor : " + request.getNumber());
        }
        else
        {
            nextInChain.process(request);
        }
    }
}

class ZeroProcessor implements Chain
{

    private Chain nextInChain;

    public void setNext(Chain c)
    {
        nextInChain = c;
```

```
    }

    public void process(Number request)
    {
        if (request.getNumber() == 0)
        {
            System.out.println("ZeroProcessor : " + request.getNumber());
        }
        else
        {
            nextInChain.process(request);
        }
    }
}

class PositiveProcessor implements Chain
{

    private Chain nextInChain;

    public void setNext(Chain c)
    {
        nextInChain = c;
    }

    public void process(Number request)
    {
        if (request.getNumber() > 0)
        {
            System.out.println("PositiveProcessor : " + request.getNumber());
        }
        else
        {
            nextInChain.process(request);
        }
    }
}

class TestChain
{
    public static void main(String[] args) {
        //configure Chain of Responsibility
        Chain c1 = new NegativeProcessor();
        Chain c2 = new ZeroProcessor();
        Chain c3 = new PositiveProcessor();
        c1.setNext(c2);
        c2.setNext(c3);
```

```
        //calling chain of responsibility
        c1.process(new Number(90));
        c1.process(new Number(-50));
        c1.process(new Number(0));
        c1.process(new Number(91));
    }
}
```

Output :

```
PositiveProcessor : 90
NegativeProcessor : -50
ZeroProcessor : 0
PositiveProcessor : 91
```

Advantages of Chain of Responsibility Design Pattern

- To reduce the coupling degree. Decoupling it will request the sender and receiver.
- Simplified object. The object does not need to know the chain structure.
- Enhance flexibility of object assigned duties. By changing the members within the chain or change their order, allow dynamic adding or deleting responsibility.
- Increase the request processing new class of very convenient.

DisAdvantages of Chain of Responsibility Design Pattern

- The request must be received not guarantee.
- The performance of the system will be affected, but also in the code debugging is not easy may cause cycle call.
- It may not be easy to observe the characteristics of operation, due to debug.

Source

https://www.geeksforgeeks.org/chain-responsibility-design-pattern/

Chapter 8

Command Pattern

Command Pattern - GeeksforGeeks

Like previous articles, let us take up a design problem to understand command pattern. Suppose you are building a home automation system. There is a programmable remote which can be used to turn on and off various items in your home like lights, stereo, AC etc. It looks something like this.

You can do it with simple if-else statements like

```
if (buttonPressed == button1)
     lights.on()
```

But we need to keep in mind that turning on some devices like stereo comprises of many steps like setting cd, volume etc. Also we can reassign a button to do something else. By using simple if-else we are coding to implementation rather than interface. Also there is tight coupling.

So what we want to achieve is a design that provides loose coupling and remote control should not have much information about a particular device. The command pattern helps us do that.

Definition: The **command pattern** encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

The definition is a bit confusing at first but let's step through it. In analogy to our problem above remote control is the client and stereo, lights etc. are the receivers. In command pattern there is a Command object that *encapsulates a request* by binding together a set of actions on a specific receiver. It does so by exposing just one method execute() that causes some actions to be invoked on the receiver.

Parameterizing other objects with different requests in our analogy means that the button used to turn on the lights can later be used to turn on stereo or maybe open the garage door.

queue or log requests, and support undoable operations means that Command's Execute operation can store state for reversing its effects in the Command itself. The Command may have an added unExecute operation that reverses the effects of a previous call to execute.It may also support logging changes so that they can be reapplied in case of a system crash.

Below is the Java implementation of above mentioned remote control example:

```
 // A simple Java program to demonstrate
// implementation of Command Pattern using
// a remote control example.

// An interface for command
interface Command
{
    public void execute();
}

// Light class and its corresponding command
// classes
class Light
{
    public void on()
    {
        System.out.println("Light is on");
    }
    public void off()
    {
        System.out.println("Light is off");
    }
}
class LightOnCommand implements Command
{
    Light light;

    // The constructor is passed the light it
    // is going to control.
    public LightOnCommand(Light light)
    {
       this.light = light;
    }
    public void execute()
    {
       light.on();
    }
}
class LightOffCommand implements Command
{
    Light light;
    public LightOffCommand(Light light)
```

```
    {
        this.light = light;
    }
    public void execute()
    {
         light.off();
    }
}

// Stereo and its command classes
class Stereo
{
    public void on()
    {
        System.out.println("Stereo is on");
    }
    public void off()
    {
        System.out.println("Stereo is off");
    }
    public void setCD()
    {
        System.out.println("Stereo is set " +
                           "for CD input");
    }
    public void setDVD()
    {
        System.out.println("Stereo is set"+
                          " for DVD input");
    }
    public void setRadio()
    {
        System.out.println("Stereo is set" +
                           " for Radio");
    }
    public void setVolume(int volume)
    {
       // code to set the volume
       System.out.println("Stereo volume set"
                          + " to " + volume);
    }
}
class StereoOffCommand implements Command
{
    Stereo stereo;
    public StereoOffCommand(Stereo stereo)
    {
        this.stereo = stereo;
```

```
    }
    public void execute()
    {
       stereo.off();
    }
}
class StereoOnWithCDCommand implements Command
{
     Stereo stereo;
     public StereoOnWithCDCommand(Stereo stereo)
     {
         this.stereo = stereo;
     }
     public void execute()
     {
         stereo.on();
         stereo.setCD();
         stereo.setVolume(11);
     }
}

// A Simple remote control with one button
class SimpleRemoteControl
{
    Command slot;  // only one button

    public SimpleRemoteControl()
    {
    }

    public void setCommand(Command command)
    {
        // set the command the remote will
        // execute
        slot = command;
    }

    public void buttonWasPressed()
    {
        slot.execute();
    }
}

// Driver class
class RemoteControlTest
{
    public static void main(String[] args)
    {
```

```
        SimpleRemoteControl remote =
                  new SimpleRemoteControl();
        Light light = new Light();
        Stereo stereo = new Stereo();

        // we can change command dynamically
        remote.setCommand(new
                    LightOnCommand(light));
        remote.buttonWasPressed();
        remote.setCommand(new
                StereoOnWithCDCommand(stereo));
        remote.buttonWasPressed();
        remote.setCommand(new
                   StereoOffCommand(stereo));
        remote.buttonWasPressed();
    }
}
```

Output:

```
Light is on
Stereo is on
Stereo is set for CD input
Stereo volume set to 11
Stereo is off
```

Notice that the remote control doesn't know anything about turning on the stereo. That information is contained in a separate command object. This reduces the coupling between them.

Advantages:

- Makes our code extensible as we can add new commands without changing existing code.
- Reduces coupling the invoker and receiver of a command.

Disadvantages:

- Increase in the number of classes for each individual command

References:

- Head First Design Patterns (book)
- https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/command

Source

https://www.geeksforgeeks.org/command-pattern/

Chapter 9

Composite Design Pattern

Composite Design Pattern - GeeksforGeeks

Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. It allows you to have a tree structure and ask each node in the tree structure to perform a task.

- As described by Gof, "Compose objects into tree structure to represent **part-whole hierarchies**. Composite lets client treat individual objects and compositions of objects uniformly".
- When dealing with Tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an **interface** that allows treating complex and primitive objects uniformly.
- In object-oriented programming, a composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality. This is known as a **"has-a"** relationship between objects.

1. **Component** − Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.
2. **Leaf** − Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
3. **Composite** − Composite stores child components and implements child related operations in the component interface.
4. **Client** − Client manipulates the objects in the composition through the component interface.

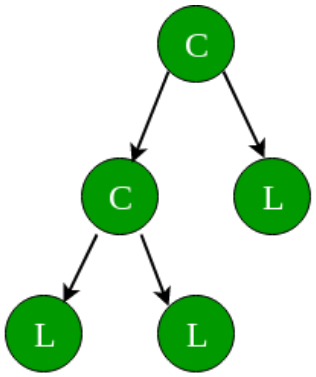
Client use the component class interface to interact with objects in the composition structure. If recipient is a leaf then request is handled directly. If recipient is a composite, then it

usually forwards request to its child components, possibly performing additional operations before and after forwarding.

Real Life example

In an organization, It have general managers and under general managers, there can be managers and under managers there can be developers. Now you can set a tree structure and ask each node to perform common operation like getSalary().
Composite design pattern treats each node in two ways:
1) **Composite** – Composite means it can have other objects below it.
2) **leaf** – leaf means it has no objects below it.

Tree structure:



Where, C = Composite & L = Leaf

The above figure shows a typical Composite object structure. As you can see, there can be many children to a single parent i.e. Composite, but only one parent per child.

Interface Component.java

```
 public interface Employee
{
    public void showEmployeeDetails();
}
```

Leaf.java

```
 public class Developer implements Employee
{
    private String name;
    private long empId;
    private String position;

    public Developer(long empId, String name, String position)
    {
        this.empId = empId;
```

```
        this.name = name;
                this.position = position;
    }

    @Override
    public void showEmployeeDetails()
    {
        System.out.println(empId+" " +name+);
    }
}
```

Leaf.java

```
 public class Manager implements Employee
{
    private String name;
    private long empId;
        private String position;

    public Manager(long empId, String name, String position)
    {
        this.empId = empId;
        this.name = name;
                this.position = position;
    }

    @Override
    public void showEmployeeDetails()
    {
        System.out.println(empId+" " +name);
    }
}
```

Composite.java

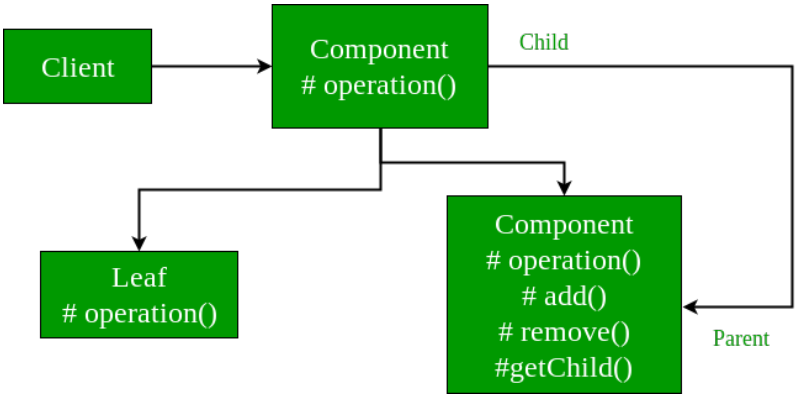
```
 import java.util.ArrayList;
import java.util.List;

public class CompanyDirectory implements Employee
{
    private List<Employee> employeeList = new ArrayList<Employee>();

    @Override
    public void showEmployeeDetails()
    {
        for(Employee emp:employeeList)
        {
```

```
            emp.showEmployeeDetails();
        }
    }

    public void addEmployee(Employee emp)
    {
        employeeList.add(emp);
    }

    public void removeEmployee(Employee emp)
    {
        employeeList.remove(emp);
    }
}
```

Client.java

```
 public class Company
{
    public static void main (String[] args)
    {
        Developer dev1 = new Developer(100, "Lokesh Sharma", "Pro Developer");
        Developer dev2 = new Developer(101, "Vinay Sharma", "Developer");
        CompanyDirectory engDirectory = new CompanyDirectory();
        engDirectory.addEmployee(dev1);
        engDirectory.addEmployee(dev2);

        Manager man1 = new Manager(200, "Kushagra Garg", "SEO Manager");
        Manager man2 = new Manager(201, "Vikram Sharma ", "Kushagra's Manager");

        CompanyDirectory accDirectory = new CompanyDirectory();
        accDirectory.addEmployee(man1);
        accDirectory.addEmployee(man2);

        CompanyDirectory directory = new CompanyDirectory();
        directory.addEmployee(engDirectory);
        directory.addEmployee(accDirectory);
        directory.showEmployeeDetails();
    }
}
```

UML Diagram for the Composite Design Pattern :



Full Running Code for the above example :

```
 // A Java program to demonstrate working of
// Composite Design Pattern with example
// of a company with different
//  employee details

import java.util.ArrayList;
import java.util.List;


// A common interface for all employee
interface Employee
{
    public void showEmployeeDetails();
}

class Developer implements Employee
{
    private String name;
    private long empId;
    private String position;

    public Developer(long empId, String name, String position)
    {
        // Assign the Employee id,
        // name and the position
        this.empId = empId;
        this.name = name;
        this.position = position;
    }

    @Override
    public void showEmployeeDetails()
    {
```

```
        System.out.println(empId+" " +name+ " " + position );
    }
}

class Manager implements Employee
{
    private String name;
    private long empId;
    private String position;

    public Manager(long empId, String name, String position)
    {
        this.empId = empId;
        this.name = name;
        this.position = position;
    }

    @Override
    public void showEmployeeDetails()
    {
        System.out.println(empId+" " +name+ " " + position );
    }
}


// Class used to get Employee List
// and do the opertions like
// add or remove Employee

class CompanyDirectory implements Employee
{
    private List<Employee> employeeList = new ArrayList<Employee>();

    @Override
    public void showEmployeeDetails()
    {
        for(Employee emp:employeeList)
        {
            emp.showEmployeeDetails();
        }
    }

    public void addEmployee(Employee emp)
    {
        employeeList.add(emp);
    }

    public void removeEmployee(Employee emp)
```

```
    {
        employeeList.remove(emp);
    }
}

// Driver class
public class Company
{
    public static void main (String[] args)
    {
        Developer dev1 = new Developer(100, "Lokesh Sharma", "Pro Developer");
        Developer dev2 = new Developer(101, "Vinay Sharma", "Developer");
        CompanyDirectory engDirectory = new CompanyDirectory();
        engDirectory.addEmployee(dev1);
        engDirectory.addEmployee(dev2);

        Manager man1 = new Manager(200, "Kushagra Garg", "SEO Manager");
        Manager man2 = new Manager(201, "Vikram Sharma ", "Kushagra's Manager");

        CompanyDirectory accDirectory = new CompanyDirectory();
        accDirectory.addEmployee(man1);
        accDirectory.addEmployee(man2);

        CompanyDirectory directory = new CompanyDirectory();
        directory.addEmployee(engDirectory);
        directory.addEmployee(accDirectory);
        directory.showEmployeeDetails();
    }
}
```

Output :

```
100 Lokesh Sharma Pro Developer
101 Vinay Sharma Developer
200 Kushagra Garg SEO Manager
201 Vikram Sharma  Kushagra's Manager
```

When to use Composite Design Pattern?

Composite Pattern should be used when clients need to ignore the difference between compositions of objects and individual objects. If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice, it is less complex in this situation to treat primitives and composites as homogeneous.

1. Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.

2. Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

When not to use Composite Design Pattern?

1. Composite Design Pattern makes it harder to restrict the type of components of a composite. So it should not be used when you don't want to represent a full or partial hierarchy of objects.
2. Composite Design Pattern can make the design overly general. It makes harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. Instead you'll have to use run-time checks.

Source

https://www.geeksforgeeks.org/composite-design-pattern/

Chapter 10

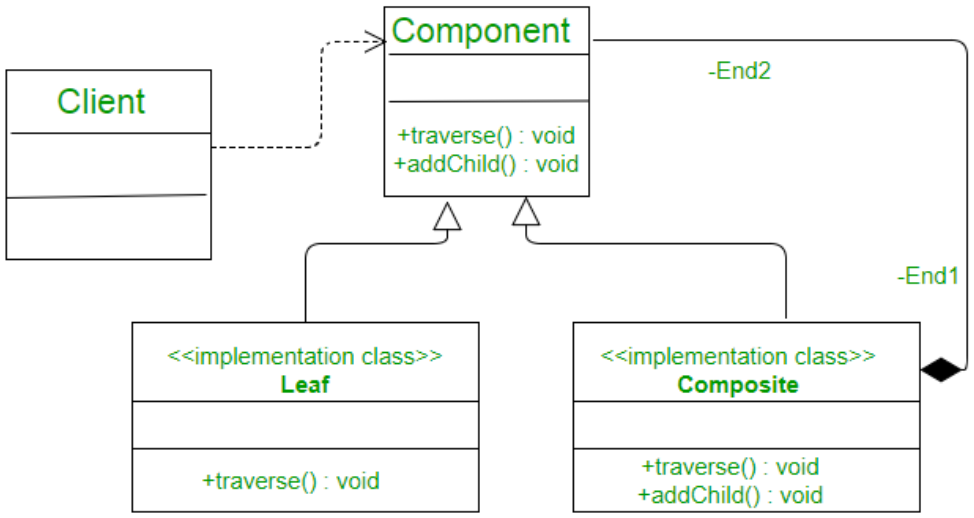
Composite Design Pattern in C++

Composite Design Pattern in C++ - GeeksforGeeks

Prerequisite : Composite Design Pattern

Composite pattern is one of the most widely used patterns in the industry and addresses a very significant and subtle problem. It is used whenever the user wants to treat the individual object in the same way as the collection of those individual objects for e.g you might want to consider a page from the copy as same as the whole copy which is basically a collection of the pages or if you want to create a hierarchy of something where you might want to consider the whole thing as the object .
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

In the case of photoshop where we draw many individual objects and then those objects compose a whole unique object and you might want to apply some operation on the whole object instead of the each of the individual objects.



Here in this diagram, as you can see both composite and Leaf implements Component diagram, thus allowing the same operation on both objects but the important part is Composite Class which also contain the Component Objects which is symbolized by the black diamond indicating composition relationship between Composite and Component class.

Then how to design our classes to accommodate such scenarios. We will try to understand it by implementing our copy example. Say you have to create a page which has operations like add, delete, remove and also a copy which will have the same operations as the individual pages.

Such situation is the best dealt with the composite pattern.

```
 // CPP program to illustrate
// Composite design pattern
#include <iostream>
#include <vector>
using namespace std;

class PageObject {
public:
    virtual void Add(PageObject a)
    {
    }
    virtual void Remove()
    {
    }
    virtual void Delete(PageObject a)
    {
    }
};
```

```
class Page : public PageObject {
public:
    void Add(PageObject a)
    {
        cout << "something is added to the page" << endl;
    }
    void Remove()
    {
        cout << "soemthing is removed from the page" << endl;
    }
    void Delete(PageObject a)
    {
        cout << "soemthing is deleted from page " << endl;
    }
};

class Copy : public PageObject {
    vector<PageObject> copyPages;

public:
    void AddElement(PageObject a)
    {
        copyPages.push_back(a);
    }

    void Add(PageObject a)
    {
        cout << "something is added to the copy" << endl;
    }
    void Remove()
    {
        cout << "something is removed from the copy" << endl;
    }
    void Delete(PageObject a)
    {
        cout << "something is deleted from the copy";
    }
};

int main()
{
    Page a;
    Page b;
    Copy allcopy;
    allcopy.AddElement(a);
    allcopy.AddElement(b);
```

```
    allcopy.Add(a);
    a.Add(b);

    allcopy.Remove();
    b.Remove();

    return 0;
}
```

```
something is added to the copy
something is added to the page
something is removed from the copy
soemthing is removed from the page
```

Now the same operation that can be applied to an individual object and can also be applied to the collection of those individual object makes it very easy to work with a larger object which is made of the smaller independent objects.
The most notable example of the composite pattern is in any UI toolkit. Consider the case of UI elements where each UI top-level UI element is composed of many smaller independent lower-level UI elements and the both the top and lower level UI element respond to same events and actions.

References :
1. Composite Pattern c#
2. Composite Pattern

Source

https://www.geeksforgeeks.org/composite-pattern-cpp/

Chapter 11

Curiously recurring template pattern (CRTP)

Curiously recurring template pattern (CRTP) - GeeksforGeeks

Background:

It is recommended to refer Virtual Functions and Runtime Polymorphism as a prerequisite of this. Below is an example program to demonstrate run time polymorphism.

```
 // A simple C++ program to demonstrate run-time
// polymorphism
#include <iostream>
#include <chrono>
using namespace std;

typedef std::chrono::high_resolution_clock Clock;

// To store dimensions of an image
class Dimension
{
public:
    Dimension(int _X, int _Y) {mX = _X;  mY = _Y; }
private:
    int mX, mY;
};

// Base class for all image types
class Image
{
public:
    virtual void Draw() = 0;
    virtual Dimension GetDimensionInPixels() = 0;
```

```
protected:
    int dimensionX;
    int dimensionY;
};

// For Tiff Images
class TiffImage : public Image
{
public:
    void Draw() { }
    Dimension GetDimensionInPixels() {
        return Dimension(dimensionX, dimensionY);
    }
};

// There can be more derived classes like PngImage,
// BitmapImage, etc

// Driver code that calls virtual function
int main()
{
    // An image type
    Image* pImage = new TiffImage;

    // Store time before virtual function calls
    auto then = Clock::now();

    // Call Draw 1000 times to make sure performance
    // is visible
    for (int i = 0; i < 1000; ++i)
        pImage->Draw();

    // Store time after virtual function calls
    auto now = Clock::now();

    cout << "Time taken: "
         << std::chrono::duration_cast
           <std::chrono::nanoseconds>(now - then).count()
         << " nanoseconds" << endl;

    return 0;
}
```

Output :

```
Time taken: 2613 nanoseconds
```

See this for above result.

When a method is declared virtual, compiler secretly does two things for us:

1. Defines a VPtr in first 4 bytes of the class object
2. Inserts code in constructor to initialize VPtr to point to the VTable

What are VTable and VPtr?
When a method is declared virtual in a class, compiler creates a virtual table (aka VTable) and stores addresses of virtual methods in that table. A virtual pointer (aka VPtr) is then created and initialized to point to that VTable. A VTable is shared across all the instances of the class, i.e. compiler creates only one instance of VTable to be shared across all the objects of a class. Each instance of the class has its own version of VPtr. If we print the size of a class object containing at least one virtual method, the output will be sizeof(class data) + sizeof(VPtr).
Since address of virtual method is stored in VTable, VPtr can be manipulated to make calls to those virtual methods thereby violating principles of encapsulation. See below example:

```
 // A C++ program to demonstrate that we can directly
// manipulate VPtr. Note that this program is based
// on the assumption that compiler store vPtr in a
// specific way to achieve run-time polymorphism.
#include <iostream>
using namespace std;

#pragma pack(1)

// A base class with virtual function foo()
class CBase
{
public:
    virtual void foo() noexcept {
        cout << "CBase::Foo() called" << endl;
    }
protected:
    int mData;
};

// A derived class with its own implementation
// of foo()
class CDerived : public CBase
{
public:
    void foo() noexcept  {
        cout << "CDerived::Foo() called" << endl;
    }
private:
    char cChar;
};
```

```
// Driver code
int main()
{
    // A base type pointer pointing to derived
    CBase *pBase = new CDerived;

    // Accessing vPtr
    int* pVPtr = *(int**)pBase;

    // Calling virtual method
    ((void(*)())pVPtr[0])();

    // Changing vPtr
    delete pBase;
    pBase = new CBase;
    pVPtr = *(int**)pBase;

    // Calls method for new base object
    ((void(*)())pVPtr[0])();

    return 0;
}
```

Output :

```
CDerived::Foo() called
CBase::Foo() called
```

We are able to access vPtr and able to make calls to virtual methods through it. The memory representation of objects is explained here.

Is it wise to use virtual method?
As it can be seen, through base class pointer, call to derived class method is being dispatched. Everything seems to be working fine. Then what is the problem?
If a virtual routine is called many times (order of hundreds of thousands), it drops the performance of system, reason being each time the routine is called, its address needs to be resolved by looking through VTable using VPtr. Extra indirection (pointer dereference) for each call to a virtual method makes accessing VTable a costly operation and it is better to avoid it as much as we can.

Curiously Recurring Template Pattern (CRTP)

Usage of VPtr and VTable can be avoided altogether through Curiously Recurring Template Pattern (CRTP). CRTP is a design pattern in C++ in which a class X derives from a class template instantiation using X itself as template argument. More generally it is known as F-bound polymorphism.

```
 // Image program (similar to above) to demonstrate
// working of CRTP
#include <iostream>
#include <chrono>
using namespace std;

typedef std::chrono::high_resolution_clock Clock;

// To store dimensions of an image
class Dimension
{
public:
    Dimension(int _X, int _Y)
    {
        mX = _X;
        mY = _Y;
    }
private:
    int mX, mY;
};

// Base class for all image types. The template
// parameter T is used to know type of derived
// class pointed by pointer.
template <class T>
class Image
{
public:
    void Draw()
    {
        // Dispatch call to exact type
        static_cast<T*> (this)->Draw();
    }
    Dimension GetDimensionInPixels()
    {
        // Dispatch call to exact type
        static_cast<T*> (this)->GetDimensionInPixels();
    }

protected:
    int dimensionX, dimensionY;
};


// For Tiff Images
class TiffImage : public Image<TiffImage>
{
public:
```

```
    void Draw()
    {
        // Uncomment this to check method dispatch
        // cout << "TiffImage::Draw() called" << endl;
    }
    Dimension GetDimensionInPixels()
    {
        return Dimension(dimensionX, dimensionY);
    }
};

// There can be more derived classes like PngImage,
// BitmapImage, etc

// Driver code
int main()
{
    // An Image type pointer pointing to Tiffimage
    Image<TiffImage>* pImage = new TiffImage;

    // Store time before virtual function calls
    auto then = Clock::now();

    // Call Draw 1000 times to make sure performance
    // is visible
    for (int i = 0; i < 1000; ++i)
        pImage->Draw();

    // Store time after virtual function calls
    auto now = Clock::now();

    cout << "Time taken: "
         << std::chrono::duration_cast
         <std::chrono::nanoseconds>(now - then).count()
         << " nanoseconds" << endl;

    return 0;
}
```

Output :

```
Time taken: 732 nanoseconds
```

See this for above result.

Virtual method vs CRTP benchmark
The time taken while using virtual method was 2613 nanoseconds. This (small) performance gain from CRTP is because the use of a VTable dispatch has been circumvented. Please

note that the performance depends on a lot of factors like compiler used, operations performed by virtual methods. Performance numbers might differ in different runs, but (small) performance gain is expected from CRTP.
Note: If we print size of class in CRTP, it can bee seen that VPtr no longer reserves 4 bytes of memory.

```
cout << sizeof(Image) << endl;
```

Questions? Keep them coming. We would love to answer.

Reference(s)
https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Source

https://www.geeksforgeeks.org/curiously-recurring-template-pattern-crtp-2/

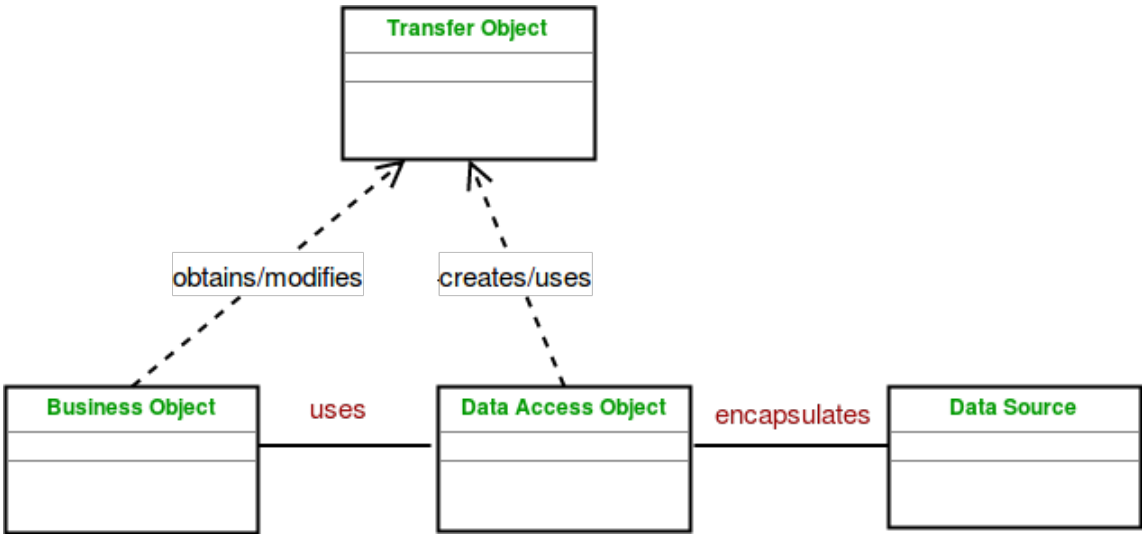
Chapter 12

Data Access Object Pattern

Data Access Object Pattern - GeeksforGeeks

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

UML Diagram Data Access Object Pattern



Design components

- **BusinessObject :** The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean or some other Java object in addition to a servlet or helper bean that accesses the data source.

- **DataAccessObject :** The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source.
- **DataSource :** This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system service or some kind of repository.
- **TransferObject :** This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

Let's see an example of Data Access Object Pattern.

```
 // Java program to illustrate
// Data Access Object Pattern
import java.util.List;
import java.util.ArrayList;
import java.util.List;

class Developer
{
    private String name;
    private int DeveloperId;

    Developer(String name, int DeveloperId)
    {
        this.name = name;
        this.DeveloperId = DeveloperId;
    }

    public String getName()
    {
    return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public int getDeveloperId()
    {
    return DeveloperId;
    }

    public void setDeveloperId(int DeveloperId)
    {
    this.DeveloperId = DeveloperId;
```

```
    }
}

interface DeveloperDao
{
    public List<Developer> getAllDevelopers();
    public Developer getDeveloper(int DeveloperId);
    public void updateDeveloper(Developer Developer);
    public void deleteDeveloper(Developer Developer);
}

class DeveloperDaoImpl implements DeveloperDao
{
    List<Developer> Developers;

    public DeveloperDaoImpl()
    {
        Developers = new ArrayList<Developer>();
        Developer Developer1 = new Developer("Kushagra",0);
        Developer Developer2 = new Developer("Vikram",1);
        Developers.add(Developer1);
        Developers.add(Developer2);
    }
    @Override
    public void deleteDeveloper(Developer Developer)
    {
        Developers.remove(Developer.getDeveloperId());
        System.out.println("DeveloperId " + Developer.getDeveloperId()
        + ", deleted from database");
    }
    @Override
    public List<Developer> getAllDevelopers()
    {
        return Developers;
    }

    @Override
    public Developer getDeveloper(int DeveloperId)
    {
        return Developers.get(DeveloperId);
    }

    @Override
    public void updateDeveloper(Developer Developer)
    {
        Developers.get(Developer.getDeveloperId()).setName(Developer.getName());
        System.out.println("DeveloperId " + Developer.getDeveloperId()
        + ", updated in the database");
```

```
    }
}

class DaoPatternDemo
{
    public static void main(String[] args)
    {
        DeveloperDao DeveloperDao = new DeveloperDaoImpl();

        for (Developer Developer : DeveloperDao.getAllDevelopers())
        {
            System.out.println("DeveloperId : " + Developer.getDeveloperId()
            + ", Name : " + Developer.getName());
        }

        Developer Developer =DeveloperDao.getAllDevelopers().get(0);
        Developer.setName("Lokesh");
        DeveloperDao.updateDeveloper(Developer);

        DeveloperDao.getDeveloper(0);
        System.out.println("DeveloperId : " + Developer.getDeveloperId()
        + ", Name : " + Developer.getName());
    }
}
```

Output:

```
DeveloperId : 0, Name : Kushagra
DeveloperId : 1, Name : Vikram
DeveloperId 0, updated in the database
DeveloperId : 0, Name : Lokesh
```

Advantages :

- The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that can but should not know anything of each other, and which can be expected to evolve frequently and independently.
- if we need to change the underlying persistence mechanism we only have to change the DAO layer, and not all the places in the domain logic where the DAO layer is used from.

Disadvantages :

- Potential disadvantages of using DAO is leaky abstraction, code duplication, and abstraction inversion.

Reference :
http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Source

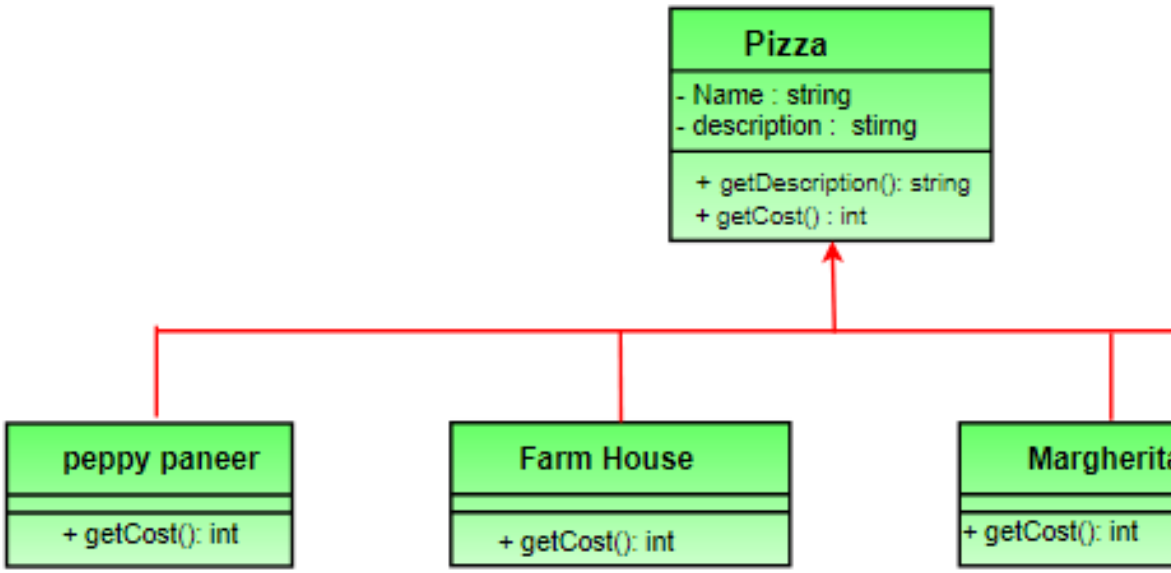
https://www.geeksforgeeks.org/data-access-object-pattern/

Chapter 13

Decorator Pattern | Set 1 (Background)

Decorator Pattern | Set 1 (Background) - GeeksforGeeks

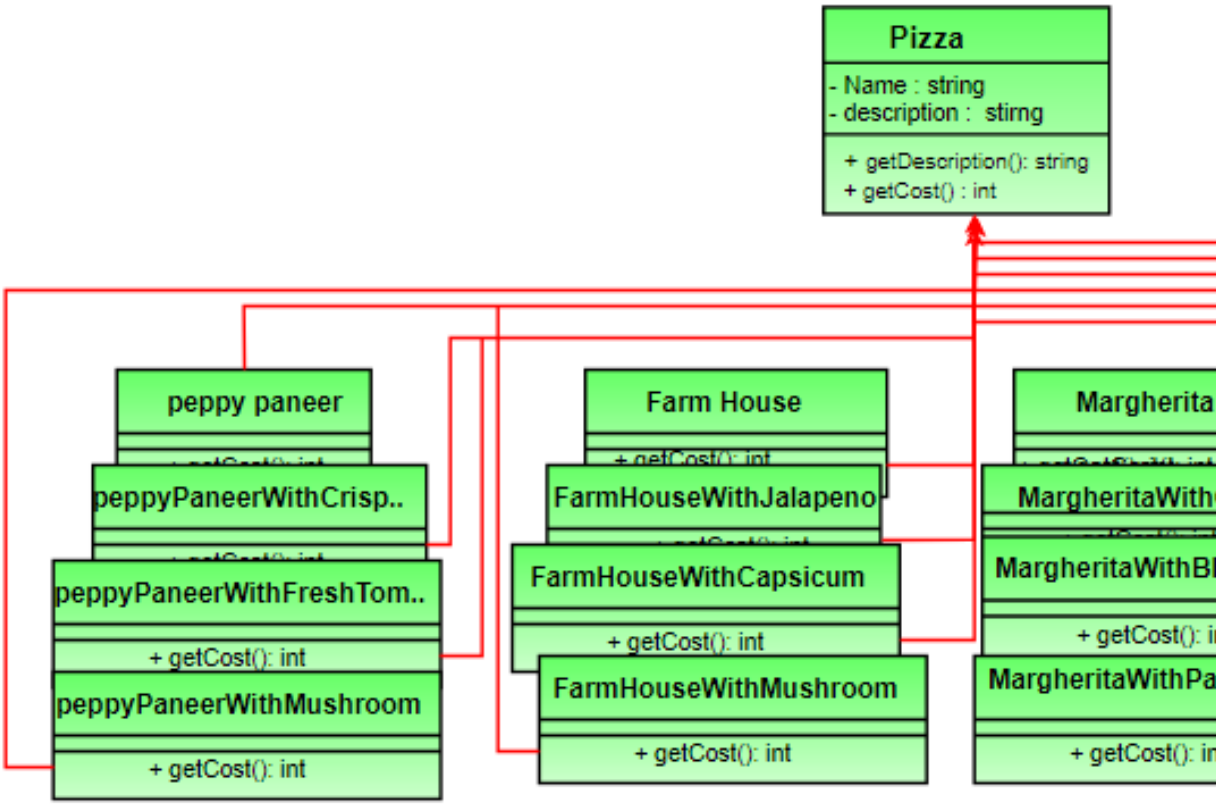
To understand decorator pattern let us consider a scenario inspired from the book "Head First Design Pattern". Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita and Chicken Fiesta. Initially we just use inheritance and abstract out the common functionality in a **Pizza** class.



Each pizza has different cost. We have overridden the getCost() in the subclasses to find the appropriate cost. Now suppose a new requirement, in addition to a pizza, customer can also ask for several toppings such as Fresh Tomato, Paneer, Jalapeno, Capsicum, Barbeque, etc. Let us think about for sometime that how do we accommodate changes in the above classes so that customer can choose pizza with toppings and we get the total cost of pizza and toppings the customer chooses.

Let us look at various options.

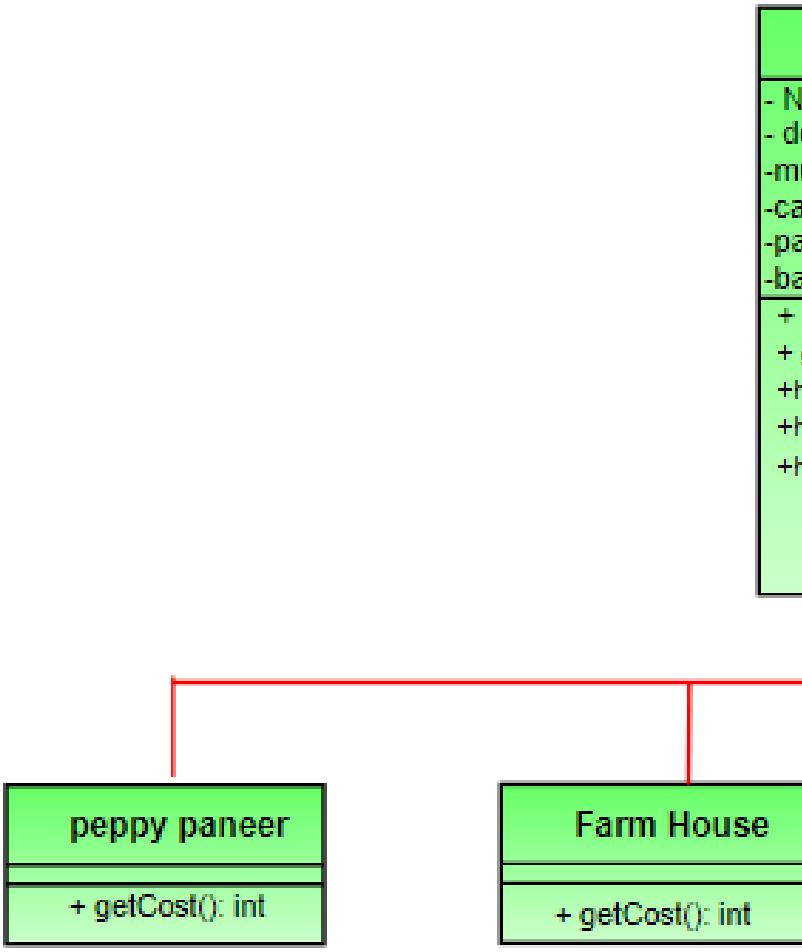
Option 1
Create a new subclass for every topping with a pizza. The class diagram would look



like:

This looks very complex. There are way too many classes and is a maintenance nightmare. Also if we want to add a new topping or pizza we have to add so many classes. This is obviously very bad design.

Option 2:
Let's add instance variables to pizza base class to represent whether or not each pizza has a



topping. The class diagram would look like:

The getCost() of superclass calculates the costs for all the toppings while the one in the subclass adds the cost of that specific pizza.

```
// Sample getCost() in super class
public int getCost()
{
    int totalToppingsCost = 0;
    if (hasJalapeno() )
        totalToppingsCost += jalapenoCost;
    if (hasCapsicum() )
        totalToppingsCost += capsicumCost;
```

```
    // similarly for other toppings
    return totalToppingsCost;
}


// Sample getCost() in subclass
public int getCost()
{
    // 100 for Margherita and super.getCost()
    // for toppings.
    return super.getCost() + 100;
}
```

This design looks good at first but lets take a look at the problems associated with it.

- Price changes in toppings will lead to alteration in the existing code.
- New toppings will force us to add new methods and alter getCost() method in superclass.
- For some pizzas, some toppings may not be appropriate yet the subclass inherits them.
- What if customer wants double capsicum or double cheeseburst?

In short our design violates one of the most popular design principle – **The Open-Closed Principle** which states that classes should be open for extension and closed for modification.

In the next set, we will be introducing Decorator Pattern and apply it to above above problem.

References: Head First Design Patterns(Book).

Source

https://www.geeksforgeeks.org/decorator-pattern/

Chapter 14

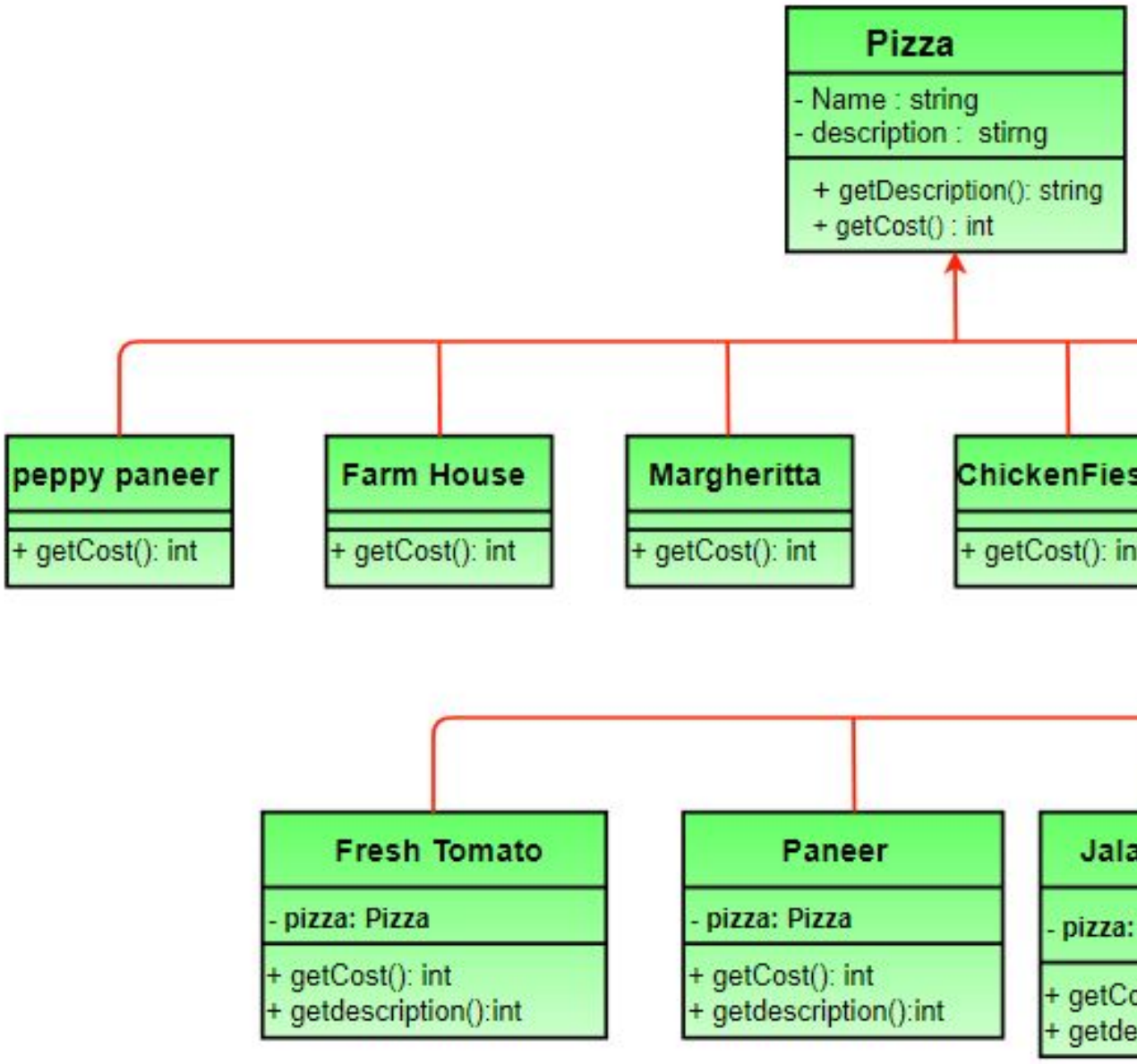
Decorator Pattern | Set 3 (Coding the Design)

Decorator Pattern | Set 3 (Coding the Design) - GeeksforGeeks

We have discussed Pizza design problem and different naive approaches to solve it in set 1. We have also introduced Decorator pattern in Set 2.

In this article, design and implementation of decorator pattern for Pizza problem is discussed. It is highly recommended that you try it yourself first.

The new class diagram (Click on the picture to see it clearly)



- **Pizza** acts as our abstract component class.
- There are four concrete components namely **PeppyPaneer** , **FarmHouse**, **Margherita**, **ChickenFiesta**.
- **ToppingsDecorator**is our abstract decorator and **FreshTomato** , **Paneer**, **Jalapeno**, **Barbeque** are concrete decorators.

Below is the java implementation of above design.

```
 // Java program to demonstrate Decorator
// pattern

// Abstract Pizza class (All classes extend
// from this)
abstract class Pizza
{
    // it is an abstract pizza
    String description = "Unkknown Pizza";

    public String getDescription()
    {
        return description;
    }

    public abstract int getCost();
}

// The decorator class :  It extends Pizza to be
// interchangable with it topings decorator can
// also be implemented as an interface
abstract class ToppingsDecorator extends Pizza
{
    public abstract String getDescription();
}

// Concrete pizza classes
class PeppyPaneer extends Pizza
{
    public PeppyPaneer() { description = "PeppyPaneer"; }
    public int getCost() {  return 100; }
}
class FarmHouse extends Pizza
{
    public FarmHouse() {  description = "FarmHouse"; }
    public int getCost() { return 200; }
}
class Margherita extends Pizza
{
    public Margherita()  { description = "Margherita"; }
    public int getCost() { return 100;  }
}
class ChickenFiesta extends Pizza
{
    public ChickenFiesta() { description = "ChickenFiesta";}
    public int getCost() { return 200; }
```

```
}
class SimplePizza extends Pizza
{
public SimplePizza() { description = "SimplePizza"; }
public int getCost() {  return 50;  }
}

// Concrete toppings classes
class FreshTomato extends ToppingsDecorator
{
    // we need a reference to obj we are decorating
    Pizza pizza;

    public FreshTomato(Pizza pizza) { this.pizza = pizza; }
    public String getDescription() {
        return pizza.getDescription() + ", Fresh Tomato ";
    }
    public int getCost() { return 40 + pizza.getCost(); }
}
class Barbeque extends ToppingsDecorator
{
    Pizza pizza;
    public Barbeque(Pizza pizza) {  this.pizza = pizza;  }
    public String getDescription() {
        return pizza.getDescription() + ", Barbeque ";
    }
    public int getCost() { return 90 + pizza.getCost(); }
}
class Paneer extends ToppingsDecorator
{
    Pizza pizza;
    public Paneer(Pizza pizza)  {  this.pizza = pizza; }
    public String getDescription() {
        return pizza.getDescription() + ", Paneer ";
    }
    public int getCost()  {  return 70 + pizza.getCost(); }
}

// Other toppings can be coded in a similar way

// Driver class and method
class PizzaStore
{
    public static void main(String args[])
    {
        // create new margherita pizza
        Pizza pizza = new Margherita();
        System.out.println( pizza.getDescription() +
```

```
                        " Cost :" + pizza.getCost());

        // create new FarmHouse pizza
        Pizza pizza2 = new FarmHouse();

        // decorate it with freshtomato topping
        pizza2 = new FreshTomato(pizza2);

        //decorate it with paneer topping
        pizza2 = new Paneer(pizza2);

        System.out.println( pizza2.getDescription() +
                        " Cost :" + pizza2.getCost());
        Pizza pizza3 = new Barbeque(null);    //no specific pizza
        System.out.println( pizza3.getDescription() + "  Cost :" + pizza3.getCost());
   }
}
```

Output:

```
Margherita Cost :100
FarmHouse, Fresh Tomato , Paneer Cost :310
```

Notice how we can add/remove new pizzas and toppings with no alteration in previously tested code and toppings and pizzas are decoupled.

```
 // CPP program to demonstrate
// Decorator pattern
#include <iostream>
#include <string>
using namespace std;

// Component
class MilkShake
{
public:
    virtual string Serve() = 0;
    virtual float price() = 0;
};


// Concrete Component
class BaseMilkShake : public MilkShake
{
public:
    string Serve()
    {
```

```
        return "MilkShake";
    }

    float price()
    {
        return 30;
    }
};

// Decorator
class MilkShakeDecorator: public MilkShake
{
protected:
    MilkShake *m_MilkShake;
public:

    MilkShakeDecorator(MilkShake *baseMilkShake): m_MilkShake(baseMilkShake){}

    string Serve()
    {
        return m_MilkShake->Serve();
    }

    float price()
    {
        return m_MilkShake->price();
    }
};


// Concrete Decorator
class MangoMilkShake: public MilkShakeDecorator
{
public:
    MangoMilkShake(MilkShake *baseMilkShake): MilkShakeDecorator(baseMilkShake){}

    string Serve()
    {
        return m_MilkShake->Serve() + " decorated with Mango ";
    }
    float price()
    {
        return m_MilkShake->price() + 40;
    }
};


class VanillaMilkShake: public MilkShakeDecorator
```

```
{
public:
    VanillaMilkShake(MilkShake *baseMilkShake): MilkShakeDecorator(baseMilkShake){}

    string Serve()
    {
        return m_MilkShake->Serve() + " decorated with Vanilla ";
    }
    float price()
    {
        return m_MilkShake->price() + 80;
    }
};

int main()
{
  MilkShake *baseMilkShake = new BaseMilkShake();
  cout << "Basic Milk shake \n";
  cout << baseMilkShake -> Serve() << endl;
  cout << baseMilkShake -> price() << endl;

  MilkShake *decoratedMilkShake = new MangoMilkShake(baseMilkShake);
  cout << "Mango decorated Milk shake \n";
  cout << decoratedMilkShake -> Serve() << endl;
  cout << decoratedMilkShake -> price() << endl;

  delete decoratedMilkShake;

  decoratedMilkShake = new VanillaMilkShake(baseMilkShake);
  cout << "Vanilla decorated Milk shake \n";
  cout << decoratedMilkShake -> Serve() << endl;
  cout << decoratedMilkShake -> price() << endl;

 delete decoratedMilkShake;
 delete baseMilkShake;
  return 0;
}
```

Output:

```
Basic Milk shake
MilkShake
Price of MilkShake : 30
Mango decorated Milk shake
MilkShake decorated with Mango
Price of Mango MilkShake : 70
Vanilla decorated Milk shake
```

```
MilkShake decorated with Vanilla
Price of Vanilla MilkShake : 110
```

Source

https://www.geeksforgeeks.org/decorator-pattern-set-3-coding-the-design/

Chapter 15

Dependency Inversion Principle (SOLID)

Dependency Inversion Principle (SOLID) - GeeksforGeeks

Let's understand one of the key principles of SOLID principles group namely, Dependency inversion principle.

Dependency inversion principle is one of the principles on which most of the design patterns are build upon. Dependency inversion talks about the coupling between the different classes or modules. It focuses on the approach where the higher classes are not dependent on the lower classes instead depend upon the abstraction of the lower classes. The main motto of the dependency inversion is *Any higher classes should always depend upon the abstraction of the class rather than the detail*.

This aims to reduce the coupling between the classes is achieved by introducing abstraction between the layer, thus doesn't care about the real implementation.

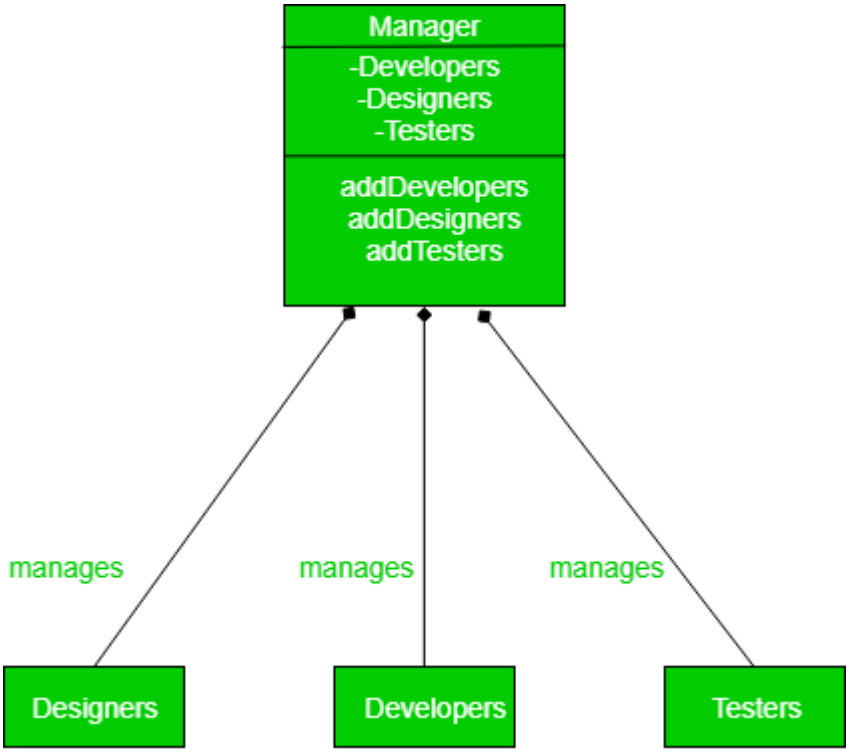
Let's understand the dependency inversion principle with an example. Say you are a manager having some persons as an employee of which some are developers and some are graphic designers and rest are testers.

Now let's see how a naive design would look without any dependency inversion and what are the loopholes in that design.

```
class Manager(object):
   def __init__(self):
       self.developers=[]
       self.designers=[]
       self.testers=[]

   def addDeveloper(self,dev):
       self.developers.append(dev)
```

```
    def addDesigners(self,design):
        self.designers.append(design)

    def addTesters(self,testers):
        self.testers.append(testers)


class Developer(object):
    def __init__(self):
        print "developer added"

class Designer(object):
    def __init__(self):
        print "designer added"

class Testers(object):
    def __init__(self):
        print "tester added"

if __name__ == "__main__":
    a=Manager()
    a.addDeveloper(Developer())
    a.addDesigners(Designer())


Output :
  developer added
  designer added
```

This can be easily be visualized by the following UML Diagram.



Now, let's look at the design loop holes in the source code :
First, you have exposed everything about the lower layer to the upper layer, thus abstraction is not mentioned. That means Manager must already know about the type of the workers that he can supervise.

Now if another type of worker comes under the manager lets say, QA (quality assurance), then the whole class needs to be rejigged. This is where dependency inversion principle pitch in.

Let's see how to solve the problem to the better extent using Dependency Inversion Principle.

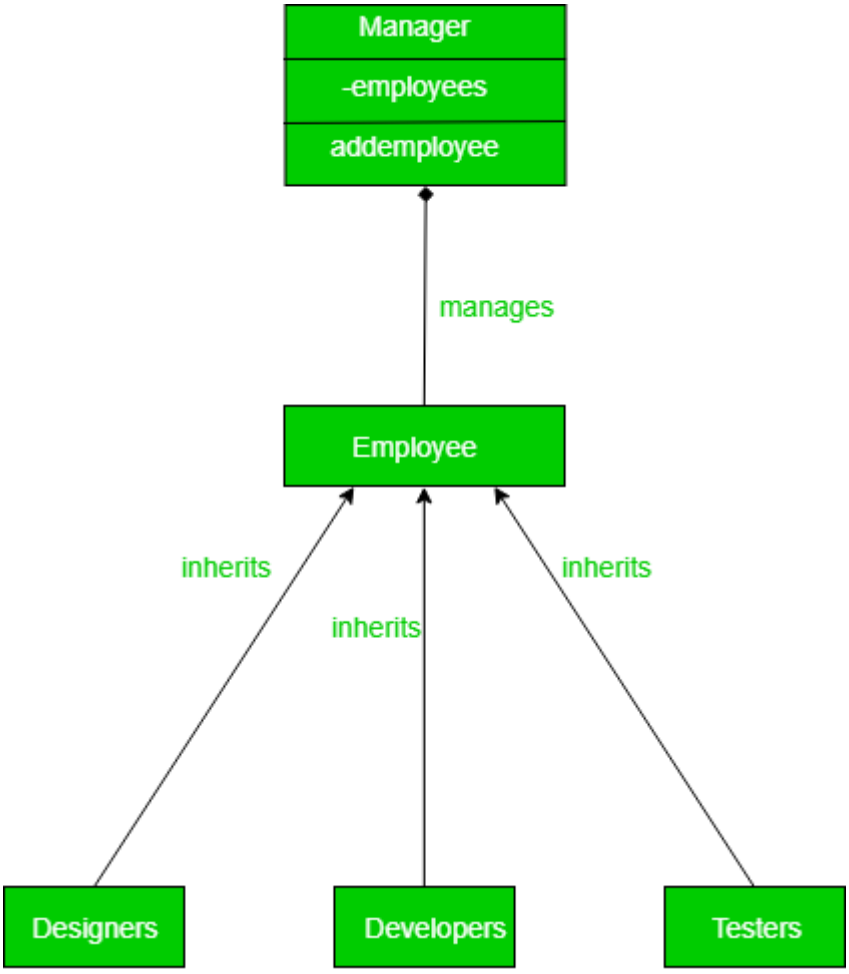
```
class Employee(object):
    def Work():
        pass

class Manager():
    def __init__(self):
        self.employees=[]
    def addEmployee(self,a):
        self.employees.append(a)

class Developer(Employee):
    def __init__(self):
        print "developer added"
```

```
    def Work():
        print "turning coffee into code"

class Designer(Employee):
    def __init__(self):
        print "designer added"
    def Work():
        print "turning lines to wireframes"

class Testers(Employee):
    def __init__(self):
        print "tester added"
    def Work():
        print "testing everything out there"

if __name__ == "__main__":
    a=Manager()
    a.addEmployee(Developer())
    a.addEmployee(Designer())


Output :
  developer added
  designer added
```

Now if any other kind of the employee is added it can be simply be added to Manager without making the manager explicitly aware of it. Now to add another class of employee we can simply call

```
class QA(Employee):
        def Work():
             print "testing everything out there"
a.add(QA())
```

The creation of the abstraction between different employees and Manager has resulted in very good looking design code which is easily maintainable and extendable.Please have a look into the UML diagram below.



In this code, the manager doesn't have an idea beforehand about all the type of workers that may come under him/her making the code truly decoupled. There are many design patterns where this is a core idea and other things are built upon it.

Source

https://www.geeksforgeeks.org/dependecy-inversion-principle-solid/

Chapter 16

Design Patterns | Set 1 (Introduction)

Design Patterns | Set 1 (Introduction) - GeeksforGeeks

Design pattern is a general reusable solution or template to a commonly occurring problem in software design. The patterns typically show relationships and interactions between classes or objects. The idea is to speed up the development process by providing tested, proven development paradigm.

Goal:
- Understand the problem and matching it with some pattern.
- Reusage of old interface or making the present design reusable for the future usage.

Example:
For example, in many real world situations we want to create only one instance of a class. For example, there can be only one active president of country at a time regardless of personal identity. This pattern is called Singleton pattern. Other software examples could be 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.

Types of Design Patterns
There are mainly three types of design patterns:
1. Creational
These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational 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.

Creational design patterns are Factory Method, Abstract Factory, Builder, Singleton, Object Pool and Prototype.

2. Structural

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

3. Behavioral
Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.

Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor

References:
https://sourcemaking.com/design_patterns
https://sourcemaking.com/design_patterns/singleton

This article is contributed by **Abhijit Saha**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : Sachin Araballi

Source

https://www.geeksforgeeks.org/design-patterns-set-1-introduction/

Chapter 17

Design Patterns | Set 2 (Factory Method)

Design Patterns | Set 2 (Factory Method) - GeeksforGeeks

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.

A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library(explained below) in a way such that it doesn't have tight coupling with the class hierarchy of the library.

What is meant when we talk about library and clients?
A library is something which is provided by some third party which exposes some public APIs and clients make calls to those public APIs to complete its task. A very simple example can be different kinds of Views provided by Android OS.

Why factory pattern?
Let us understand it with an example:

```
 // A design without factory pattern
#include <iostream>
using namespace std;

// Library classes
class Vehicle {
public:
    virtual void printVehicle() = 0;
};
class TwoWheeler : public Vehicle {
public:
```

```
    void printVehicle()  {
        cout << "I am two wheeler" << endl;
    }
};
class FourWheeler : public Vehicle {
    public:
    void printVehicle()  {
        cout << "I am four wheeler" << endl;
    }
};

// Client (or user) class
class Client {
public:
    Client(int type)  {

        // Client explicitly creates classes according to type
        if (type == 1)
            pVehicle = new TwoWheeler();
        else if (type == 2)
            pVehicle = new FourWheeler();
        else
            pVehicle = NULL;
    }

    ~Client()   {
        if (pVehicle)
        {
            delete[] pVehicle;
            pVehicle = NULL;
        }
    }

    Vehicle* getVehicle() {
        return pVehicle;
    }
private:
    Vehicle *pVehicle;
};

// Driver program
int main() {
    Client *pClient = new Client(1);
    Vehicle * pVehicle = pClient->getVehicle();
    pVehicle->printVehicle();
    return 0;
}
```

Output:

```
I am two wheeler
```

What is the problems with above design?
As you must have observed in the above example, Client creates objects of either TwoWheeler or FourWheeler based on some input during constructing its object.
Say, library introduces a new class ThreeWheeler to incorporate three wheeler vehicles also. What would happen? Client will end up chaining a new else if in the conditional ladder to create objects of ThreeWheeler. Which in turn will need Client to be recompiled. So, each time a new change is made at the library side, Client would need to make some corresponding changes at its end and recompile the code. Sounds bad? This is a very bad practice of design.

How to avoid the problem?
The answer is, create a static (or factory) method. Let us see below code.

```
 // C++ program to demonstrate factory method design pattern
#include <iostream>
using namespace std;

enum VehicleType {
    VT_TwoWheeler,    VT_ThreeWheeler,    VT_FourWheeler
};

// Library classes
class Vehicle {
public:
    virtual void printVehicle() = 0;
    static Vehicle* Create(VehicleType type);
};
class TwoWheeler : public Vehicle {
public:
    void printVehicle() {
        cout << "I am two wheeler" << endl;
    }
};
class ThreeWheeler : public Vehicle {
public:
    void printVehicle() {
        cout << "I am three wheeler" << endl;
    }
};
class FourWheeler : public Vehicle {
    public:
    void printVehicle() {
        cout << "I am four wheeler" << endl;
    }
};
```

```
// Factory method to create objects of different types.
// Change is required only in this function to create a new object type
Vehicle* Vehicle::Create(VehicleType type) {
    if (type == VT_TwoWheeler)
        return new TwoWheeler();
    else if (type == VT_ThreeWheeler)
        return new ThreeWheeler();
    else if (type == VT_FourWheeler)
        return new FourWheeler();
    else return NULL;
}

// Client class
class Client {
public:

    // Client doesn't explicitly create objects
    // but passes type to factory method "Create()"
    Client()
    {
        VehicleType type = VT_ThreeWheeler;
        pVehicle = Vehicle::Create(type);
    }
    ~Client() {
        if (pVehicle) {
            delete[] pVehicle;
            pVehicle = NULL;
        }
    }
    Vehicle* getVehicle()  {
        return pVehicle;
    }

private:
    Vehicle *pVehicle;
};

// Driver program
int main() {
    Client *pClient = new Client();
    Vehicle * pVehicle = pClient->getVehicle();
    pVehicle->printVehicle();
    return 0;
}
```

Output:

```
I am three wheeler
```

In the above example, we have totally decoupled the selection of type for object creation from Client. The library is now responsible to decide which object type to create based on an input. Client just needs to make call to library's factory Create method and pass the type it wants without worrying about the actual implementation of creation of objects.

Thanks to Rumplestiltskin for providing above explanation.

Other examples of Factory Method:

1. Say, in a 'Drawing' system, depending on user's input, different pictures like square, rectangle, circle can be drawn. Here we can use factory method to create instances depending on user's input. For adding new type of shape, no need to change client's code.
2. Another example: 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.

Thanks to Abhijit Saha providing the first 2 examples.

Source

https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/

Chapter 18

Design Scalable System like Foursquare

Design Scalable System like Foursquare - GeeksforGeeks

1- What is Foursquare?

As you know, if you decide to design scalable system, you should first the point that is the definition of the system. This means that you should be sure that what this system is. So firstly, we should explain the goal of the system. This is basically the system that presents locations to users based on nearby their locations. You can find nearby locations such as restaurants, hospitals, cafes and etc… In addition, you can search any location based on your demand. If you do this when you use this system, it should suggest best options that are closest to your location.

2- Identifying requirements

As we said before, the most important goal is presenting the most suitable places to you based on your searching query and location. There are three requirements that we need to define at first.

Functional Requirements
a-Users can register the system.
b-Users can login or log out the system.
c-Users can search a place such as a restaurant, theatre or etc..
d-Users can comment a place.
e-Users can like or dislike a place.
f-Users can review a place.
g-Users can add a place.
h-Users can add pictures to added places.
j-Users can update or delete a place if they added a place themselves.

Nonfunctional Requirements
a-Users should have a real-time experience.
b-As we know, the system should be under read-heavy. Minimum latency is one of the

most important parts of the system.

c-The system should be highly available.

d-The System should be reliable. This is important because nobody wants to lose their comments or places that they added.

e-Consistency can take a hit. This means that if users don't see the hottest review or places for a while, this should be ok.

Extended Requirements

a-The system should monitor.

b-The system should suggest best options as soon as possible. The way to save the data and the search and recommendation algorithms are very important.

3- Capacity Estimation

Let's assume there are totally 100 Million places in our system and 10.000 queries come to the system each day. We should keep in our mind that this system continues to grow each year and we can assume that the growth rate of the system equals to 10%. We have talked about the total capacity of the system for ten years. If you realized that system capacity is huge, we need to scale the system effectively. Notice that when we mention about the capacity, we should be careful about the optimum availability. Using 80% of each server is the most optimal use. To illustrate this, if the system capacity is 80 TB, we need to 100 TB for total capacity to reach optimum usability.

4- Database Design

As you know, we have two options which are SQL and NoSQL to select best database method. NoSQL is the best way to scale system easily. On the other hand, if we think the relations and constraints of SQL, scaling SQL databases is very difficult.

Place
PlaceID
AddedUserID
Name
Longitude
Latitude
Category
Description
AddedDate
Point
LikeCount
DislikeCount

User
UserID
UserName
RegisterionDate
LastLogin Date

UserComment
UserID
PlaceID
Like

Dislike
Comment
CommentDate

** Notice that we can create UserID and PlaceID through Key Generation Service. KGS ensures that each key is unique in the system. The second important point is that 8 byte is enough to keep placeID and userID. Additionally, we can keep longitude and latitude with 8 bytes.

5- API Estimation
We can choose one of those two methods, REST or SOAP. There are four main API for designing the system.

a-) AddPlace(api_dev_key, name, description, longitude, latitude, category): This API returns the HTTP response showing operation is a success or not.

b-) DeletePlace(api_dev_key, placeID): This API returns the HTTP response showing operation is a success or not.

c-) UpdatePlace(api_dev_key, placeID, name = null, description = null, longitude = null, latitude = null, category = null)

d-) GetPlace(api_dev_key, search_query, maximum_return_count, radius, userlongitude, userlatitude, categoryfilter = null, sortby, page_token): return JSON including the list of most suitable places with their name, like count, dislike count, explanation, location, thumbnail and etc…

6- System Design

You know how important indexing is in large scalable systems. There are no many updating operations in our system. If we select the indexed pieces carefully, this system responds faster. Indexing is not recommended for operations with too many updates. Our search query should focus the user's and places' locations so the way to store data is so important. There are three ways to store data which are SQL, Grid, Dynamic sizable grid.

SQL:

SQL is the simple solution. If you decide to use SQL, you need to index based on placeID, longitude, and latitude. This method is not effective because there is two source to get the data which are longitude and latitude. All these two sources are indexed sources. This is a phase that reduces the response time because both of them return a response and these coming responses combine for getting an optimum result.
select * from Places where placeLongitude between userLongitude – Radius and userLongitude + Radius and placeLatitude between userLatitude – Radius and userLatitude + Radius order by (selected ranking method) desc.

Grid:

We can use a grid to solve performance problems. We can divide the whole world into smaller grids. This ensures that we can only deal with the few amount of grids. Thanks to this method, we just only focus on the user location grid and its neighbour grids. A map is the best choice to use a grid. Key is the gridID. Value is the whole places in this grid.
Select * from Places where Latitude between userLatitude – Radius and userLatitude + Radius and GridID in (GridID1, GridID2, …)

There is a problem with using a grid. As you know, all grids may not have same places so we can use a dynamic sizable grid.
We can keep all index in memory. This provides system become closer to real-time experience.

Dynamic Sizable Grid:

A quadtree is a tree data structure in which each internal node has exactly four children. A quadtree is a good option to use a dynamic sizable grid. The way to walk about the child nodes of each grid can be provided by using doubly linked list. If we assume we want that each grid cannot have over the 200 places, when the limit is exceeded and division begins.

7- Data Partitioning

We can partition the system based on regionID or locationID. If we use regionID, the system may be non-uniform distributed. To illustrate this, It can be very difficult to fit popular regions or regions having more places. We can handle this problem by using consistent hashing. The second option to partition data is sharding based on locationID. If we use locationID for sharding, the system becomes more uniformly distributed.

8- Replication and Load Balancer

As we mentioned above, the replication process is a valuable process to provide high availability, high reliability, and real-time experience. It is recommended to have 3 or more replicas for each server and service. This ensures that system automatically becomes more reliable and available. We can keep same data onto three different resources and thanks to this process if one server dies, the system automatically continues to work replicas. One more advantage of replication is the system may continue to run at an update to the system.

In replication process, there is a master-slave relationship between the server and its replicas. A Master is for writing and reading operations; a slave is for reading operations.

Load balancing refers to distributing requests across a group of services and servers. When we have talked about the replication and sharding, an incoming request must be directed and this is done by a load balancer. We can use Round-Robin method to redirect incoming requests but there may be a problem in this case. Round-Robin stops sending requests to dead servers but Round-Robin cannot stop sending requests to a server exposed to heavy traffic. We can prepare intelligent Round-Robin algorithm to take away this problem. Additionally, we can use consistent hashing to redirect incoming requests. Consistent hashing ensures that system becomes more uniformly distributed.

The possible problem for sharding with a load balancer is how we rebuilt the system when this system dies. The possible approach is brute-force. This method is slow because we need to rebuilt whole the system from the beginning. We can eliminate this problem by reverse indexing. We can have another Index server that will hold all information about reverse indexing. Reverse index maps all places. We need to build a map and key is the serverID (QuadTree server) and value is all places.

9- Caching Mechanism

There are many caching methods that we can use but LRU is the good choice for this system. Additionally, 80%-20% rule is valid for caching mechanism. This explains that we need to memory capacity to 20% of daily data. Moreover, as we explained before, keeping indexes in memory provides the system becomes more performance.

10- Ranking Mechanism
We can have various ranking mechanisms which are most popular, most relevant, newest and etc… As you know, we can have many servers and we need to get data from these servers. Because of this reason, we need to aggregate function to combine all these data to obtain the most desirable solution.

```
 // Java Program to illustrate the Design
public class Server {
    ArrayList<Machine> machines = new ArrayList<Machine>();
}
public class Machine {
    public ArrayList<Place> places = new ArrayList<Place>();
}
public class Place {

    public Place(int ID, int machineID)
    {
        this.ID = ID;
        this.machineID = machineID;
    }
    private int ID;
    private int machineID;
    private String information;
    private Server server = new Server();

    public String getInfo()
    {
        return info;
    }

    public void setInformation(String information)
    {
        this.information = information;
    }

    public int getID()
    {
        return ID;
    }

    public int getMachineID()
    {
        return machineID;
    }

    public void addPlace(Place place, machineID)
    {
        for (Machine m : server.machines) {
```

```
            if (m.machineID == machineID) {
                m.places.add(place);,
            }
        }
    }

    public void deletePlace(int placeID, int machineID)
    {
        for (Machine m : server.machines) {
            if (m.machineID == machineID) {
                m.places.remove(placeID);
            }
        }
    }

    public machine lookUpMachine(int machineID)
    {
        for (Machine m : server.machines) {
            if (m.machineID == machineID) {
                return m;
            }
            return null;
        }
    }

    public Place lookUpPlace(int machineID, int ID)
    {
        for (Machine m : server.machines) {
            if (m.machineID == machineID) {
                for (Place p : m.places) {
                    if (p.ID == ID) {
                        return p;
                    }
                }
            }
        }
        return null;
    }
}
```

Reference: gorkemgenc.com

Source

https://www.geeksforgeeks.org/design-scalable-system-like-foursquare/

Chapter 19

Design Scalable System like Instagram

Design Scalable System like Instagram - GeeksforGeeks

1- WHAT IS INSTAGRAM?

Instagram is one of the best social media platforms in today's world. Most of the people use the Instagram every day effectively and continuously. This causes that Instagram is under heavy-traffic. As you know, heavy-traffic means that there are too many incoming requests to the system and system should respond all these requests under reliability, availability and minimum latency.

As you know Instagram is a social media platform that ensures users to upload, share, view, comment pictures via this platform. Additionally, you can follow other users. If you are using Instagram, you can realize that user's timeline is one the best and important topic for Instagram because every user meet this system in its timeline and timeline creation time is so important under this conditions.

2- REQUIREMENTS AND GOALS OF THE SYSTEM

As you know, if you need to design your design carefully, you should focus three requirement topics which are functional, nonfunctional, extended requirements.

a-) Functional Requirements

– Users can register the system.
– Users can log in and log out the system.
– Users can share, download, view pictures of their pictures or other user's pictures.
– Users can follow other users.
– Users can share (upload) pictures when they register and log into the system.
– Users can view (download) pictures when other users allow that their pictures can be viewed publicly.
– Users can view hottest pictures in their timeline.
– Users can search pictures based on titles.

b-) Nonfunctional Requirements

– The system should be highly available.
– The system should be highly reliable. As we said, users can upload pictures and all data of users should not be lost.
– The system can work based on CAP Theorem. If we talk about a system like Instagram. Transaction operations are not as important as financial operations. This means that consistency can take time. This is sufficient if the system will be consistent within a certain period of time. Availability is more important than consistency so a user may not see photo for a while, this is fine.

As we said, timeline creation process is one of the hardest and important points of the Instagram design. This is good if the optimum time for creation timeline is 250 ms.

c-) Extended Requirements

– The system should monitor.
– The creation of timeline needs to effective algorithms for decreasing time.
– We can realize that this system is read-heavy so we should focus on the uploading and present pictures, so reliability and minimum latency are two main points of consideration.

Note: Instagram is a huge system and I didn't deal with the comment process, recommendations and tags. Recommendation systems are another important system that it should be designed carefully.

3- CAPACITY AND ESTIMATION

When we talk about the define capacity and estimation we should think the future of the system. To illustrate this, we can deal with the data that collect up to 5 or 10 years. It helps us to scale easily. Let's assume the total user count of Instagram is 500 Million and the daily user is 10 Million. If we assume that each user uploads 2 picture in each day, daily incoming picture count is 20 Million. Notice that we can think the average size of the picture is 500 KB so total required space for 1 day is 20 Million * 500 KB = 10 * 10^12 = 10 TB. Total space required for 5 years can easily be estimated. Note that, this capacity doesn't contain replicate data. Additionally, we should just only use until 80% of total existing capacity.

4- API DESIGN

We can use REST or SOAP to server our APIs. Basically, there are three important API of our system.

1-) *UploadPicture (api_dev_key, picture, title, picture_description, tags[], picture_details)*
UploadPicture base on the uploading picture. api_dev_key is the API developer key of a registered account. We can eliminate hacker attacks with api_dev_key. This API returns HTTP response. (202 accepted if success)

2-) *DownloadPicture (api_dev_key, search_query, user_location, maximum_video_count, page_token)*
Return JSON containing information about the list of pictures. Each picture resource will have a picture title, creation date, like count, dislike count, total view count.

3-) *DeletePicture (api_dev_key, pictureID)*
Return HTTP response if success.

**There are a lot of another APIs to design Instagram, however, these three APIs are more important than the others.

5- DATABASE SCHEMA

As you know, we have talked about the pictures and users basically. We have to decide whether to use SQL or NoSQL before defining database tables. We can use RDBMS to keep data but as you know, scale process of a traditional database system is hard when we decide to keep data o a traditional database system. On the other hand, if we use NoSQL, we can scale system easily. There are three tables to store data;

User
- UserID : Int
- UserName : Nvarchar
- UserRealName : Nvarchar
- UserSurname : Nvarchar
- Mail : Nvarchar
- BirthDate : DateTime
- RegisterDate : DateTime
- LastLoginDate: DateTime

Picture
- PictureID : Int
- UserID : Int
- PicturePath : Nvarchar
- PictureLatitude : Int
- PictureLongitude : Int
- CreationDate : DateTime

UserFollow
- UserID1 : Int
- UserID2 : Int

** If we choose the NoSQL to keep data, we need to add a new table system. (PictureUser)

** We can store photos in S3 or HDFS.

** We can store all information about pictures with a key-value pair like Redis. Key is pictureID, a value is other information about the picture. (For NoSQL)

** We can store all information about users with a key-value pair like Redis. Key is userID, a value is other information about the user. (For NoSQL)

** We can use Cassandra, column-based data storage, to save follow-up of users.

Note: A lot of NoSQL database supports replication.

Note: We need to have an index on PictureID and CreationDate because we need to get hottest pictures.

6- COMPONENT DESIGN

We can realize that uploading and downloading operation are not same. Uploading operation is slower than downloading operation because uploading operation just based on disk. On the other hand, read operation could be faster if we are using a cache.

If a user tries to upload a picture, he/she can consume all the connections. This causes to when uploading operation continues, the system may not respond to read operation. If we divide uploading and downloading operations into two separate services, then we can handle this bottleneck. Notice that, web servers have connection limits at any time and we need

to focus on this point. Notice that, separating of uploading and downloading operations ensure that system can be more scalable and optimize.

7- HIGH-LEVEL SYSTEM DESIGN

If we are designing a system, the basic concepts we need are;

- Client
- Services
- Web server
- Application server
- Picture Storage
- Database
- Caching
- Replication
- Redundancy
- Load balancing
- Sharding

There are two separate services in this system, which are upload image and download image. Picture storage is used to keep pictures. A database is used to save all information about users and pictures. (metadata). When a request comes to the system, the first to meet request is the web servers. Web servers redirect an incoming request to application servers.

8- REPLICATION AND REDUNDANCY

Replication is a very important concept to provide availability and reliability. As we said, Instagram should ensure that any files cannot be lost. Replication is a very important concept to handle a failure of services or servers. Replication and redundancy basically mean the copy of services or servers. We can replicate database servers, web servers, application servers, image storages and etc.. Actually we can replicate all parts of the system. Notice that replication also helps system to decrease response time. You imagine, if we divide incoming requests into more resources rather than one resource, the system can easily meet all incoming requests. In addition, the optimum number of a replica to each resource is 3 or more. Thanks to replications, if any server dies, the system continues to respond via secondary resource.

9- DATA SHARDING

As you know, sharding is a very important concept which helps system to keep data into different resources according to sharding process. There can be two sharding procedure to use. First is partitioning based on UserID and second is partitioning based on PhotoID.

Partitioning based on UserID: We can divide incoming requests based on UserID. We will find the shard number by UserID % number of shards. Conditioning with shard based on UserID causes to problems. First is that system may be non-uniform distributed and second is if a user is more active than the other user, the data of this user may not be fitted into one resource. Another possible problem is handling the PictureID creation. PictureID should be unique in the system, so each shard needs to have its creation policy.
Partitioning based on PictureID: We can divide incoming requests based on PictureID. We will find the shard number by PictureID% number of shards. We can handle the non-uniform distribution problem and popular user problem. We can easily create PhotoID with Key

Generation Service. Key Generation Service creates unique identifiers at first then serve this unique identifier to incoming pictures. This helps us to handle PhotoID problem.

10- CACHING

Cache memory is a crucial part of reading data faster. Cache memory usage can base on 80-20 rule. This means that cache capacity is the 20% of the daily data size. We can use LRU cache policy (Least Recently Used).

-CDN: CDN, Content Delivery Network is for distributed file cache servers. We can usage CDN for keeping pictures.
-Memcache / Redis: Keep metadata in the cache with Memcache or Redis.

11- LOAD BALANCING

Load balancer allows incoming requests to be redirected to resources according to certain criteria. We can use load balancer at every layer of the system.

– Between requests and web servers.
– Between web servers and application servers.
– Between application servers and databases
– Between application servers and image storages.
– Between application servers and cache databases.
– We can use Round Robin method for the load balancer. Round Robin method prevents requests from going to dead servers but Round Robin method doesn't deal with the situation that any server is under heavy-traffic. We can modify Round Robin method to be a more intelligent method to handle this problem.

12- DESIGN CONSIDERATION

Notice that we need to get the popular, latest and relevant photos of other people that we follow. We can use a pre-generate timeline to decrease latency because you image that system will fetch all friends of us firstly then fetch all pictures of our friends. After that, the system will combine all pictures based on creation time, like count or other properties. This action can take time and causes to late response. Pre-generate timeline keeps users' timelines into a table previously. Thanks to the pre-generate timeline, the system can serve them without the hassle of processing when they need to. What needs to be discussed here is what to do when new data comes in. There are three important methods we can mention,

– Pull: The client asks if there are any changes at regular intervals. This creates some problems. To illustrate this, we may not get new data when we use the system. Another problem is most of the type, the client can encounter with the empty response.

– Push: In push method, a server can push new data to clients as soon as it is available. Long Polling is one the best methods to use this method efficiently. Long polling is a method that there is an open connection between the client and the server, and if any change occurs about data, server return response to the client as soon as possible. This method may cause a problem for users that follow a lot of users.

– Hybrid: Hybrid method is a combination of pull and push methods. Push method is for users that follow few users and pull method is for users that follow a lot of users.

```
// Java Program to explain the design
```

```
public class Server{
    ArrayList<Machine> machines = new ArrayList<Machine>();
}
public class Storage{
    ArrayList<StorageMachine> machines = new ArrayList<StorageMachine>();
}
public class Machine{
    public ArrayList<User> users = new ArrayList<User>();
    public int machineID;
}
public class StorageMachine{
    public ArrayList<Picture> pictures = new ArrayList<Picture>();
    public int machineID;
}
public class User{
    private ArrayList<Integer> friends;
    private ArrayList<Integer> pictures;
    private int userID;
    private int machineID;
    private String information;
    private Server server = new Server();
    private Storage storage = new Storage();

    public User(int userID, int machineID){
        this.userID = userID;
        this.machineID = machineID;
pictures = new ArrayList<Integer>();
friends = new ArrayList<Integer>();
    }

    public String getInformation() {
        return information;
    }

    public void setInformation(String information){
        this.information = information;
    }

    public getID(){
        return userID;
    }

    public int getMachineID(){
        return machineID;
    }

    public void addFriend(int id){
```

```
    friends.add(i);
}

public void addPicture(int id){
    pictures.add(i);
}

public int[] getFriends(){
    int[] temp = new int[friends.size()];
    for(int i=0; i<temp.length; i++){
        temp[i] = friends.get(i);
    }
    return temp;
}

public int[] getPictures(){
    int[] temp = new int[pictures.size()];
    for(int i=0; i<temp.length; i++){
        temp[i] = pictures.get(i);
    }
    return temp;
}

public User lookUpFriend(int machineID, int ID){
    for(Machine m : server.machine){
        if(m.machineID  = machineID){
            for(User user : m.users){
                if(user.userID = ID){
                    return user;
                }
            }
        }
    }
    return null;
}

public Picture lookUpPicture(int machineID, int ID){
    for(StorageMachine m : storage.machine){
        if(m.machineID  = machineID){
            for(Picture picture : m.pictures){
                if(picture.pictureID = ID){
                    return picture;
                }
            }
        }
    }
    return null;
}
```

```
}
public class Picture{
    private int machineID;
    private int pictureID;
    private String photoPath;

    public Picture(int machineID, int pictureID, String photoPath){
        this.machineID = machineID;
        this.pictureID = pictureID;
        this.photoPath = photoPath;
    }

    public int getMachineID(){
        return machineID;
    }

    public void setMachineID(int machineID){
        this.machineID = machineID;
    }

    public int getPictureID(){
        return pictureID;
    }

    public int getPhotoPath(){
        return photoPath;
    }

    public void setPhotoPath(String photoPath){
        this.photoPath = photoPath;
    }
}
```

Reference: gorkemgenc.com

Source

https://www.geeksforgeeks.org/design-scalable-system-like-instagram/

Chapter 20

Design Video Sharing System Like Youtube

Design Video Sharing System Like Youtube - GeeksforGeeks

1- WHAT IS YOUTUBE?

Youtube is one of most popular video sharing system that presents for the people. Users can easily share, upload, view, rate, comment videos via the system. Users can register the system and use the system by their account. Youtube also is one the best systems that recommend videos to users according to their interest. This means that people can continue with the these suggested videos. Moreover, with having the special account on Youtube, users can follow other users or channels. Additionally, as we mentioned above, users can comment videos regardless of whether they are logged in or not.

2- REQUIREMENTS AND GOALS OF THE SYSTEM

As you know, Youtube is one the biggest system today's world, however, we can design the main features of the Youtube. Although there are a lot of different hard process to design Youtube, regardless of the recommendation system, channels, my favourite videos feature, most popular videos, detailed search mechanism, we can design the Youtube according to other features listed below,

Note that, there are three types of requirements which are functional requirements, nonfunctional requirements and extended requirements. We should define all required steps before starting the designing operation. Also, we should be careful to estimate capacity at the beginning of the design because when we define all required and capacity steps literally, we handle problems we can encounter at the design steps.

a- Functional Requirements

– Users can register the system.
– Users can login or log out the system.
– Users can share, upload, view, comment videos in the system.
– Users can find videos by searching the videos.

– Users can like or dislike the videos, under this condition, the system should be kept a number of likes, dislikes, comments, views to present these number to users.

b Non Functional Requirements

– System should be highly reliable. As we mentioned above, users can share and upload videos to the system and system should guarantee to keep these videos against loss. Replication and sharding help system to keep data against loss. We will explain replication in detail under step8.
– System should be highly available. If we talk about Youtube, Netflix or other large scalable systems, this means that these systems are going to be exposed to large traffic. There can be a huge number of request at the same time at the system and system should tend to respond to all requests in a real-time experience. Replication, sharding and load balancer helps system to be highly available. We will talk about all three features under step7, step8 and step9.
– System should respond with minimum latency. You imagine that you would be a user and you want to do anything on the Youtube, do you really want to wait to much time to get the response? I think you don't want to wait too much time as well as you want to experience the system with real-time. To illustrate this, when you search a video in the system, this system should suggest related videos as soon as possible.

c- Extended Requirements

– Extended requirements mainly for improving system performance. To illustrate this, the system can be monitored to define usability and durability.

3- CAPACITY ESTIMATION

To estimate the capacity of the scalable system, we should focus on the total user, active user, number of coming request and data. All these features can be considered for next 10 years because when we do that, we can easily handle the scaling problems that we can encounter after years. We can estimate the storage capacity, total usage, bandwidth estimate, cache capacity etc.. before starting the design process. Notice that, at systems like Youtube, the number of the read and write operations can be different. For example, in our system, we can estimate that the ratio of the write and read operation is 1:100. This is briefly called an upload and download ratio.

a- User estimation

We can assume that total user count of the Youtube is 1 billion and Youtube has 500 million active users in each day.

b- Storage estimation

Let's talk about the average size of videos. We are just focusing on the uploading operations because only uploading operations take up space at the storage. We assume one minute of video needs 50 MB to storage. This also should include the multiple formats of the uploaded video. Let's try to find the total minute of uploaded video in each day. As you remember, we have mentioned the ratio of write and read operation which is 1:100. If we assume that each active user downloads 5 videos then the number of uploaded video is 500 Million * 5 / 100 = 25 Million. We can assume that each video is approximately 2 minute. A total video minute is 50 Million in each day. As a result, daily needed storage approximately equals to 50 Million * 50 MB = 125 * 10^6 * 50 * 10^6 = 125 * 50 * 10^12 = 6250 * 10^12 =

6250 TB. This amount of daily storage is huge. Note that, this calculation doesn't contain the replication, duplication and etc… and because of that the storage estimation can change and may be overestimated.

c- Cache Estimation

Cache is the fast way to catch the data. I have explained the cache system in this link Caching. You can get the detail information of the cache system.
In general, a cache is estimated according to daily storage. To estimate the cache capacity, we use the popular ratio based on the 80-20 rule. This means 20% of popular data are cached. Briefly, we can estimate that Youtube cache capacity approximately equals to 6250 TB / 5 = 1250 TB.

d- Bandwith Estimation

An internet connection with a larger bandwidth can move a set amount of data much faster than an internet connection with a lower bandwidth. Bandwith is calculated according to both of download and upload operations for a second.

4- DATABASE DESIGN

There are two choices to define the database schema. These are SQL and NoSQL. We can use traditional database management system like MsSQL or MySQL to keep data. As you know, we should keep information about videos and users into RDBMS. Other information about videos, called metadata, should be kept too. Now we have main three tables to keep data. (Notice that we just only think the basic properties of Youtube. We can forget the recommendation system).

User

- UserID (primary key)
- Name (nvarchar)
- Age (Integer)
- Email (nvarchar)
- Adress (nvarchar)
- Register Date (DateTime)
- Last Login (DateTime)

Video

- VideoID (primary key – generated by KGS – Key generation service)
- VideoTitle (nvarchar)
- Size (float)
- UserID (foreign key with User Table)
- Description (nvarchar)
- CategoryID (int) : Note that we can create Category Table to define categories
- Number of Likes (int)
- Number of Dislikes (int)
- Number of Displayed (int) – We can use big int to keep displayed number
- Uploaded Date (DateT?me)

VideoComment

- CommentID (primary key)
- UserID (foreign key with User Table)

– VideoID (foreing key with Video Table)
– Comment (nvarchar)
– CommentDate (DateTime)

5- API DESIGN

In today's world, a lot of systems support mobile platform so APIs are the best choices to be able to provide the distinction between developers and support mobile support as well. We can use REST or SOAP. A lot of huge companies prefer to REST or SOAP according to their systems. There are three main API's we will mention below:

1- UploadVideo(apiKey, title, description, categoryID, language)

Upload video is the first API that we should mention. There are basically five main properties of this API. You can add more properties to UploadVideo API. Note that, apiKey is the developer key of registered account of service. Thanks to apiKey we can eliminate hacker attacks. UploadVideo returns the HTTP response that demonstrates video is uploaded successfully or not.

2- DeleteVideo (apiKey, videoID)

DeleteVideo returns HTTP response like that demonstrates video is deleted successfully or not.

3- SearchVideo (apiKey, query, videoCountToReturn, pageNumber)

SearchVideo takes 4 main properties and you can add more properties to SearchVideo API. SearchVideo returns the related videos according to query. We should not forget that SearchVideo is based on a title of the video. SearchVideo API returns the JSON that includes the information of related videos. Notice that this API can return the likes count, dislikes count, comments count and etc…

If we talk about the real Youtube service, SearchVideo API considers the recommendation.

6- HIGH-LEVEL DESIGN PROCESS

There are basic features found in web-based systems. The main ones are client, web server, application server, database and cache systems. Depending on the intensity of system traffic, the number of servers or services increases and the load balancer distributes incoming requests between these servers or services. Additionally, queues can be used depending on the density of incoming requesters. Queue operation helps users to keep from waiting more time to get respond. In our Youtube service;

– Client
– Web Server
– Application Server
– Database
– Video Storage
– Encode
– Queue

We can distribute services to three parts to decrease response time because video uploading takes more time from video downloading. Video can be downloaded from the cache and getting data from the cache is a fast way. The client basically users who use the system. Web Server is the first entity that meets request. Incoming request can take place in upload

service, search service or download service. If we give an opportunity to users that download video asynchronously, system traffic will be relaxed. An encoder is to encode uploaded video into multiple formats. There are three types of databases which are Video content database, user database and video metadata storage. Queue process takes place between an application server and encode.

Our Youtube service would be read-heavy and we should be careful when building a system. Our main goal should be returning videos quickly. We can keep copies of videos on different servers to handle the traffic problems. Additionally, this ensures the safety of this system. We can distribute our traffic to different servers with using a load balancer. The system can keep two more replicas of metadata database, user database and video content database. We can use CDN to cache the popular data.

Flow diagram of the system;

1- A client sends a request.
2- Request meets from the web server.
3- Web server controls the cache. There can be two more cache databases on the system.
4- If the request takes place into a cache, a response is redirected to the client.
5- Otherwise, web server redirects the request an to the application server.
6- There can be load balancer between web servers and application servers.

** If this request is search or view service, it tries to find the request by looking at the metadata database and the video content database. A load balancer can be placed each layer of the system such as between application server and video content database, between the application server and metadata database etc…
When a server responds the request to the client, related data is cached according to the cache process.

7- OPTIMIZATION OF THE SYSTEM

As we know, Youtube is huge video sharing system. Users can upload videos and the number of uploading videos grows exponentially day by day. According to uploading videos, there may one more same video in the system. To eliminate the duplication of videos we can implement an intelligent algorithm. For example, when a video comes to a system, the algorithm automatically checks whether this video is already kept in the system or not. If the system has already this video, then we don't need to keep duplicate data. It saves us from unnecessary use of space. Additionally, if incoming video includes a video kept in the system, then we can divide videos into small chunks and we just give the only reference to same video chunks to handle the duplication problem.

8- SHARDING PROCESS

As we mentioned above, sharding ensures that the system is more reliable and available. There can be two types of sharding process which are userID based and videoID based. Notice that userID based sharding process may cause problems. To illustrate this, when we shard data according to userID, the system tries to fit all user data in one machine and if the user uploads huge video to Youtube, we can face a fit problem. Another problem for sharding based on userID is a uniform distribution. Not every person uploads the same data size. This causes that system can be distributed non-uniformly. When we shard data according to videoID, we can eliminate these two problems.

9- REPLICATION PROCESS

As we mentioned above, the replication process is a valuable process to provide high availability, high reliability and real-time experience. As you know, it is recommended to have 3 or more replicas for each server and service. If our system has 2 more replicas for an application server, web server, CDN, encoding service, user database, metadata database, cache DB, system automatically becomes more reliable and available. We can keep same data onto three different resources and thanks to this process if one server dies, the system automatically continues to work replicas. One more advantage of replication is the system may continue to run at an update to the system.

10- LOAD BALANCING

Load balancing refers to distributing requests across a group of services and servers. When we have talked about the replication and sharding, an incoming request must be directed and this is done by a load balancer. We can use Round-Robin method to redirect incoming requests but there may be a problem in this case. Round-Robin stops sending requests to dead servers but Round-Robin cannot stop sending requests to a server exposed to heavy traffic. We can prepare intelligent Round-Robin algorithm to take away this problem. Additionally, we can use consistent hashing to redirect incoming requests. Consistent hashing ensures that system becomes more uniformly distributed.

11- CACHING MECHANISM

There can be two types of caching mechanism in our system.

- Memcache (for user and video database)
- CDN (for static media)

** Memcache or Redis is the good way to create cache mechanism. As you know, caching ensures that system responds popular data quickly. Additionally, we can set up a caching mechanism on same servers or different servers. We should be careful about caching mechanism because when we keep cache data on the same server, this means we have limited capacity. On the other hand, when we set cache mechanism on different servers we have more capacity. However, using a different server for caching mechanism cause latency problem. LRU principle is a good way to set up a cache mechanism.

12- DESIGN CONSIDERATION

Video uploading is a big process. When it fails, the system should ensure that it should continue to upload video from the failing point.
Video encoding process should include the queue operations. When an uploaded video comes, this new task is added to a queue and all tasks in the queue are taken one by one from a queue.

13-) BASIC CODING OF SYSTEM

```
 // Java program for design
public class Server { ArrayList<Machine> machines = new ArrayList<Machine>(); }
}

public class Storage {
  ArrayList<VideoMachine> machines = new ArrayList<VideoMachine>();
}
```

```
public class Machine {
  public ArrayList<User> users = new ArrayList<User>();
  public int machineID;
}

public class VideoMachine {
  public ArrayList<Video> videos = new ArrayList<Video>();
  public int machineID;
}

public class User {
  private ArrayList<Integer> videos;
  private int userID;
  private int machineID;
  private String information;
  private Server server = new Server();
  private Storage storage = new Storage();

  public User(int userID, int machineID) {
    this.userID = userID;
    this.machineID = machineID;
    videos = new ArrayList<Integer>();
  }

  public String getInformation() { return information; }

  public void setInformation(String information) {
    this.information = information;
  }

  public getID() { return userID; }

  public int getMachineID() { return machineID; }

  public void addV ? deo(int id) { videos.add(i); }

  public int[] getVideos() {
    int[] temp = new int[videos.size()];
    for (int i = 0; i < temp.length; i++) {
      temp[i] = videos.get(i);
    }
    return temp;
  }

  public User lookUpVideo(int machineID, int ID) {
    for (Machine m : storage.machine) {
      if (m.machineID = machineID) {
```

```
        for (Video video : m.video) {
          if (video.videoID = ID) {
            return video;
          }
        }
      }
    }
    return null;
  }
}

public class Video {
  private int machineID;
  private int videoID;
  private String videoPath;

  public Video(int machineID, int videoID, String photoPath) {
    this.machineID = machineID;
    this.videoID = videoID;
    this.videoPath = videoPath;
  }

  public int getMachineID() { return machineID; }

  public void setMachineID(int machineID) { this.machineID = machineID; }

  public int getVideoID() { return videoID; }

  public int getVideoPath() { return videoPath; }

  public void setVideoPath(String videoPath) { this.videoPath = videoPath; }
}
```

Reference: gorkemgenc.com

Source

https://www.geeksforgeeks.org/design-video-sharing-system-like-youtube/

Chapter 21

Design a movie ticket booking system like Bookmyshow

Design a movie ticket booking system like Bookmyshow - GeeksforGeeks

We need to design an online Movie ticket booking system where a user can search a movie in a given city and book it.This article will explain you the architecture of the booking system.

How to implement seat booking process?

Solution :

The Main Classes to be used for the user personas :

- User
- Movie
- Theater
- Booking
- Address
- Facilities

```
 // Java skeleton code to design an online movie
// booking system.
Enums :

    public enum SeatStatus {
    SEAT_BOOKED,
    SEAT_NOT_BOOKED;
}

public enum MovieStatus {
    Movie_Available,
    Movie_NotAvailable;
}
```

```
public enum MovieType {
    ENGLISH,
    HINDI;
}

public enum SeatType {
    NORMAL,
    EXECUTIVE,
    PREMIUM,
    VIP;
}

public enum PaymentStatus {
    PAID,
    UNPAID;
}

class User {

    int userId;
    String name;
    Date dateOfBirth;
    String mobNo;
    String emailId;
    String sex;
}

class Movie {

    int movieId;
    int theaterId;
    MovieType movieType;
    MovieStatus movieStatus;
}

class Theater {

    int theaterId;
    String theaterName;
    Adress adress;

    List<Movie> movies;
    float rating;
}

class Booking {
    int bookingId;
```

```
    int userId;
    int movieId;
    List<Movie> bookedSeats;
    int amount;
    PaymentStatus status_of_payment;
    Date booked_date;
    Time movie_timing;
}

class Address {

    String city;
    String pinCode;
    String state;
    String streetNo;
    String landmark;
}
```

This is an OOP design question, so full code is not required. The above code has classes and attributes only. In the above code, as you can see enums are self-explanatory.
We have users class in which users details are kept.
Theater class in which name of the theater, it's address and list of movies currently running are kept.
Booking class lets you book the seat in a particular theater. It keeps a reference in Movie, Payment class.

How to handle the cases where two persons are trying to access the same seat almost same time?

Lets take SeatBook and Transactions class which will be called from the main class. Here from the above code, We expand a bit the payment process which is not shown in the above code. In SeatBook class we will have reference to Transaction class also.

Now to ensure when two persons are trying to access the same seat almost at the same time then we would use Synchronized method of Thread class and will call a thread belong to each log in user.

```
 Class SeatBook
{
    Transaction transaction_obj;
    bool seats[total_seats];
    String place;
    String ticketType;

    bool check_availability();

    int  position_of_seat()
    {
        return seat_pos_in_theator;
```

```
    }

    void multiple tickets();

    void final_booking()
    {
        place = positon_of_seat();
        if (single_ticket)
            continue;
        else
            mutliple_ticket_booking();

        Transaction_obj.pay(ticketType, seats_booked, place);
    }
}
```

Source

https://www.geeksforgeeks.org/design-movie-ticket-booking-system-like-bookmyshow/

Chapter 22

Design an online book reader system

Design an online book reader system - GeeksforGeeks

Design an online book reader system (Object Oriented Design).

Asked In: Amazon, Microsoft, and many more interviews

Solution: Let's assume we want to design a basic online reading system which provides the following functionality:

- Searching the database of books and reading a book.
- User membership creation and extension.
- Only one active user at a time and only one active book by this user

The class OnlineReaderSystem represents the body of our program. We could implement the class such that it stores information about all the books, deals with user management, and refreshes the display, but that would make this class rather hefty.Instead, we've chosen to tear off these components into Library, UserManager, and Display classes.

The classes:

1. User
2. Book
3. Library
4. UserManager
5. Display
6. OnlineReaderSystem

Full code is given below :

```
 import java.util.HashMap;

/*
* This class represents the system
```

```
*/

class OnlineReaderSystem {
    private Library library;
    private UserManager userManager;
    private Display display;
    private Book activeBook;
    private User activeUser;

    public OnlineReaderSystem()
    {
        userManager = new UserManager();
        library = new Library();
        display = new Display();
    }

    public Library getLibrary()
    {
        return library;
    }

    public UserManager getUserManager()
    {
        return userManager;
    }

    public Display getDisplay()
    {
        return display;
    }

    public Book getActiveBook()
    {
        return activeBook;
    }

    public void setActiveBook(Book book)
    {
        activeBook = book;
        display.displayBook(book);
    }

    public User getActiveUser()
    {
        return activeUser;
    }

    public void setActiveUser(User user)
```

```
    {
        activeUser = user;
        display.displayUser(user);
    }
}

/*
* We then implement separate classes to handle the user
* manager, the library, and the display components
*/

/*
* This class represents the Library which is responsible
* for storing and searching the books.
*/
class Library {
    private HashMap<Integer, Book> books;

    public Library()
    {
        books = new HashMap<Integer, Book>();
    }

    public Boolean addBook(int id, String details, String title)
    {
        if (books.containsKey(id)) {
            return false;
        }
        Book book = new Book(id, details, title);
        books.put(id, book);
        return true;
    }

    public Boolean addBook(Book book)
    {
        if (books.containsKey(book.getId())) {
            return false;
        }

        books.put(book.getId(), book);
        return true;
    }

    public boolean remove(Book b)
    {
        return remove(b.getId());
    }
```

```
    public boolean remove(int id)
    {
        if (!books.containsKey(id)) {
            return false;
        }
        books.remove(id);
        return true;
    }

    public Book find(int id)
    {
        return books.get(id);
    }
}

/*
* This class represents the UserManager which is responsible
* for managing the users, their membership etc.
*/

class UserManager {
    private HashMap<Integer, User> users;

    public UserManager()
    {
        users = new HashMap<Integer, User>();
    }
    public Boolean addUser(int id, String details, String name)
    {
        if (users.containsKey(id)) {
            return false;
        }
        User user = new User(id, details, name);
        users.put(id, user);
        return true;
    }

    public Boolean addUser(User user)
    {
        if (users.containsKey(user.getId())) {
            return false;
        }

        users.put(user.getId(), user);
        return true;
    }

    public boolean remove(User u)
```

```
    {
        return remove(u.getId());
    }

    public boolean remove(int id)
    {
        if (users.containsKey(id)) {
            return false;
        }
        users.remove(id);
        return true;
    }

    public User find(int id)
    {
        return users.get(id);
    }
}

/*
* This class represents the Display, which is responsible
* for displaying the book, it's pages and contents. It also
* shows the current user. * It provides the method
* turnPageForward, turnPageBackward, refreshPage etc.
*/

class Display {
    private Book activeBook;
    private User activeUser;
    private int pageNumber = 0;

    public void displayUser(User user)
    {
        activeUser = user;
        refreshUsername();
    }

    public void displayBook(Book book)
    {
        pageNumber = 0;
        activeBook = book;

        refreshTitle();
        refreshDetails();
        refreshPage();
    }

    public void turnPageForward()
```

```
    {
        pageNumber++;
        System.out.println("Turning forward to page no " +
                   pageNumber + " of book having title " +
                                    activeBook.getTitle());
        refreshPage();
    }

    public void turnPageBackward()
    {
        pageNumber--;
        System.out.println("Turning backward to page no " +
                    pageNumber + " of book having title " +
                                   activeBook.getTitle());
        refreshPage();
    }

    public void refreshUsername()
    {
        /* updates username display */
        System.out.println("User name " + activeUser.getName() +
                                              " is refreshed");
    }

    public void refreshTitle()
    {
        /* updates title display */
        System.out.println("Title of the book " +
                        activeBook.getTitle() + " refreshed");
    }

    public void refreshDetails()
    {
        /* updates details display */
        System.out.println("Details of the book " +
                        activeBook.getTitle() + " refreshed");
    }

    public void refreshPage()
    {
        /* updated page display */
        System.out.println("Page no " + pageNumber + " refreshed");
    }
}

/*
* The classes for User and Book simply hold data and
* provide little functionality.
```

```
* This class represents the Book which is a simple POJO
*/

class Book {
    private int bookId;
    private String details;
    private String title;

    public Book(int id, String details, String title)
    {
        bookId = id;
        this.details = details;
        this.title = title;
    }

    public int getId()
    {
        return bookId;
    }

    public void setId(int id)
    {
        bookId = id;
    }

    public String getDetails()
    {
        return details;
    }

    public void setDetails(String details)
    {
        this.details = details;
    }

    public String getTitle()
    {
        return title;
    }

    public void setTitle(String title)
    {
        this.title = title;
    }
}

/*
* This class represents the User which is a simple POJO
```

```
*/

class User {
    private int userId;
    private String name;
    private String details;

    public void renewMembership()
    {
    }

    public User(int id, String details, String name)
    {
        this.userId = id;
        this.details = details;
        this.name = name;
    }

    public int getId()
    {
        return userId;
    }

    public void setId(int id)
    {
        userId = id;
    }

    public String getDetails()
    {
        return details;
    }

    public void setDetails(String details)
    {
        this.details = details;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}
```

```
// This class is used to test the Application

public class AppTest {

    public static void main(String[] args)
    {

        OnlineReaderSystem onlineReaderSystem = new OnlineReaderSystem();

        Book dsBook = new Book(1, "It contains Data Structures", "Ds");
        Book algoBook = new Book(2, "It contains Algorithms", "Algo");

        onlineReaderSystem.getLibrary().addBook(dsBook);
        onlineReaderSystem.getLibrary().addBook(algoBook);

        User user1 = new User(1, " ", "Ram");
        User user2 = new User(2, " ", "Gopal");

        onlineReaderSystem.getUserManager().addUser(user1);
        onlineReaderSystem.getUserManager().addUser(user2);

        onlineReaderSystem.setActiveBook(algoBook);
        onlineReaderSystem.setActiveUser(user1);

        onlineReaderSystem.getDisplay().turnPageForward();
        onlineReaderSystem.getDisplay().turnPageForward();
        onlineReaderSystem.getDisplay().turnPageBackward();
    }
}
```

Reference :
Book :Cracking the Coding Interview

Source

https://www.geeksforgeeks.org/design-an-online-book-reader-system/

Chapter 23

Design an online hotel booking system like OYO Rooms

Design an online hotel booking system like OYO Rooms - GeeksforGeeks

We need to design an online hotel booking system where a user can search a hotel in a given city and book it. This is an OOP design question, so I have not written the full code in this solution. I have created the classes and attributes only.

Solution :

Main Classes :

1. User
2. Room
3. Hotel
4. Booking
5. Address
6. Facilities

```
 // Java code skeleton to design an online hotel
// booking system.
Enums:

public enum RoomStatus {
    EMPTY
        NOT_EMPTY;
}

public enum RoomType {
    SINGLE,
    DOUBLE,
    TRIPLE;
}
```

```
public enum PaymentStatus {
    PAID,
    UNPAID;
}

public enum Facility {
    LIFT;
    POWER_BACKUP;
    HOT_WATERR;
    BREAKFAST_FREE;
    SWIMMING_POOL;
}

class User {

    int userId;
    String name;
    Date dateOfBirth;
    String mobNo;
    String emailId;
    String sex;
}

// For the room in any hotel
class Room {

    int roomId; // roomNo
    int hotelId;
    RoomType roomType;
    RoomStatus roomStatus;
}

class Hotel {

    int hotelId;
    String hotelName;
    Adress adress;

    // hotel contains the list of rooms
    List<Room> rooms;
    float rating;
    Facilities facilities;
}

// a new booking is created for each booking
// done by any user
class Booking {
```

```
    int bookingId;
    int userId;
    int hotelId;

    // We are assuming that in a single
    // booking we can book only the rooms
    // of a single hotel
    List<Rooms> bookedRooms;

    int amount;
    PaymentStatus status_of_payment;
    Date bookingTime;
    Duration duration;
}

class Address {

    String city;
    String pinCode;
    String state;
    String streetNo;
    String landmark;
}

class Duration {

    Date from;
    Date to;

}

class Facilities {

    List<Facility> facilitiesList;
}
```

Let me explain about the classes and the relationships among themselves.

The enums defined here are self-explanatory. The classes User, Room, and Address also self-explanatory. The class Facilities contains a list of facilities (enum) that the hotel provides. We can add more facilities in the Facility enum if required. The duration class has two attributes “from” and “to”, which is obvious.

Now, the class “Hotel” contains:
1. List of rooms (Room class) // this is the list of rooms the hotel has
2. Address class // its address
3. Facilities class // the facilities it has

The class “Booking” contains:
1. User // information about the

2. Hotel // Information about the hotel
3. List of rooms
4. PaymentStatus etc.

Other fields in this class are also self-explanatory.

Source

https://www.geeksforgeeks.org/design-online-hotel-booking-system-like-oyo-rooms/

Chapter 24

Design data structures and algorithms for in-memory file system

Design data structures and algorithms for in-memory file system - GeeksforGeeks

Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in the code logic where possible.

Asked In: Amazon

A file system, in its most simplistic version, consists of Files and Directories. Each Directory contains a set of Files and Directories. Since Files and Directories share so many characteristics, we've implemented them such that they inherit from the same class, Entry.

Implemented Main logic in Java

```
 // Entry is superclass for both File and Directory
public abstract class Entry
{
    protected Directory parent;
    protected long created;
    protected long lastUpdated;
    protected long lastAccessed;
    protected String name;

    public Entry(String n, Directory p)
    {
        name = n;
        parent = p;
        created= System.currentTimeMillis();
        lastUpdated = System.currentTimeMillis();
        lastAccessed = System.currentTimeMillis();
```

```
    }

    public boolean delete()
    {
        if (parent == null)
            return false;
        return parent.deleteEntry(this);
    }

    public abstract int size();

    /* Getters and setters. */
    public long getcreationTime()
    {
        return created;
    }
    public long getLastUpdatedTime()
    {
        return lastUpdated;
    }
    public long getLastAccessedTime()
    {
        return lastAccessed;
    }
    public void changeName(String n)
    {
        name = n;
    }
    public String getName()
    {
        return name;
    }
}

// A class to represent a File (Inherits
// from Entry)
public class File extends Entry
{
    private String content;
    private int size;

    public File(String n, Directory p, int sz)
    {
        super(n, p);
        size = sz;
    }
    public int size()
    {
```

```
        return size;
    }
    public String getContents()
    {
        return content;
    }
    public void setContents(String c)
    {
        content = c;
    }
}

// A class to represent a Directory (Inherits
// from Entry)
public class Directory extends Entry
{
    protected Arraylist<Entry> contents;

    public Directory(String n, Directory p)
    {
        super(n, p);
        contents = new Arraylist<Entry>();
    }
    public int size()
    {
        int size = 0;
        for (Entry e : contents)
            size += e.size();

        return size;
    }
    public int numberOfFiles()
    {
        int count = 0;
        for (Entry e : contents)
        {
            if (e instanceof Directory)
            {
                count++; // Directory counts as a file
                Directory d = (Directory) e;
                count += d. numberOfFiles ();
            }
            else if (e instanceof File)
                count++;
        }
        return count;
    }
```

```
    public boolean deleteEntry(Entry entry)
    {
        return contents.remove(entry);
    }

    public void addEntry(Entry entry)
    {
        contents.add(entry);
    }

    protected ArrayList<Entry> getContents()
    {
        return contents;
    }
}
```

Alternatively, we could have implemented Directory such that it contains separate lists for files and subdirectories. This makes the nurnberOfFiles () method a bit cleaner, since it doesn't need to use the instanceof operator, but it does prohibit us from cleanly sorting files and directories by dates or names.

For data block allocation, we can use bitmask vector and linear search (see "Practical File System Design") or B+ trees (see Reference or Wikipedia).

References:
https://www.careercup.com/question?id=13618661
https://stackoverflow.com/questions/14126575/data-structures-used-to-build-file-systems

Source

https://www.geeksforgeeks.org/design-data-structures-algorithms-memory-file-system/

Chapter 25

Design the Data Structures(classes and objects)for a generic deck of cards

Design the Data Structures(classes and objects)for a generic deck of cards - GeeksforGeeks

Design the data structures for a generic deck of cards Explain how you would sub-class it to implement particular card games and how you would subclass the data structures to implement blackjack.

Asked In : Amazon Interview

Solution:

First, we need to recognize that a "generic" deck of cards can mean many things. Generic could mean a standard deck of cards that can play a poker-like game, or it could even stretch to Uno or Baseball cards.

To implement particular card games

Let's assume that the deck is a standard 52-card set like you might see used in a blackjack or poker game. If so, the design might look like this:

The structure is clear here: a deck contains four suits and a suit contains 13 card. Each card has a numerical value from 1 to 13. If you think about a card game, different games differ from ways of dealing cards and putting cards back in. So we can have a set of abstract methods inside the class 'Deck' to allow sub-class implements its own way of dealing. The class diagram I draw is here:

Below is C++ implementation of the idea.

```
 /*1. Investigation on an individual card instead of
   a collection of cards, focus on a card's state
   and interface.
2. A card game has its own specific constrain and
   requirement on cards, such that a generic card
   cannot satisfy a blackjack card
3. Player manage multiple cards */

#include <bits/stdc++.h>
using namespace std;

namespace SUIT {
enum Enum {
    SPADE,
    HEART,
    CLUB,
    DIAMOND
};
};

class Card {
private:
    SUIT::Enum s;
    int v;

public:
    virtual SUIT::Enum suit() const
    {
        return s;
    };

    virtual int val() const
    {
        return v;
    };

    Card(int val, SUIT::Enum suit)
        : s(suit), v(val){};
};

class BlackJackCard : public Card {
public:
    virtual int val()
    {
        int v = Card::val();
        if (v < 10)
```

```
            return v;
        return 10;
    }

    BlackJackCard(int val, SUIT::Enum suit)
        : Card(val, suit){};
};

class player {
private:
    int id;
    int bet;
    set<int> points;
    vector<BlackJackCard*> bjcs;
    bool addPoint(set<int>& amp; points, BlackJackCard * card)
    {
        if (points.empty()) {
            points.insert(card->val());
            if (card->val() == 1)
                points.insert(11);
        } else {

            /* Set elements are ALWAYS CONST, they can't
               be modified once inserted. */
            set<int> tmp;
            for (auto it = points.begin(); it != points.end(); ++it) {
                tmp.insert(*it + card->val());
                if (card->val() == 1)
                    tmp.insert(*it + 11);
            }
            points = tmp;
        }
    }

    void getPoints()
    {
        cout << "You All Possible Points : " << endl;
        for (auto it = points.begin(); it != points.end(); ++it) {
            cout << *it << endl;
        }
    };

    int getMinPoints()
    {
        /* set is implemented by commonly BST, so else
          are in order!!!
          learn to use lower_bound() and upper_bound()
          "they allow the direct iteration on subsets
```

```
           based on their order."
          which gives us another option to find min. preferable */

        // return *(points.lower_bound(0));
        return *(points.begin());
    };

    void printCards()
    {
        cout << "You Cards : " << endl;
        for (auto it = bjcs.begin(); it != bjcs.end(); ++it) {
            cout << (*it)->val() << endl;
        }
    }

public:
    player(int i, int j)
        : id(i), bet(j)
    {
        bjcs.push_back(new BlackJackCard(rand() % 13 + 1, SUIT::SPADE));
        bjcs.push_back(new BlackJackCard(rand() % 13 + 1, SUIT::SPADE));
        addPoint(points, bjcs[0]);
        addPoint(points, bjcs[1]);
    };

    void getAnotherCard()
    {
        for (set<int>::iterator it = points.begin(); it != points.end(); ++it) {

            /* predefined strategy for the player    */
            if (*it <= 21 && 21 - *it <= 4) {
                printCards();
                getPoints();
                cout << "Stand" << endl;
                exit(1);
            }
        }
        bjcs.push_back(new BlackJackCard(rand() % 13 + 1, SUIT::SPADE));
        addPoint(points, bjcs.back());
        if (getMinPoints() > 21) {
            printCards();
            getPoints();
            cout << "Busted" << endl;
            exit(2);
        }
    };

    virtual ~player()
```

```
    {
        for (auto it = bjcs.begin(); it != bjcs.end(); ++it) {
            delete *it;
        }
    };
};
// Driver code
int main()
{
    srand(time(NULL));
    player p(1, 1000);
    p.getAnotherCard();
    p.getAnotherCard();
    p.getAnotherCard();

    return 0;
}
```

Output:

```
You Cards :
10
10
You All Possible Points :
20
Stand
```

To implement Blackjack.

Note: Now, let's say we're building a blackjack game, so we need to know the value of the cards. Face cards are 10 and an ace is 11 (most of the time, but that's the job of the Hand class, not the following class).

At the start of a blackjack game, the players and the dealer receive two cards each. The players' cards are normally dealt face up, while the dealer has one face down (called the hole card) and one face up.

The best possible blackjack hand is an opening deal of an ace with any ten-point card. This is called a "blackjack", or a natural 21, and the player holding this automatically wins unless the dealer also has a blackjack. If a player and the dealer each have a blackjack, the result is a push for that player. If the dealer has a blackjack, all players not holding a blackjack lose.

Main logic of Blackjack in Java

```
 public class BlackJackHand extends Hand<BlackJackCard> {

    /* There are multiple possible scores for a blackjack
   hand, since aces have 3 * multiple values. Return
   the highest possible score that's under 21, or the
   4 * lowest score that's over. */

    public int score()
    {
        Arraylist<Integer> scores = possibleScores();
        int maxUnder = Integer.MIN_VALUE;
        int minOver = Integer.MAX_VALUE;
        for (int score : scores) {
            if (score > 21 & amp; &score < minOver) {
                minOver = score;
            } else if (score <= 21 & amp; &score > maxUnder) {
                maxUnder = score;
            }
        }
        return maxUnder Integer.MIN_VALUE ? minOver maxUnder;
    }

    /* return a list of all possible scores this hand could have
     (evaluating each * ace as both 1 and 11 */
    private Arraylist<Integer> possibleScores() { ... }

    public boolean busted() { return score() > 21; }
    public boolean is21() { return score() == 21; }
    public boolean isBlackJack() { ... }
}

public class BlackJackCard extends Card {
    public BlackJackCard(int c, Suit s) { super(c, s); }
    public int value()
    {
        if (isAce())
            return 1;
        else if (faceValue >= 11 & amp; &faceValue <= 13)
            return 10;
        else
            return faceValue;
    }

    public int minValue()
    {
        if (isAce())
            return 1;
        else
```

```
            return value();
    }

    public int maxValue()
    {
        if (isAce())
            return 11;
        else
            return value();
    }

    public boolean isAce()
    {
        return faceValue == 1;
    }

    public boolean isFaceCard()
    {
        return faceValue >= 11 & amp;
        &faceValue <= 13;
    }
}

/* This is just one way of handling aces. We could, alternatively, create a class
of type Ace that extends BlackJackCard. */
```

References :
https://www.careercup.com/question?id=2983
http://stackoverflow.com/questions/37363008/a-singleton-class-to-design-a-generic-deck-of-card

Source

https://www.geeksforgeeks.org/design-data-structuresclasses-objectsfor-generic-deck-cards/

Chapter 26

Facade Design Pattern | Introduction

Facade Design Pattern | Introduction - GeeksforGeeks

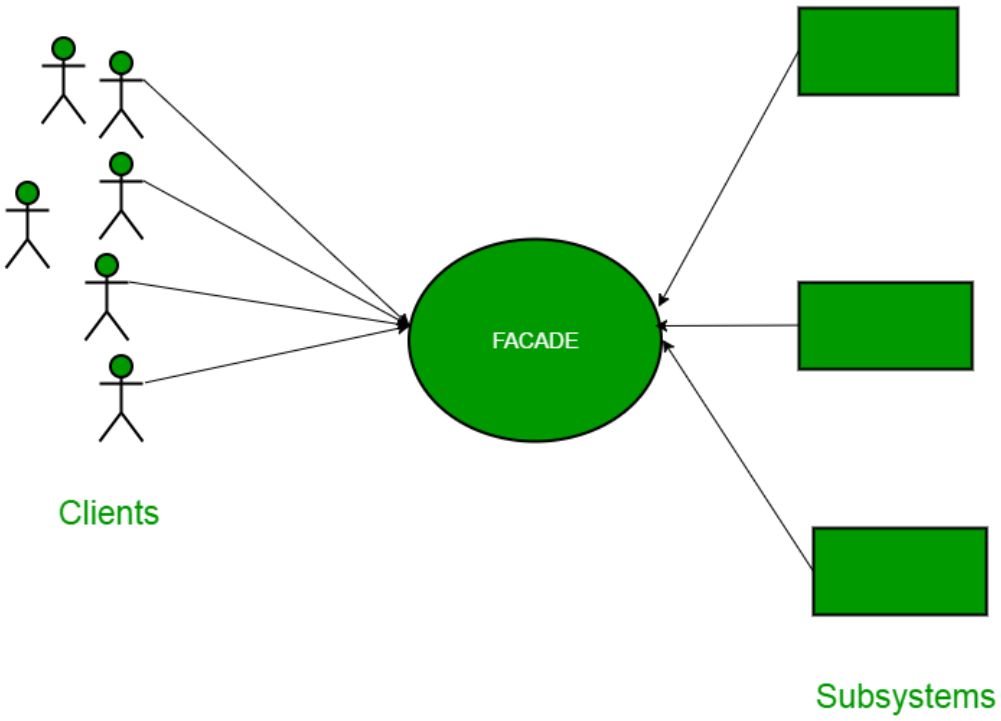
Facade is a part of Gang of Four design pattern and it is categorized under Structural design patterns. Before we dig into the details of it, let us discuss some examples which will be solved by this particular Pattern.

So, As the name suggests, it means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.

More examples

In Java, the interface JDBC can be called a facade because, we as users or clients create connection using the "java.sql.Connection" interface, the implementation of which we are not concerned about. The implementation is left to the vendor of driver.

Another good example can be the startup of a computer. When a computer starts up, it involves the work of cpu, memory, hard drive, etc. To make it easy to use for users, we can add a facade which wrap the complexity of the task, and provide one simple interface instead.
Same goes for the **Facade Design Pattern**. It hides the complexities of the system and provides an interface to the client from where the client can access the system.



Facade Design Pattern Diagram

Now Let's try and understand the facade pattern better using a simple example. Let's consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non Both restaurants.
You, as client want access to different menus of different restaurants . You do not know what are the different menus they have. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the **facade**, as he hides the complexities of the system hotel.
Let's see how it works :

Interface of Hotel

```
 package structural.facade;
public interface Hotel
{
    public Menus getMenus();
}
```

The hotel interface only returns Menus.
Similarly, the Restaurant are of three types and can implement the hotel interface. Let's have a look at the code for one of the Restaurants.

NonVegRestaurant.java

```
 package structural.facade;

public class NonVegRestaurant implements Hotel
{
    public Menus getMenus()
    {
        NonVegMenu nv = new NonVegMenu();
        return nv;
    }
}
```

VegRestaurant.java

```
 package structural.facade;

public class VegRestaurant implements Hotel
{
    public Menus getMenus()
    {
        VegMenu v = new VegMenu();
        return v;
    }
}
```

VegNonBothRestaurant.java

```
 package structural.facade;

public class VegNonBothRestaurant implements Hotel
{
    public Menus getMenus()
    {
        Both b = new Both();
        return b;
    }
}
```

Now let's consider the facade,

HotelKeeper.java

```
 package structural.facade;

public class HotelKeeper
{
    public VegMenu getVegMenu()
    {
```

```
        VegRestaurant v = new VegRestaurant();
        VegMenu vegMenu = (VegMenu)v.getMenus();
        return vegMenu;
    }

    public NonVegMenu getNonVegMenu()
    {
        NonVegRestaurant v = new NonVegRestaurant();
        NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();
        return NonvegMenu;
    }

    public Both getVegNonMenu()
    {
        VegNonBothRestaurant v = new VegNonBothRestaurant();
        Both bothMenu = (Both)v.getMenus();
        return bothMenu;
    }
}
```

From this, It is clear that the complex implementation will be done by HotelKeeper himself. The client will just access the HotelKeeper and ask for either Veg, NonVeg or VegNon Both Restaurant menu.

How will the client program access this façade?

```
 package structural.facade;

public class Client
{
    public static void main (String[] args)
    {
        HotelKeeper keeper = new HotelKeeper();

        VegMenu v = keeper.getVegMenu();
        NonVegMenu nv = keeper.getNonVegMenu();
        Both = keeper.getVegNonMenu();

    }
}
```

In this way the implementation is sent to the façade. The client is given just one interface and can access only that. This hides all the complexities.

When Should this pattern be used?

The facade pattern is appropriate when you have a **complex system** that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces,

not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

Source

https://www.geeksforgeeks.org/facade-design-pattern-introduction/

Chapter 27

Flyweight Design Pattern

Flyweight Design Pattern - GeeksforGeeks

Flyweight pattern is one of the structural design patterns as this pattern provides ways to decrease object count thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects (say 10^5). One important feature of flyweight objects is that they are **immutable**. This means that they cannot be modified once they have been constructed.

Why do we care for number of objects in our program?

- Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
- Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

In Flyweight pattern we use a HashMap that stores reference to the object which have already been created, every object is associated with a key. Now when a client wants to create an object, he simply has to pass a key associated with it and if the object has already been created we simply get the reference to that object else it creates a new object and then returns it reference to the client.

Intrinsic and Extrinsic States

To understand Intrinsic and Extrinsic state, let us consider an example.

Suppose in a text editor when we enter a character, an object of Character class is created, the attributes of the Character class are {name, font, size}. We do not need to create an object every time client enters a character since letter 'B' is no different from another 'B' . If client again types a 'B' we simply return the object which we have already created before. Now all these are intrinsic states (name, font, size), since they can be shared among the different objects as they are similar to each other.

Now we add to more attributes to the Character class, they are row and column. They specify the position of a character in the document. Now these attributes will not be similar

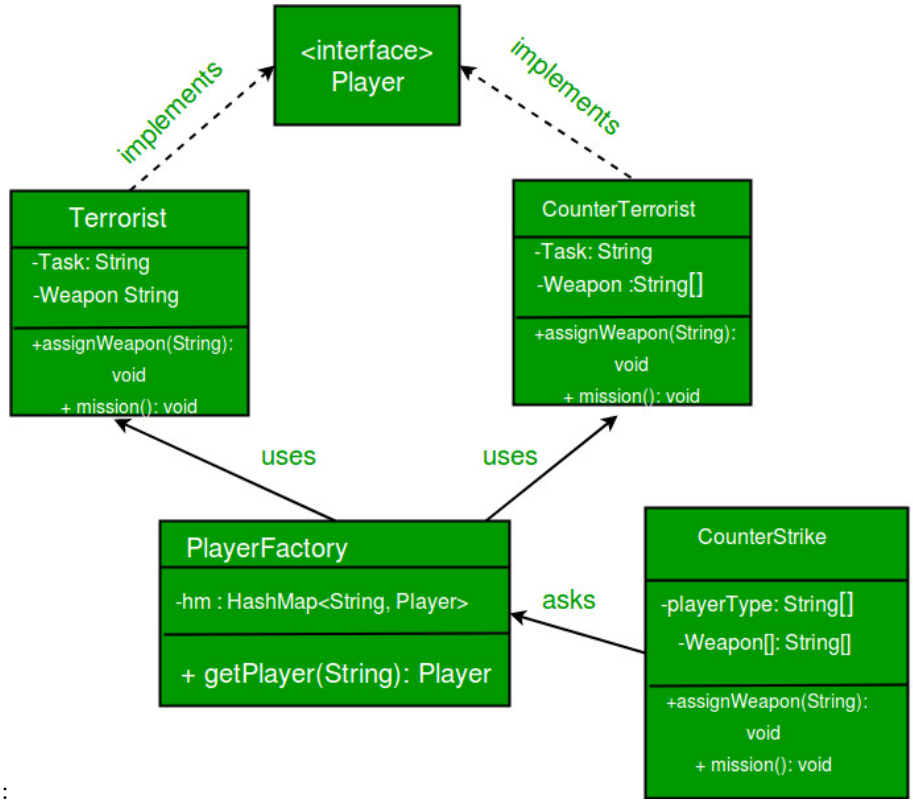
even for same characters, since no two characters will have the same position in a document, these states are termed as extrinsic states, and they can't be shared among objects.

Implementation : We implement the creation of Terrorists and Counter Terrorists In the game of Counter Strike. So we have 2 classes one for **T**errorist(**T**) and other for **C**ounter **T**errorist(**CT**). Whenever a player asks for a weapon we assign him the asked weapon. In the mission, terrorist's task is to plant a bomb while the counter terrorists have to diffuse the bomb.

Why to use Flyweight Design Pattern in this example? Here we use the Fly Weight design pattern, since here we need to reduce the object count for players. Now we have n number of players playing CS 1.6, if we do not follow the Fly Weight Design Pattern then we will have to create n number of objects, one for each player. But now we will only have to create 2 objects one for terrorists and other for counter terrorists, we will reuse then again and again whenever required.

Intrinsic State : Here 'task' is an intrinsic state for both types of players, since this is always same for T's/CT's. We can have some other states like their color or any other properties which are similar for all the Terrorists/Counter Terrorists in their respective Terrorists/Counter Terrorists class.

Extrinsic State : Weapon is an extrinsic state since each player can carry any weapon of his/her choice. Weapon need to be passed as a parameter by the client itself.



Class Diagram :

```
// A Java program to demonstrate working of
```

```
// FlyWeight Pattern with example of Counter
// Strike Game
import java.util.Random;
import java.util.HashMap;

// A common interface for all players
interface Player
{
    public void assignWeapon(String weapon);
    public void mission();
}

// Terrorist must have weapon and mission
class Terrorist implements Player
{
    // Intrinsic Attribute
    private final String TASK;

    // Extrinsic Attribute
    private String weapon;

    public Terrorist()
    {
        TASK = "PLANT A BOMB";
    }
    public void assignWeapon(String weapon)
    {
        // Assign a weapon
        this.weapon = weapon;
    }
    public void mission()
    {
        //Work on the Mission
        System.out.println("Terrorist with weapon "
                           + weapon + "|" + " Task is " + TASK);
    }
}

// CounterTerrorist must have weapon and mission
class CounterTerrorist implements Player
{
    // Intrinsic Attribute
    private final String TASK;

    // Extrinsic Attribute
    private String weapon;

    public CounterTerrorist()
```

```
    {
        TASK = "DIFFUSE BOMB";
    }
    public void assignWeapon(String weapon)
    {
        this.weapon = weapon;
    }
    public void mission()
    {
        System.out.println("Counter Terrorist with weapon "
                           + weapon + "|" + " Task is " + TASK);
    }
}

// Claass used to get a playeer using HashMap (Returns
// an existing player if a player of given type exists.
// Else creates a new player and returns it.
class PlayerFactory
{
    /* HashMap stores the reference to the object
       of Terrorist(TS) or CounterTerrorist(CT).  */
    private static HashMap <String, Player> hm =
                          new HashMap<String, Player>();

    // Method to get a player
    public static Player getPlayer(String type)
    {
        Player p = null;

        /* If an object for TS or CT has already been
           created simply return its reference */
        if (hm.containsKey(type))
                p = hm.get(type);
        else
        {
            /* create an object of TS/CT  */
            switch(type)
            {
            case "Terrorist":
                System.out.println("Terrorist Created");
                p = new Terrorist();
                break;
            case "CounterTerrorist":
                System.out.println("Counter Terrorist Created");
                p = new CounterTerrorist();
                break;
            default :
                System.out.println("Unreachable code!");
```

```
            }

            // Once created insert it into the HashMap
            hm.put(type, p);
        }
        return p;
    }
}

// Driver class
public class CounterStrike
{
    // All player types and weopons (used by getRandPlayerType()
    // and getRandWeapon()
    private static String[] playerType =
                    {"Terrorist", "CounterTerrorist"};
    private static String[] weapons =
      {"AK-47", "Maverick", "Gut Knife", "Desert Eagle"};


    // Driver code
    public static void main(String args[])
    {
        /* Assume that we have a total of 10 players
           in the game. */
        for (int i = 0; i < 10; i++)
        {
            /* getPlayer() is called simply using the class
               name since the method is a static one */
            Player p = PlayerFactory.getPlayer(getRandPlayerType());

            /* Assign a weapon chosen randomly uniformly
               from the weapon array  */
            p.assignWeapon(getRandWeapon());

            // Send this player on a mission
            p.mission();
        }
    }

    // Utility methods to get a random player type and
    // weapon
    public static String getRandPlayerType()
    {
        Random r = new Random();

        // Will return an integer between [0,2)
        int randInt = r.nextInt(playerType.length);
```

```
        // return the player stored at index 'randInt'
        return playerType[randInt];
    }
    public static String getRandWeapon()
    {
        Random r = new Random();

        // Will return an integer between [0,5)
        int randInt = r.nextInt(weapons.length);

        // Return the weapon stored at index 'randInt'
        return weapons[randInt];
    }
}
```

Output:

```
Counter Terrorist Created
Counter Terrorist with weapon Gut Knife| Task is DIFFUSE BOMB
Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB
Terrorist Created
Terrorist with weapon AK-47| Task is PLANT A BOMB
Terrorist with weapon Gut Knife| Task is PLANT A BOMB
Terrorist with weapon Gut Knife| Task is PLANT A BOMB
Terrorist with weapon Desert Eagle| Task is PLANT A BOMB
Terrorist with weapon AK-47| Task is PLANT A BOMB
Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB
Counter Terrorist with weapon Gut Knife| Task is DIFFUSE BOMB
Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB
```

References:

- Elements of Reusable Object-Oriented Software(By Gang Of Four)
- https://en.wikipedia.org/wiki/Flyweight_pattern

Source

https://www.geeksforgeeks.org/flyweight-design-pattern/

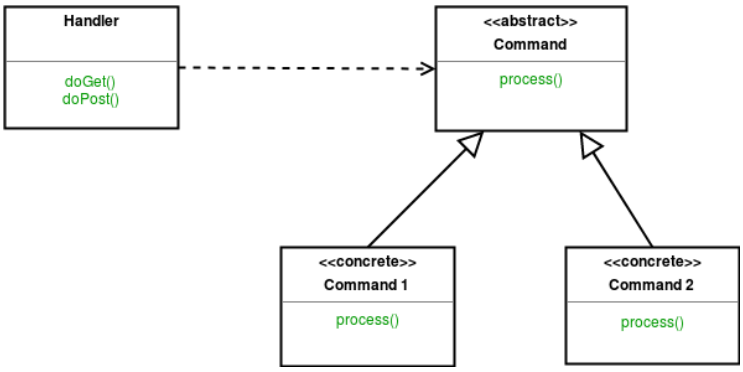
Chapter 28

Front Controller Design Pattern

Front Controller Design Pattern - GeeksforGeeks

The front controller design pattern means that all requests that come for a resource in an application will be handled by a single handler and then dispatched to the appropriate handler for that type of request. The front controller may use other helpers to achieve the dispatching mechanism.

UML Diagram Front Controller Design Pattern



Design components

- **Controller :** The controller is the initial contact point for handling all requests in the system. The controller may delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval.
- **View:** A view represents and displays information to the client. The view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data model for use in the display.
- **Dispatcher:** A dispatcher is responsible for view management and navigation, managing the choice of the next view to present to the user, and providing the mechanism for vectoring control to this resource.

- **Helper :** A helper is responsible for helping a view or controller complete its processing. Thus, helpers have numerous responsibilities, including gathering data required by the view and storing this intermediate model, in which case the helper is sometimes referred to as a value bean.

Let's see an example of Front Controller Design Pattern.

```
 class TeacherView
{
    public void display()
    {
        System.out.println("Teacher View");
    }
}

class StudentView
{
    public void display()
    {
        System.out.println("Student View");
    }
}

class Dispatching
{
    private StudentView studentView;
    private TeacherView teacherView;

    public Dispatching()
    {
        studentView = new StudentView();
        teacherView = new TeacherView();
    }

    public void dispatch(String request)
    {
        if(request.equalsIgnoreCase("Student"))
        {
            studentView.display();
        }
        else
        {
            teacherView.display();
        }
    }
}

class FrontController
```

```
{
    private Dispatching Dispatching;

    public FrontController()
    {
        Dispatching = new Dispatching();
    }

    private boolean isAuthenticUser()
    {
        System.out.println("Authentication successfull.");
        return true;
    }

    private void trackRequest(String request)
    {
        System.out.println("Requested View: " + request);
    }

    public void dispatchRequest(String request)
    {
        trackRequest(request);

        if(isAuthenticUser())
        {
            Dispatching.dispatch(request);
        }
    }
}

class FrontControllerPattern
{
    public static void main(String[] args)
    {
        FrontController frontController = new FrontController();
        frontController.dispatchRequest("Teacher");
        frontController.dispatchRequest("Student");
    }
}
```

Output:

```
Requested View: Teacher
Authentication successfull.
Teacher View
Requested View: Student
Authentication successfull.
```

```
Student View
```

Advantages :

- **Centralized control :** Front controller handles all the requests to the Web application. This implementation of centralized control that avoids using multiple controllers is desirable for enforcing application-wide policies such as users tracking and security.
- **Thread-safety :** A new command object arises when receiving a new request and the command objects are not meant to be thread safe. Thus, it will be safe in the command classes. Though safety is not guaranteed when threading issues are gathered, codes that act with command is still thread safe.

Disadvantages :

- It is not possible to scale an application using a front controller.
- Performance is better if you deal with a single request uniquely.

Source

https://www.geeksforgeeks.org/front-controller-design-pattern/

Chapter 29

How to design a parking lot using object-oriented principles?

How to design a parking lot using object-oriented principles? - GeeksforGeeks

Design a parking lot using object-oriented principles.

Asked In : Amazon, Apple, Google and many more interviews

Solution: For our purposes right now, we'll make the following assumptions. We made these specific assumptions to add a bit of complexity to the problem without adding too much. If you made different assumptions, that's totally fine.
1) The parking lot has multiple levels. Each level has multiple rows of spots.
2) The parking lot can park motorcycles, cars, and buses.
3) The parking lot has motorcycle spots, compact spots, and large spots.
4) A motorcycle can park in any spot.
5) A car can park in either a single compact spot or a single large spot.
6) A bus can park in five large spots that are consecutive and within the same row. It cannot park in small spots.

In the below implementation, we have created an abstract class Vehicle, from which Car, Bus, and Motorcycle inherit. To handle the different parking spot sizes, we have just one class ParkingSpot which has a member variable indicating the size.

Main Logic in Java given below

```
 // Vehicle and its inherited classes.
public enum VehicleSize { Motorcycle, Compact,Large }

public abstract class Vehicle
{
      protected ArrayList<ParkingSpot> parkingSpots =
                              new ArrayList<ParkingSpot>();
      protected String licensePlate;
      protected int spotsNeeded;
```

```
      protected VehicleSize size;

      public int getSpotsNeeded()
      {
          return spotsNeeded;
      }
      public VehicleSize getSize()
      {
          return size;
      }

      /* Park vehicle in this spot (among others,
         potentially) */
      public void parkinSpot(ParkingSpot s)
      {
          parkingSpots.add(s);
      }


      /* Remove vehicle from spot, and notify spot
         that it's gone */
      public void clearSpots() { ... }

      /* Checks if the spot is big enough for the
         vehicle (and is available).
         This * compares the SIZE only.It does not
        check if it has enough spots. */
      public abstract boolean canFitinSpot(ParkingSpot spot);
}

public class Bus extends Vehicle
{
    public Bus()
    {
        spotsNeeded = 5;
        size = VehicleSize.Large;
    }

    /* Checks if the spot is a Large. Doesn't check
     num of spots */
    public boolean canFitinSpot(ParkingSpot spot)
    {... }
}

public class Car extends Vehicle
{
    public Car()
    {
```

```
        spotsNeeded = 1;
        size = VehicleSize.Compact;
    }

    /* Checks if the spot is a Compact or a Large. */
    public boolean canFitinSpot(ParkingSpot spot)
    { ... }
}

public class Motorcycle extends Vehicle
{
    public Motorcycle()
    {
        spotsNeeded = 1;
        size = VehicleSize.Motorcycle;
    }
    public boolean canFitinSpot(ParkingSpot spot)
    { ... }
}
```

The **ParkingSpot** is implemented by having just a variable which represents the size of the spot. We could have implemented this by having classes for LargeSpot, CompactSpot, and MotorcycleSpot which inherit from ParkingSpot, but this is probably overkilled. The spots probably do not have different behaviors, other than their sizes.

```
 public class ParkingSpot
{
    private Vehicle vehicle;
    private VehicleSize spotSize;
    private int row;
    private int spotNumber;
    private Level level;

    public ParkingSpot(Level lvl, int r, int n,
                       VehicleSize s)
    { ... }

    public boolean isAvailable()
    {
        return vehicle == null;
    }

    /* Check if the spot is big enough and is available */
    public boolean canFitVehicle(Vehicle vehicle) { ... }
```

```
    /* Park vehicle in this spot. */
    public boolean park(Vehicle v) {..}

    public int getRow()
    {
        return row;
    }
    public int getSpotNumber()
    {
        return spotNumber;
    }

    /* Remove vehicle from spot, and notify
      level that a new spot is available */
    public void removeVehicle() { ... }
}
```

Source :
www.andiamogo.com/S-OOD.pdf

More References :
https://www.quora.com/How-do-I-answer-design-related-questions-like-design-a-parking-lot-in-an-Amazon-intervi
http://stackoverflow.com/questions/764933/amazon-interview-question-design-an-oo-parking-lot

Source

https://www.geeksforgeeks.org/design-parking-lot-using-object-oriented-principles/

Chapter 30

How to prevent Singleton Pattern from Reflection, Serialization and Cloning?

How to prevent Singleton Pattern from Reflection, Serialization and Cloning? - GeeksforGeeks

Prerequisite: Singleton Pattern

In this article, we will see that what are various concepts which can break singleton property of a class and how to avoid them. There are mainly 3 concepts which can break singleton property of a class. Let's discuss them one by one.

1. **Reflection:** Reflection can be caused to destroy singleton property of singleton class, as shown in following example:

```
 // Java code to explain effect of Reflection
// on Singleton property

import java.lang.reflect.Constructor;

// Singleton class
class Singleton
{
    // public instance initialized when loading the class
    public static Singleton instance = new Singleton();

    private Singleton()
    {
        // private constructor
    }
```

```
}

public class GFG
{

    public static void main(String[] args)
    {
        Singleton instance1 = Singleton.instance;
        Singleton instance2 = null;
        try
        {
            Constructor[] constructors =
                    Singleton.class.getDeclaredConstructors();
            for (Constructor constructor : constructors)
            {
                // Below code will destroy the singleton pattern
                constructor.setAccessible(true);
                instance2 = (Singleton) constructor.newInstance();
                break;
            }
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }

    System.out.println("instance1.hashCode():- "
                                        + instance1.hashCode());
    System.out.println("instance2.hashCode():- "
                                        + instance2.hashCode());
    }
}


Output:-
instance1.hashCode():- 366712642
instance2.hashCode():- 1829164700
```

After running this class, you will see that hashCodes are different that means, 2 objects of same class are created and singleton pattern has been destroyed.

Overcome reflection issue: To overcome issue raised by reflection, enums are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

```
 //Java program for Enum type singleton
public enum GFG
```

```
{
  INSTANCE;
}
```

As enums don't have any constructor so it is not possible for Reflection to utilize it. Enums have their by-default constructor, we can't invoke them by ourself. **JVM handles the creation and invocation of enum constructors internally.** As enums don't give their constructor definition to the program, it is not possible for us to access them by Reflection also. Hence, reflection can't break singleton property in case of enums.

2. **Serialization:-** Serialization can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of byte stream and save in a file or send over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

```
 // Java code to explain effect of
// Serilization on singleton classes
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Singleton implements Serializable
{
    // public instance initialized when loading the class
    public static Singleton instance = new Singleton();

    private Singleton()
    {
        // private constructor
    }
}


public class GFG
{

    public static void main(String[] args)
    {
        try
        {
            Singleton instance1 = Singleton.instance;
            ObjectOutput out
```

```
                = new ObjectOutputStream(new FileOutputStream("file.text"));
            out.writeObject(instance1);
            out.close();

            // deserailize from file to object
            ObjectInput in
                = new ObjectInputStream(new FileInputStream("file.text"));

            Singleton instance2 = (Singleton) in.readObject();
            in.close();

            System.out.println("instance1 hashCode:- "
                                                + instance1.hashCode());
            System.out.println("instance2 hashCode:- "
                                                + instance2.hashCode());
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}


Output:-
instance1 hashCode:- 1550089733
instance2 hashCode:- 865113938
```

As you can see, hashCode of both instances is different, hence there are 2 objects of a singleton class. Thus, the class is no more singleton.

Overcome serialization issue:- To overcome this issue, we have to implement method readResolve() method.

```
 // Java code to remove the effect of
// Serialization on singleton classes
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Singleton implements Serializable
{
    // public instance initialized when loading the class
    public static Singleton instance = new Singleton();
```

```
    private Singleton()
    {
        // private constructor
    }

    // implement readResolve method
    protected Object readResolve()
    {
        return instance;
    }
}

public class GFG
{

    public static void main(String[] args)
    {
        try
        {
            Singleton instance1 = Singleton.instance;
            ObjectOutput out
                = new ObjectOutputStream(new FileOutputStream("file.text"));
            out.writeObject(instance1);
            out.close();

            // deserailize from file to object
            ObjectInput in
                = new ObjectInputStream(new FileInputStream("file.text"));
            Singleton instance2 = (Singleton) in.readObject();
            in.close();

            System.out.println("instance1 hashCode:- "
                                            + instance1.hashCode());
            System.out.println("instance2 hashCode:- "
                                            + instance2.hashCode());
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

Output:-
instance1 hashCode:- 1550089733
instance2 hashCode:- 1550089733
```

Above both hashcodes are same hence no other instance is created.

3. **Cloning:** Cloning is a concept to create duplicate objects. Using clone we can create copy of object. Suppose, we ceate clone of a singleton object, then it wil create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

```
 // JAVA code to explain cloning
// issue with singleton
class SuperClass implements Cloneable
{
  int i = 10;

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return super.clone();
  }
}

// Singleton class
class Singleton extends SuperClass
{
  // public instance initialized when loading the class
  public static Singleton instance = new Singleton();

  private Singleton()
  {
    // private constructor
  }
}

public class GFG
{
  public static void main(String[] args) throws CloneNotSupportedException
  {
    Singleton instance1 = Singleton.instance;
    Singleton instance2 = (Singleton) instance1.clone();
    System.out.println("instance1 hashCode:- "
                       + instance1.hashCode());
    System.out.println("instance2 hashCode:- "
                       + instance2.hashCode());
  }
}


Output :-
instance1 hashCode:- 366712642
instance2 hashCode:- 1829164700
```

Two different hashCode means there are 2 different objects of singleton class.

Overcome Cloning issue:- To overcome this issue, override clone() method and throw an exception from clone method that is CloneNotSupportedException. Now whenever user will try to create clone of singleton object, it will throw exception and hence our class remains singleton.

```
 // JAVA code to explain overcome
// cloning issue with singleton
class SuperClass implements Cloneable
{
  int i = 10;

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return super.clone();
  }
}

// Singleton class
class Singleton extends SuperClass
{
  // public instance initialized when loading the class
  public static Singleton instance = new Singleton();

  private Singleton()
  {
    // private constructor
  }

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    throw new CloneNotSupportedException();
  }
}

public class GFG
{
  public static void main(String[] args) throws CloneNotSupportedException
  {
    Singleton instance1 = Singleton.instance;
    Singleton instance2 = (Singleton) instance1.clone();
    System.out.println("instance1 hashCode:- "
                       + instance1.hashCode());
    System.out.println("instance2 hashCode:- "
                       + instance2.hashCode());
  }
```

```
}


Output:-
Exception in thread "main" java.lang.CloneNotSupportedException
    at GFG.Singleton.clone(GFG.java:29)
    at GFG.GFG.main(GFG.java:38)
```

Now we have stopped user to create clone of singleton class. If you don;t want to throw exception you can also return the same instance from clone method.

```
 // JAVA code to explain overcome
// cloning issue with singleton
class SuperClass implements Cloneable
{
  int i = 10;

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return super.clone();
  }
}

// Singleton class
class Singleton extends SuperClass
{
  // public instance initialized when loading the class
  public static Singleton instance = new Singleton();

  private Singleton()
  {
    // private constructor
  }

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return instance;
  }
}

public class GFG
{
  public static void main(String[] args) throws CloneNotSupportedException
  {
    Singleton instance1 = Singleton.instance;
    Singleton instance2 = (Singleton) instance1.clone();
```

```
        System.out.println("instance1 hashCode:- "
                                + instance1.hashCode());
        System.out.println("instance2 hashCode:- "
                                + instance2.hashCode());
    }
}
```

```
Output:-
instance1 hashCode:- 366712642
instance2 hashCode:- 366712642
```

Now, as hashcode of both the instances is same that means they represent a single instance.

Source

https://www.geeksforgeeks.org/prevent-singleton-pattern-reflection-serialization-cloning/

Chapter 31

Implementing Iterator pattern of a single Linked List

Implementing Iterator pattern of a single Linked List - GeeksforGeeks

STL is one of the pillars of C++. It makes life lot easier, especially when your focus is on problem solving and you don't want to spend time in implementing something that is already available which guarantees a robust solution. One of the key aspects of Software Engineering is to avoid reinventing the wheel. Reusability is **always** preferred.

While relying on library functions directly impacts our efficiency, without having a proper understanding of how it works sometimes loses meaning of the engineering efficiency we keep on talking. A wrongly chosen data structure may come back sometime in future to haunt us. The solution is simple. Use library methods, but know how does it handles operations under the hood.

Enough said! Today we will look on how we can implement our own **Iterator pattern of a single Linked List**. So, here is how an STL implementation of Linked List looks like:

```
 #include <iostream>
#include <vector>
using namespace std;

int main()
{
    // creating  a list
    vector<int> list;

    // elements to be added at the end.
    // in the above created list.
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);
```

```
    // elements of list are retrieved through iterator.
    for (vector<int>::iterator it = list.begin();
                                it != list.end(); ++it)
        cout << *it << " ";

    return 0;
}
```

Output

```
1 2 3
```

One of the beauty of **cin** and **cout** is that they don't demand format specifiers to work with the type of data. This combined with templates make the code much cleaner and readable. Although I prefer naming method in C++ start with caps, this implementation follows STL rules to mimic exact set of method calls, viz push_back, begin, end.

Here is our own implementation of LinkedList and its Iterator pattern:

```
 // C++ program to implement Custom Linked List and
// iterator pattern.
#include <iostream>
using namespace std;

// Custom class to handle Linked List operations
// Operations like push_back, push_front, pop_back,
// pop_front, erase, size can be added here
template <typename T>
class LinkedList
{
    // Forward declaration
    class Node;

public:
    LinkedList<T>() noexcept
    {
        // caution: static members can't be
        // initialized by initializer list
        m_spRoot = nullptr;
    }

    // Forward declaration must be done
    // in the same access scope
    class Iterator;

    // Root of LinkedList wrapped in Iterator type
    Iterator begin()
```

```
{
    return Iterator(m_spRoot);
}

// End of LInkedList wrapped in Iterator type
Iterator end()
{
    return Iterator(nullptr);
}

// Adds data to the end of list
void push_back(T data);

void Traverse();

// Iterator class can be used to
// sequentially access nodes of linked list
class Iterator
{
public:
Iterator() noexcept :
    m_pCurrentNode (m_spRoot) { }

Iterator(const Node* pNode) noexcept :
    m_pCurrentNode (pNode) { }

    Iterator& operator=(Node* pNode)
    {
        this->m_pCurrentNode = pNode;
        return *this;
    }

    // Prefix ++ overload
    Iterator& operator++()
    {
        if (m_pCurrentNode)
            m_pCurrentNode = m_pCurrentNode->pNext;
        return *this;
    }

    // Postfix ++ overload
    Iterator operator++(int)
    {
        Iterator iterator = *this;
        ++*this;
        return iterator;
    }
```

```
        bool operator!=(const Iterator& iterator)
        {
            return m_pCurrentNode != iterator.m_pCurrentNode;
        }

        int operator*()
        {
            return m_pCurrentNode->data;
        }

    private:
        const Node* m_pCurrentNode;
    };

private:

    class Node
    {
        T data;
        Node* pNext;

        // LinkedList class methods need
        // to access Node information
        friend class LinkedList;
    };

    // Create a new Node
    Node* GetNode(T data)
    {
        Node* pNewNode = new Node;
        pNewNode->data = data;
        pNewNode->pNext = nullptr;

        return pNewNode;
    }

    // Return by reference so that it can be used in
    // left hand side of the assignment expression
    Node*& GetRootNode()
    {
        return m_spRoot;
    }

    static Node* m_spRoot;
};

template <typename T>
/*static*/ typename LinkedList<T>::Node* LinkedList<T>::m_spRoot = nullptr;
```

```
template <typename T>
void LinkedList<T>::push_back(T data)
{
    Node* pTemp = GetNode(data);
    if (!GetRootNode())
    {
        GetRootNode() = pTemp;
    }
    else
    {
        Node* pCrawler = GetRootNode();
        while (pCrawler->pNext)
        {
            pCrawler = pCrawler->pNext;
        }

        pCrawler->pNext = pTemp;
    }
}

template <typename T>
void LinkedList<T>::Traverse()
{
    Node* pCrawler = GetRootNode();

    while (pCrawler)
    {
        cout << pCrawler->data << " ";
        pCrawler = pCrawler->pNext;
    }

    cout << endl;
}

//Driver program
int main()
{
    LinkedList<int> list;

    // Add few items to the end of LinkedList
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);

    cout << "Traversing LinkedList through method" << endl;
    list.Traverse();
```

```
    cout << "Traversing LinkedList through Iterator" << endl;
    for ( LinkedList<int>::Iterator iterator = list.begin();
            iterator != list.end(); iterator++)
    {
        cout << *iterator << " ";
    }

    cout << endl;

    return 0;
}
```

Output:

```
Traversing LinkedList through method
1 2 3
Traversing LinkedList through Iterator
1 2 3
```

Exercise:
The above implementation works well when we have one data. Extend this code to work for set of data wrapped in a class.

Source

https://www.geeksforgeeks.org/implementing-iterator-pattern-of-a-single-linked-list/

Chapter 32

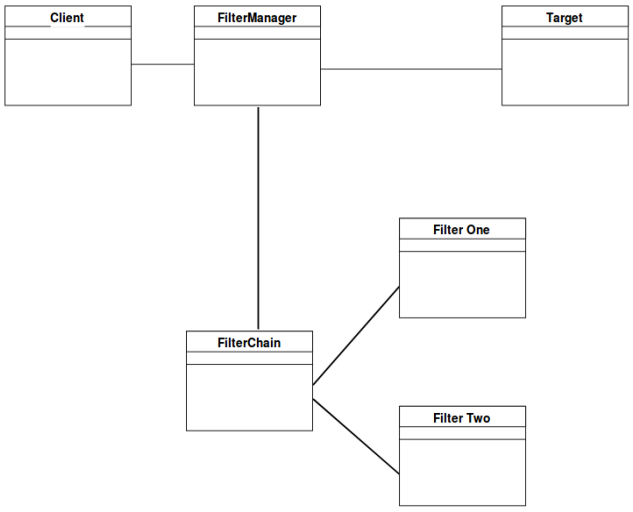
Intercepting Filter Pattern

Intercepting Filter Pattern - GeeksforGeeks

Preprocessing and postprocessing of a request refer to actions taken before and after the core processing of that request. Some of these actions determine whether processing will continue, while others manipulate the incoming or outgoing data stream into a form suitable for further processing.

The classic solution consists of a series of conditional checks, with any failed check aborting the request. Nested if/else statements are a standard strategy, but this solution leads to code fragility and a copy-and-paste style of programming, because the flow of the filtering and the action of the filters is compiled into the application.
The key to solving this problem in a flexible and unobtrusive manner is to have a simple mechanism for adding and removing processing components, in which each component completes a specific filtering action.

UML Diagram Intercepting Filter Pattern



Design components

- **Filter Manager :** The FilterManager manages filter processing. It creates the FilterChain with the appropriate filters, in the correct order, and initiates processing.
- **FilterChain :** The FilterChain is an ordered collection of independent filters.
- **FilterOne, FilterTwo :** These are the individual filters that are mapped to a target. The FilterChain coordinates their processing.
- **Target :** The Target is the resource requested by the client.

Let's see an example of Intercepting Filter Pattern.

```
 // Java program to illustrate
// Intercepting Filter Pattern
import java.util.ArrayList;
import java.util.List;

interface Filter
{
    public void execute(String request);
}

class AuthenticationFilter implements Filter
{
    public void execute(String request)
    {
        System.out.println("Authenticating : " + request);
    }
}

class DebugFilter implements Filter
{
    public void execute(String request)
    {
        System.out.println("Log: " + request);
    }
}

class Target
{
    public void execute(String request)
    {
        System.out.println("Executing : " + request);
    }
}

class FilterChain
{
```

```
    private List<Filter> filters = new ArrayList<Filter>();
    private Target target;

    public void addFilter(Filter filter)
    {
        filters.add(filter);
    }

    public void execute(String request)
    {
        for (Filter filter : filters)
        {
            filter.execute(request);
        }
        target.execute(request);
    }

    public void setTarget(Target target)
    {
        this.target = target;
    }
}

class FilterManager
{
    FilterChain filterChain;

   public FilterManager(Target target)
   {
       filterChain = new FilterChain();
        filterChain.setTarget(target);
   }
   public void setFilter(Filter filter)
   {
       filterChain.addFilter(filter);
   }

   public void filterRequest(String request)
   {
       filterChain.execute(request);
   }
}

class Client
{
    FilterManager filterManager;

    public void setFilterManager(FilterManager filterManager)
```

```
    {
        this.filterManager = filterManager;

    }

    public void sendRequest(String request)
    {
        filterManager.filterRequest(request);
    }
}

class InterceptingFilter
{
    public static void main(String[] args)
    {
        FilterManager filterManager = new FilterManager(new Target());
        filterManager.setFilter(new AuthenticationFilter());
        filterManager.setFilter(new DebugFilter());

        Client client = new Client();
        client.setFilterManager(filterManager);
        client.sendRequest("Downloads");

    }
}
```

Output:

```
Authenticating : Downloads
Log: Downloads
Executing : Downloads
```

Advantages :

- **Improved reusability:** Common code is centralized in pluggable components enhancing reuse.
- **Increased flexibility:** Generic common components can be applied and removed declaratively, improving flexibility.

Disadvantages :

- Information sharing is inefficient in intercepting pattern.

Source

https://www.geeksforgeeks.org/intercepting-filter-pattern/

Chapter 33

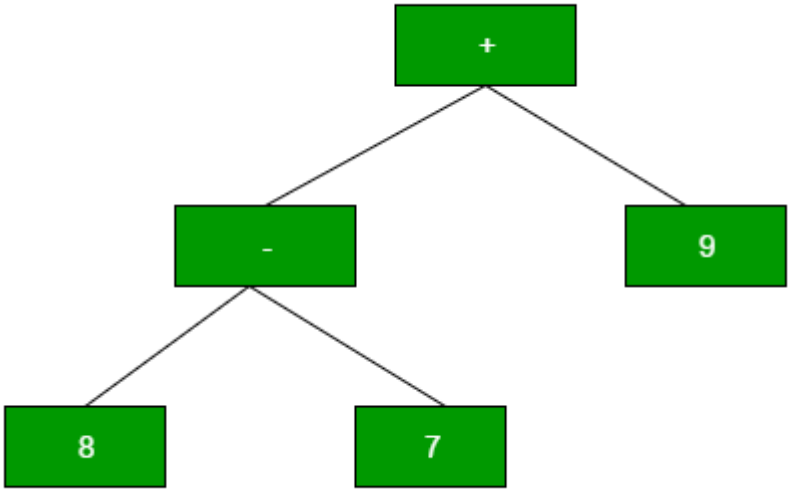
Interpreter Design Pattern

Interpreter Design Pattern - GeeksforGeeks

Interpreter design pattern is one of the **behavioral** design pattern. Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

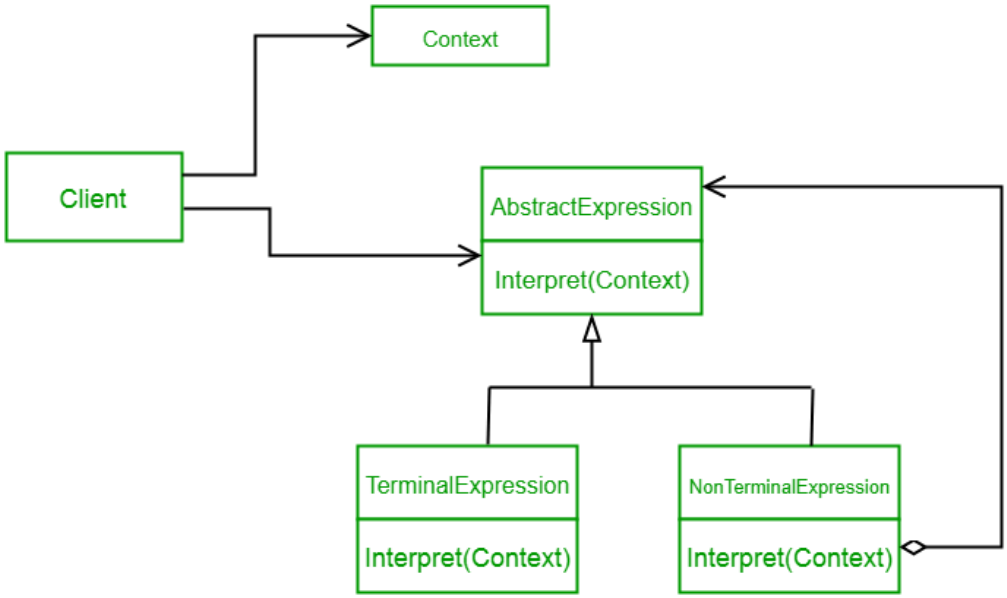
- This pattern involves implementing an expression interface which tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine etc.
- This pattern performs upon a hierarchy of expressions. Each expression here is a terminal or non-terminal.
- The tree structure of Interpreter design pattern is somewhat similar to that defined by the composite design pattern with terminal expressions being leaf objects and non-terminal expressions being composites.
- The tree contains the expressions to be evaluated and is usually generated by a parser. The parser itself is not a part of the interpreter pattern.

For Example :
Here is the hierarchy of expressions for **"+ − 9 8 7"** :



Implementing the Interpreter Pattern

UML Diagram Interpreter Design Pattern



Design components

- **AbstractExpression** (Expression): Declares an interpret() operation that all nodes

(terminal and nonterminal) in the AST overrides.

- **TerminalExpression** (NumberExpression): Implements the interpret() operation for terminal expressions.
- **NonterminalExpression** (AdditionExpression, SubtractionExpression, and MultiplicationExpression): Implements the interpret() operation for all nonterminal expressions.
- **Context** (String): Contains information that is global to the interpreter. It is this String expression with the Postfix notation that has to be interpreted and parsed.
- **Client** (ExpressionParser): Builds (or is provided) the AST assembled from TerminalExpression and NonTerminalExpression. The Client invokes the interpret() operation.

Let's see an example of Interpreter Design Pattern.

```
 // Expression interface used to
// check the interpreter.
interface Expression
{
    boolean interpreter(String con);
}

// TerminalExpression class implementing
// the above interface. This interpreter
// just check if the data is same as the
// interpreter data.
class TerminalExpression implements Expression
{
    String data;

    public TerminalExpression(String data)
    {
        this.data = data;
    }

    public boolean interpreter(String con)
    {
        if(con.contains(data))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
// OrExpression class implementing
// the above interface. This interpreter
// just returns the or condition of the
```

```
// data is same as the interpreter data.
class OrExpression implements Expression
{
    Expression expr1;
    Expression expr2;

    public OrExpression(Expression expr1, Expression expr2)
    {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public boolean interpreter(String con)
    {
        return expr1.interpreter(con) || expr2.interpreter(con);
    }
}

// AndExpression class implementing
// the above interface. This interpreter
// just returns the And condition of the
// data is same as the interpreter data.
class AndExpression implements Expression
{
    Expression expr1;
    Expression expr2;

    public AndExpression(Expression expr1, Expression expr2)
    {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public boolean interpreter(String con)
    {
        return expr1.interpreter(con) && expr2.interpreter(con);
    }
}

// Driver class
class InterpreterPattern
{

    public static void main(String[] args)
    {
        Expression person1 = new TerminalExpression("Kushagra");
        Expression person2 = new TerminalExpression("Lokesh");
        Expression isSingle = new OrExpression(person1, person2);

        Expression vikram = new TerminalExpression("Vikram");
```

```
        Expression committed = new TerminalExpression("Committed");
        Expression isCommitted = new AndExpression(vikram, committed);

        System.out.println(isSingle.interpreter("Kushagra"));
        System.out.println(isSingle.interpreter("Lokesh"));
        System.out.println(isSingle.interpreter("Achint"));

        System.out.println(isCommitted.interpreter("Committed, Vikram"));
        System.out.println(isCommitted.interpreter("Single, Vikram"));

    }
}
```

Output:

```
true
true
false
true
false
```

In the above code , We are creating an interface **Expression** and **concrete** classes implementing the Expression interface. A class **TerminalExpression** is defined which acts as a main interpreter and other classes **OrExpression**, **AndExpression** are used to create combinational expressions.

Advantages

- It's easy to change and extend the grammar. Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar. Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.
- Implementing the grammar is easy, too. Classes defining nodes in the abstract syntax tree have similar implementations. These classes are easy to write, and often their generation can be automated with a compiler or parser generator.

Disadvantages

- Complex grammars are hard to maintain. The Interpreter pattern defines at least one class for every rule in the grammar. Hence grammars containing many rules can be hard to manage and maintain.

Source

https://www.geeksforgeeks.org/interpreter-design-pattern/

Chapter 34

Iterator Pattern

Iterator Pattern - GeeksforGeeks

Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language. Each collection must provide an iterator that lets it iterate through its objects. However while doing so it should make sure that it does not expose its implementation.
Suppose we are building an application that requires us to maintain a list of notifications. Eventually, some part of your code will require to iterate over all notifications. If we implemented your collection of notifications as array you would iterate over them as:

```
// If a simple array is used to store notifications
for (int i = 0; i < notificationList.length; i++)
     Notification notification = notificationList[i]);
```

```
// If ArrayList is Java is used, then we would iterate
// over them as:
for (int i = 0; i < notificationList.size(); i++)
    Notification notification = (Notification)notificationList.get(i);
```

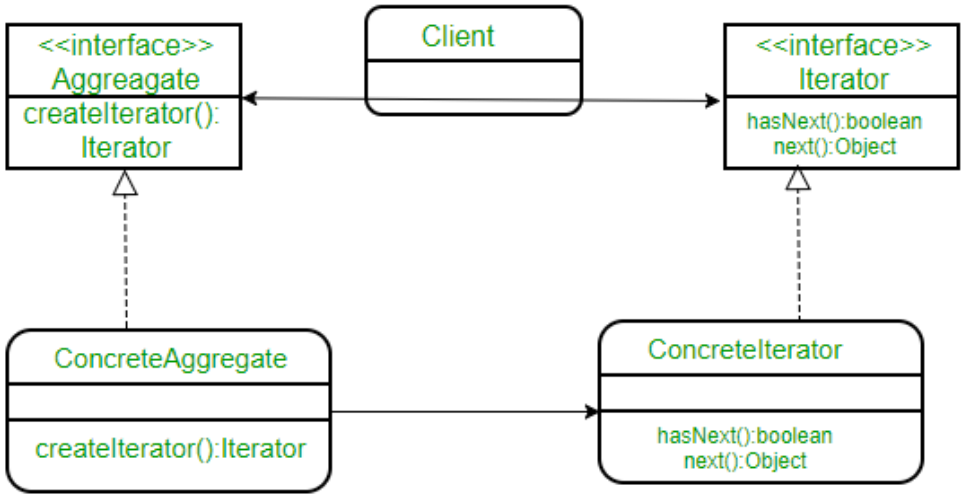
And if it were some other collection like set, tree etc. way of iterating would change slightly. Now, what if we build an iterator that provides a generic way of iterating over a collection independent of its type.

```
// Create an iterator
Iterator iterator = notificationList.createIterator();

// It wouldn't matter if list is Array or ArrayList or
```

```
// anything else.
while (iterator.hasNext())
{
    Notification notification = iterator.next());
}
```

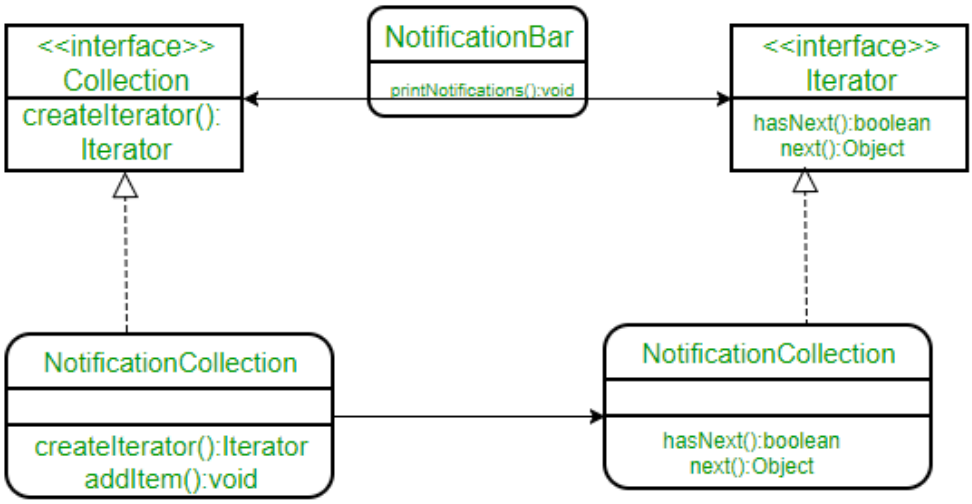
Iterator pattern lets us do just that. Formally it is defined as below:
The iterator pattern provides a way to access the elements of an aggregate object without exposing its underlying representation.

Class Diagram:



Here we have a common interface Aggregate for client as it decouples it from the implementation of your collection of objects. The ConcreteAggregate implements createIterator() that returns iterator for its collection. Each ConcreteAggregate's responsibility is to instantiate a ConcreteIterator that can iterate over its collection of objects. The iterator interface provides a set of methods for traversing or modifying the collection that is in addition to next()/hasNext() it can also provide functions for search, remove etc.
Let's understand this through an example. Suppose we are creating a notification bar in our application that displays all the notifications which are held in a notification collection. NotificationCollection provides an iterator to iterate over its elements without exposing how it has implemented the collection (array in this case) to the Client (NotificationBar).

The class diagram would be:



Below is the Java implementation of the same:

```
 // A Java program to demonstrate implementation
// of iterator pattern with the example of
// notifications

// A simple Notification class
class Notification
{
    // To store notification message
    String notification;

    public Notification(String notification)
    {
        this.notification = notification;
    }
    public String getNotification()
    {
        return notification;
    }
}

// Collection interface
interface Collection
{
    public Iterator createIterator();
}

// Collection of notifications
```

```
class NotificationCollection implements Collection
{
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    Notification[] notificationList;

    public NotificationCollection()
    {
        notificationList = new Notification[MAX_ITEMS];

        // Let us add some dummy notifications
        addItem("Notification 1");
        addItem("Notification 2");
        addItem("Notification 3");
    }

    public void addItem(String str)
    {
        Notification notification = new Notification(str);
        if (numberOfItems >= MAX_ITEMS)
            System.err.println("Full");
        else
        {
            notificationList[numberOfItems] = notification;
            numberOfItems = numberOfItems + 1;
        }
    }

    public Iterator createIterator()
    {
        return new NotificationIterator(notificationList);
    }
}

// We could also use Java.Util.Iterator
interface Iterator
{
    // indicates whether there are more elements to
    // iterate over
    boolean hasNext();

    // returns the next element
    Object next();
}

// Notification iterator
class NotificationIterator implements Iterator
{
```

```
    Notification[] notificationList;

    // maintains curr pos of iterator over the array
    int pos = 0;

    // Constructor takes the array of notifiactionList are
    // going to iterate over.
    public  NotificationIterator (Notification[] notificationList)
    {
        this.notificationList = notificationList;
    }

    public Object next()
    {
        // return next element in the array and increment pos
        Notification notification =  notificationList[pos];
        pos += 1;
        return notification;
    }

    public boolean hasNext()
    {
        if (pos >= notificationList.length ||
            notificationList[pos] == null)
            return false;
        else
            return true;
    }
}

// Contains collection of notifications as an object of
// NotificationCollection
class NotificationBar
{
    NotificationCollection notifications;

    public NotificationBar(NotificationCollection notifications)
    {
        this.notifications = notifications;
    }

    public void printNotifications()
    {
        Iterator iterator = notifications.createIterator();
        System.out.println("-------NOTIFICATION BAR------------");
        while (iterator.hasNext())
        {
            Notification n = (Notification)iterator.next();
```

```
            System.out.println(n.getNotification());
        }
    }
}

// Driver class
class Main
{
    public static void main(String args[])
    {
        NotificationCollection nc = new NotificationCollection();
        NotificationBar nb = new NotificationBar(nc);
        nb.printNotifications();
    }
}
```

Output:

```
-------NOTIFICATION BAR------------
Notification 1
Notification 2
Notification 3
```

Notice that if we would have used ArrayList instead of Array there will not be any change in the client (notification bar) code due to the decoupling achieved by the use of iterator interface.

References:
Head First Design Patterns

Improved By : Shraddhesh Bhandari

Source

https://www.geeksforgeeks.org/iterator-pattern/

Chapter 35

Java Singleton Design Pattern Practices with Examples

Java Singleton Design Pattern Practices with Examples - GeeksforGeeks

In previous articles, we discussed about singleton design pattern and singleton class implementationin detail.
In this article, we will see how we can create singleton classes. After reading this article you will be able to create your singleton class according to your use, simplicity and removed bottlenecks.
There are many ways this can be done in Java. All these ways differs in their implementation of the pattern, but in the end, they all achieve the same end result of a single instance.

1. **Eager initialization:** This is the simplest method of creating a singleton class. In this, object of class is created when it is loaded to the memory by JVM. It is done by assigning the reference an instance directly.
   It can be used when program will always use instance of this class, or the cost of creating the instance is not too large in terms of resources and time.

```
 // Java code to create singleton class by
// Eager Initialization
public class GFG
{
  // public instance initialized when loading the class
  public static GFG instance = new GFG();

  private GFG()
  {
    // private constructor
  }
}
```

   Pros:

(a) Very simple to implement.
(b) No need to implement getInstance() method. Instance can be accessed directly.

Cons:

(a) May lead to resource wastage. Because instance of class is created always, whether it is required or not.
(b) CPU time is also wasted in creation of instance if it is not required.
(c) Exception handling is not possible.

2. **Using static block:** This is also a sub part of Eager initialization. The only difference is object is created in a static block so that we can have access on its creation, like exception handling. In this way also, object is created at the time of class loading.
It can be used when there is a chance of exceptions in creating object with eager initialization.

```
 // Java code to create singleton class
// Using Static block
public class GFG
{
  // public instance
  public static GFG instance;

  private GFG()
  {
    // private constructor
  }

  {
    // static block to initialize instance
    instance = new GFG();
  }
}
```

Pros:

(a) Very simple to implement.
(b) No need to implement getInstance() method. Instance can be accessed directly.
(c) Exceptions can be handled in static block.

Cons:

(a) May lead to resource wastage. Because instance of class is created always, whether it is required or not.
(b) CPU time is also wasted in creation of instance if it is not required.

3. **Lazy initialization:** In this method, object is created only if it is needed. This may prevent resource wastage. An implementation of getInstance() method is required which return the instance. There is a null check that if object is not created then create, otherwise return previously created. To make sure that class cannot be instantiated in any other way, constructor is made final. As object is created with in a method, it ensures that object will not be created until and unless it is required. Instance is kept private so that no one can access it directly.
   It can be used in a single threaded environment because multiple threads can break singleton property because they can access get instance method simultaneously and create multiple objects.

```
 //Java Code to create singleton class
// With Lazy initialization
public class GFG
{
  // private instance, so that it can be
  // accessed by only by getInstance() method
  private static GFG instance;

  private GFG()
  {
    // private constructor
  }

  //method to return instance of class
  public static GFG getInstance()
  {
    if (instance == null)
    {
      // if instance is null, initialize
      instance = new GFG();
    }
    return instance;
  }
}
```

Pros:

(a) Object is created only if it is needed. It may overcome resource overcome and wastage of CPU time.
(b) Exception handling is also possible in method.

Cons:

(a) Every time a condition of null has to be checked.
(b) instance can't be accessed directly.
(c) In multithreaded environment, it may break singleton property.

4. **Thread Safe Singleton:** A thread safe singleton in created so that singleton property is maintained even in multithreaded environment. To make a singleton class thread-safe, getInstance() method is made synchronized so that multiple threads can't access it simultaneously.

```
 // Java program to create Thread Safe
// Singleton class
public class GFG
{
  // private instance, so that it can be
  // accessed by only by getInstance() method
  private static GFG instance;

  private GFG()
  {
    // private constructor
  }

 //synchronized method to control simultaneous access
  synchronized public static GFG getInstance()
  {
    if (instance == null)
    {
      // if instance is null, initialize
      instance = new GFG();
    }
    return instance;
  }
}
```

Pros:

(a) Lazy initialization is possible.
(b) It is also thread safe.

Cons:

(a) getInstance() method is synchronized so it causes slow performance as multiple threads can't access it simultaneously.

5. **Lazy initialization with Double check locking:** In this mechanism, we overcome the overhead problem of synchronized code. In this method, getInstance is not synchronized but the block which creates instance is synchronized so that minimum number of threads have to wait and that's only for first time.

```
 // Java code to explain double check locking
public class GFG
{
  // private instance, so that it can be
```

```
  // accessed by only by getInstance() method
  private static GFG instance;

  private GFG()
  {
    // private constructor
  }

  public static GFG getInstance()
  {
    if (instance == null)
    {
      //synchronized block to remove overhead
      synchronized (GFG.class)
      {
        if(instance==null)
        {
          // if instance is null, initialize
          instance = new GFG();
        }

      }
    }
    return instance;
  }
}
```

Pros:

(a) Lazy initialization is possible.
(b) It is also thread safe.
(c) Performance reduced because of synchronized keyword is overcome.

Cons:

(a) First time, it can affect performance.

As cons. of double check locking method is bearable so it can be used for high performance multi-threaded applications.

6. **Bill Pugh Singleton Implementation:** Prior to Java5, memory model had a lot of issues and above methods caused failure in certain scenarios in multithreaded environment. So, Bill Pugh suggested a concept of inner static classes to use for singleton.

```
 // Java code for Bill Pugh Singleton Implementaion
public class GFG
{

  private GFG()
```

```
  {
    // private constructor
  }

  // Inner class to provide instance of class
  private static class BillPughSingleton
  {
    private static final GFG INSTANCE = new GFG();
  }

  public static GFG getInstance()
  {
    return BillPughSingleton.INSTANCE;
  }
}
```

When the singleton class is loaded, inner class is not loaded and hence doesn't create object when loading the class. Inner class is created only when getInstance() method is called. So it may seem like eager initialization but it is lazy initialization.
This is the most widely used approach as it doesn't use synchronization.

When to use What

1. Eager initialization is easy to implement but it may cause resource and CPU time wastage. Use it only if cost of initializing a class is less in terms of resources or your program will always need the instance of class.
2. By using Static block in Eager initialization we can provide exception handling and also can control over instance.
3. Using synchronized we can create singleton class in multi-threading environment also but it can cause slow performance, so we can use Double check locking mechanism.
4. Bill Pugh implementation is most widely used approach for singleton classes. Most developers prefer it because of its simplicity and advantages.

Source

https://www.geeksforgeeks.org/java-singleton-design-pattern-practices-examples/

Chapter 36

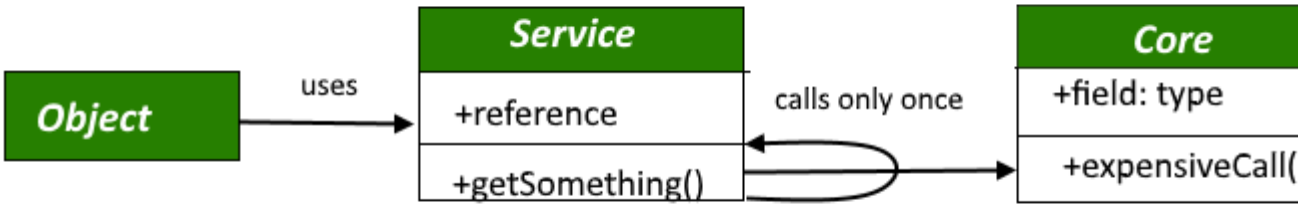
Lazy Loading Design Pattern

Lazy Loading Design Pattern - GeeksforGeeks

Lazy loading is a concept where we delay the loading of object until the point where we need it.

- Lazy loading is just a fancy name given to the process of initializing a class when it's actually needed.
- In simple words, Lazy loading is a software design pattern where the initialization of an object occurs only when it is actually needed and not before to preserve simplicity of usage and improve performance.
- Lazy loading is essential when the cost of object creation is very high and the use of the object is very rare. So this is the scenario where it's worth implementing lazy loading.The fundamental idea of lazy loading is to load object/data when needed.

For Example, Suppose You are creating an application in which there is a Company object and this object contains a list of employees of the company in a ContactList object. There could be thousands of employees in a company. Loading the Company object from the database along with the list of all its employees in the ContactList object could be very time consuming. In some cases you don't even require the list of the employees, but you are forced to wait until the company and its list of employees loaded into the memory.
One way to save time and memory is to avoid loading of the employee objects until required and this is done using the **Lazy Loading Design Pattern**.



There are four common implementations of Lazy Loading pattern :

1. Virtual proxy
2. Lazy initialization
3. Ghost
4. Value holder

Virtual proxy

The Virtual Proxy pattern is a memory saving technique that recommends postponing an object creation until it is needed. It is used when creating an object the is expensive in terms of memory usage or processing involved.

```
 // Java program to illustrate
// virtual proxy in
// Lazy Loading Design Pattern
import java.util.List;
import java.util.ArrayList;

interface ContactList
{
public List<Employee> getEmployeeList();
}

class Company {
    String companyName;
    String companyAddress;
    String companyContactNo;
    ContactList contactList;

    public Company(String companyName, String companyAddress,
```

```
            String companyContactNo, ContactList contactList)
    {
        this.companyName = companyName;
        this.companyAddress = companyAddress;
        this.companyContactNo = companyContactNo;
        this.contactList = contactList;
    }

    public String getCompanyName()
    {
        return companyName;
    }
    public String getCompanyAddress()
    {
        return companyAddress;
    }
    public String getCompanyContactNo()
    {
        return companyContactNo;
    }
    public ContactList getContactList()
    {
        return contactList;
    }

}

class ContactListImpl implements ContactList {
    public List<Employee> getEmployeeList()
    {
        return getEmpList();
    }
    private static List<Employee> getEmpList()
    {
        List<Employee> empList = new ArrayList<Employee>(5);

        empList.add(new Employee("Lokesh", 2565.55, "SE"));
        empList.add(new Employee("Kushagra", 22574, "Manager"));
        empList.add(new Employee("Susmit", 3256.77, "G4"));
        empList.add(new Employee("Vikram", 4875.54, "SSE"));
        empList.add(new Employee("Achint", 2847.01, "SE"));

        return empList;
    }
}

class ContactListProxyImpl implements ContactList {
    private ContactList contactList;
```

```
    public List<Employee> getEmployeeList()
    {
        if (contactList == null) {
            System.out.println("Fetching list of employees");
            contactList = new ContactListImpl();
        }
        return contactList.getEmployeeList();
    }
}

class Employee {
    private String employeeName;

    private double employeeSalary;
    private String employeeDesignation;

    public Employee(String employeeName,
             double employeeSalary, String employeeDesignation)
    {
        this.employeeName = employeeName;
        this.employeeSalary = employeeSalary;
        this.employeeDesignation = employeeDesignation;
    }
    public String getEmployeeName()
    {
        return employeeName;
    }
    public double getEmployeeSalary()
    {
        return employeeSalary;
    }
    public String getEmployeeDesignation()
    {
        return employeeDesignation;
    }
    public String toString()
    {
        return "Employee Name: " + employeeName + ",
               EmployeeDesignation : " + employeeDesignation + ",
               Employee Salary : " + employeeSalary;
    }
}

class LazyLoading {
    public static void main(String[] args)
    {
        ContactList contactList = new ContactListProxyImpl();
        Company company = new Company
```

```
        ("Geeksforgeeks", "India", "+91-011-28458965", contactList);

        System.out.println("Company Name: " + company.getCompanyName());
        System.out.println("Company Address: " + company.getCompanyAddress());
        System.out.println("Company Contact No.: " + company.getCompanyContactNo());
        System.out.println("Requesting for contact list");

        contactList = company.getContactList();
        List<Employee> empList = contactList.getEmployeeList();
        for (Employee emp : empList) {
            System.out.println(emp);
        }
    }
}
```

Output:

```
Company Name: ABC Company
Company Address: India
Company Contact No.: +91-011-28458965
Requesting for contact list
Fetching list of employees
Employee Name: Lokesh, EmployeeDesignation: SE, Employee Salary: 2565.55
Employee Name: Kushagra, EmployeeDesignation: Manager, Employee Salary: 22574.0
Employee Name: Susmit, EmployeeDesignation: G4, Employee Salary: 3256.77
Employee Name: Vikram, EmployeeDesignation: SSE, Employee Salary: 4875.54
Employee Name: Achint, EmployeeDesignation: SE, Employee Salary: 2847.01
```

Now, In the above code have a Company object is created with a proxy ContactList object. At this time, the Company object holds a proxy reference, not the real ContactList object's reference, so there no employee list loaded into the memory.

Lazy Initialization

The Lazy Initialization technique consists of checking the value of a class field when it's being used. If that value equals to null then that field gets loaded with the proper value before it is returned.
Here is the example :

```
 // Java program to illustrate
// Lazy Initialization in
// Lazy Loading Design Pattern
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

enum CarType {
```

```
    none,
    Audi,
    BMW,
}

class Car {
    private static Map<CarType, Car> types = new HashMap<>();

    private Car(CarType type) {}

    public static Car getCarByTypeName(CarType type)
    {
        Car Car;

        if (!types.containsKey(type)) {
            // Lazy initialisation
            Car = new Car(type);
            types.put(type, Car);
        } else {
            // It's available currently
            Car = types.get(type);
        }

        return Car;
    }

    public static Car getCarByTypeNameHighConcurrentVersion(CarType type)
    {
        if (!types.containsKey(type)) {
            synchronized(types)
            {
                // Check again, after having acquired the lock to make sure
                // the instance was not created meanwhile by another thread
                if (!types.containsKey(type)) {
                    // Lazy initialisation
                    types.put(type, new Car(type));
                }
            }
        }

        return types.get(type);
    }

    public static void showAll()
    {
        if (types.size() > 0) {

            System.out.println("Number of instances made = " + types.size());
```

```
            for (Entry<CarType, Car> entry : types.entrySet()) {
                String Car = entry.getKey().toString();
                Car = Character.toUpperCase(Car.charAt(0)) + Car.substring(1);
                System.out.println(Car);
            }

            System.out.println();
        }
    }
}

class Program {
    public static void main(String[] args)
    {
        Car.getCarByTypeName(CarType.BMW);
        Car.showAll();
        Car.getCarByTypeName(CarType.Audi);
        Car.showAll();
        Car.getCarByTypeName(CarType.BMW);
        Car.showAll();
    }
}
```

Output :

```
Number of instances made = 1
BMW

Number of instances made = 2
BMW
Audi

Number of instances made = 2
BMW
Audi
```

Value Holder

Basically, A value holder is a generic object that handles the lazy loading behavior and appears in place of the object's data fields.When the user needs to access it, they simply ask the value holder for its value by calling the GetValue method. At that time (and only then), the value gets loaded from a database or from a service.(this is not always needed).

```
 // Java function to illustrate
// Lazy Initialization in
// Lazy Loading Design Pattern
```

```
public class ValueHolder<T> {
    private T value;
    private readonly Func<object, T> valueRetrieval;

    // Constructor
    public ValueHolder(Func<object, T> valueRetrieval)
    {
        valueRetrieval = this.valueRetrieval;
    }

    // We'll use the signature "GetValue" for convention
    public T GetValue(object parameter)
    {
        if (value == null)
            value = valueRetrieval(parameter);
        return value;
    }
}
```

Note : The main drawback of this approach is that the user has to know that a value holder is expected.

Ghost

A ghost is the object that is to be loaded in a partial state. It corresponds to the real object but not in its full state. It may be empty or it may contain just some fields (such as the ID). When the user tries to access some fields that haven't been loaded yet, the ghost object fully initializes itself (this is not always needed).

For example, Let's consider that a developer what add an online form so that any user can request content via that online form. At the time of creation all we know is that content will be accessed but what action or content is unknown to the user.

```
 $userData = array(
    "UID" = > uniqid(),
    "requestTime" = > microtime(true),
    "dataType" = > "",
    "request" = > "");

if (isset($_POST['data']) && $userData) {
    //...
}
```

In the above PHP example, the content from the online form can be accessed to the user in the form of text file or any source.

- **UID** is the unique id for the every particular user.
- **requestTime** is the time when user requested the content from the online form.
- **dataType** is the type of data. Mostly it is text but depends on the form.

- **request** is the boolean function to notify the user about the request being completed or not.

Advantages

- This approach is a faster application start-up time as it is not required to created and load all of the application objects.

Disadvantages

- The code becomes complicated as we need to check if loading is needed or not. So this may cause slower in the performance.

Source

https://www.geeksforgeeks.org/lazy-loading-design-pattern/

Chapter 37

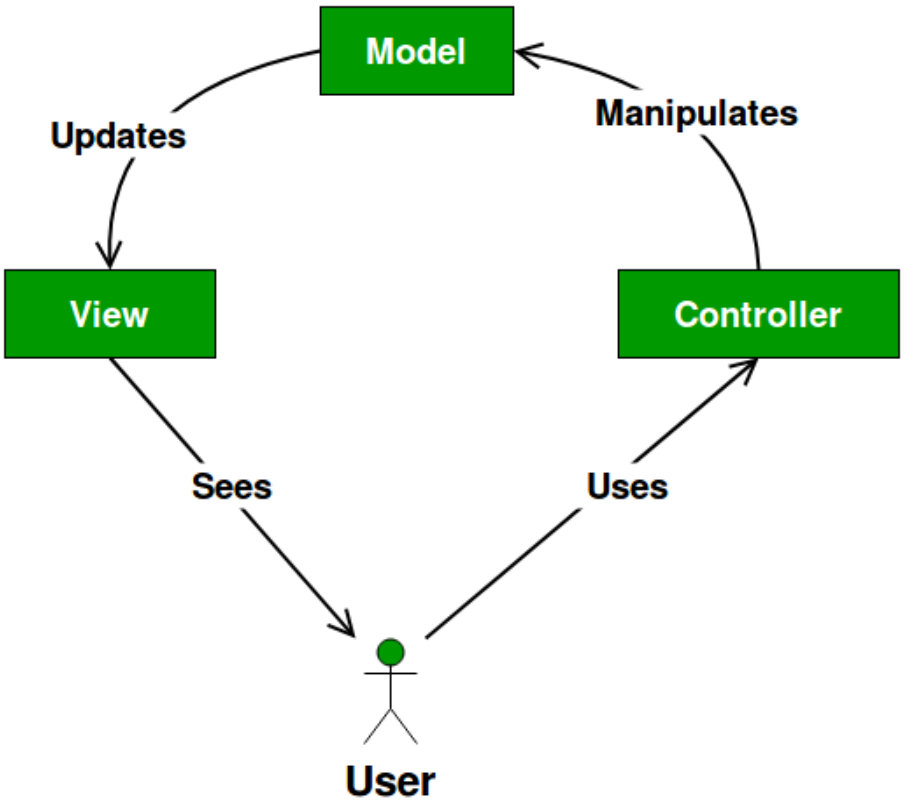
MVC Design Pattern

MVC Design Pattern - GeeksforGeeks

The **Model View Controller** (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

MVC is more of an architectural pattern, but not for complete application. MVC mostly relates to the UI / interaction layer of an application. You're still going to need business logic layer, maybe some service layer and data access layer.

UML Diagram MVC Design Pattern



Design components

- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user.
- The **View** presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.
- The **Controller** exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.

Let's see an example of MVC Design Pattern.

```
 class Student
{
    private String rollNo;
    private String name;
```

```
    public String getRollNo()
    {
        return rollNo;
    }

    public void setRollNo(String rollNo)
    {
        this.rollNo = rollNo;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}

class StudentView
{
    public void printStudentDetails(String studentName, String studentRollNo)
    {
        System.out.println("Student: ");
        System.out.println("Name: " + studentName);
        System.out.println("Roll No: " + studentRollNo);
    }
}

class StudentController
{
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view)
    {
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name)
    {
        model.setName(name);
    }
```

```
    public String getStudentName()
    {
        return model.getName();
    }

    public void setStudentRollNo(String rollNo)
    {
        model.setRollNo(rollNo);
    }

    public String getStudentRollNo()
    {
        return model.getRollNo();
    }

    public void updateView()
    {
        view.printStudentDetails(model.getName(), model.getRollNo());
    }
}

class MVCPattern
{
    public static void main(String[] args)
    {
        Student model  = retriveStudentFromDatabase();

        StudentView view = new StudentView();

        StudentController controller = new StudentController(model, view);

        controller.updateView();

        controller.setStudentName("Vikram Sharma");

        controller.updateView();
    }

    private static Student retriveStudentFromDatabase()
    {
        Student student = new Student();
        student.setName("Lokesh Sharma");
        student.setRollNo("15UCS157");
        return student;
    }

}
```

Output:

```
Student:
Name: Lokesh Sharma
Roll No: 15UCS157
Student:
Name: Vikram Sharma
Roll No: 15UCS157
```

Advantages

- Multiple developers can work simultaneously on the model, controller and views.
- MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Models can have multiple views.

Disadvantages

- The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Knowledge on multiple technologies becomes the norm. Developers using MVC need to be skilled in multiple technologies.

Source

https://www.geeksforgeeks.org/mvc-design-pattern/

Chapter 38

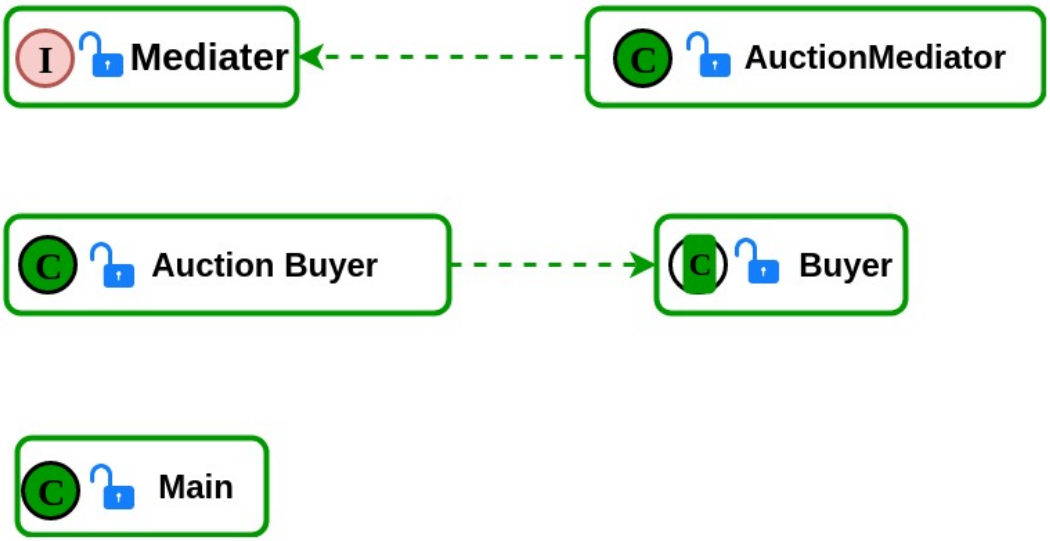
Mediator Design Pattern

Mediator Design Pattern - GeeksforGeeks

The mediator design pattern defines an object that encapsulates how a set of objects interact. The Mediator is a behavioral pattern (like the Observer or the Visitor pattern) because it can change the program's running behavior.
We are used to see programs that are made made up of a large number of classes. However, as more classes are added to a program, the problem of communication between these classes may become more complex.
Because of this, the maintenance becomes a big problem that we need to solve in this way or another.
Like in many other Design Patterns, The mediator pattern comes to solve the problem. It makes the communication between objects encapsulated with a mediator object.
Objects don't communicate directly with each other, but instead, they communicate through the mediator.

Mediator pattern implementation in Java

This program illustrate an auction. The Auction Mediator is responsible for adding the buyers, and after each buyer bid a certain amount for the item, the mediator know who won the auction.
Class diagram:




```
 // Java code to illustrate Mediator Pattern
// All public class codes should be put in
// different files.

public interface Mediator {

    // The mediator interface
    public void addBuyer(Buyer buyer);
    public void findHighestBidder();
}

public class AuctionMediator implements Mediator {

    // this class implements the interface and holds
    // all the buyers in a Array list.
    // We can add buyers and find the highest bidder
    private ArrayList buyers;

    public AuctionMediator()
    {
        buyers = new ArrayList<>();
    }

    @Override
    public void addBuyer(Buyer buyer)
    {
        buyers.add(buyer);
        System.out.println(buyer.name + " was added to" +
                "the buyers list.");
    }
```

```
    @Override
    public void findHighestBidder()
    {
        int maxBid = 0;
        Buyer winner = null;
        for (Buyer b : buyers) {
            if (b.price > maxBid) {
                maxBid = b.price;
                winner = b;
            }
        }
        System.out.println("The auction winner is " + winner.name +
        ". He paid " + winner.price + "$ for the item.");
    }
}

public abstract class Buyer {

    // this class holds the buyer
    protected Mediator mediator;
    protected String name;
    protected int price;

    public Buyer(Mediator med, String name)
    {
        this.mediator = med;
        this.name = name;
    }

    public abstract void bid(int price);

    public abstract void cancelTheBid();
}

public class AuctionBuyer extends Buyer {

    // implementation of the bidding proccess
    // There is an option to bid and an option to
    // cancel the bidding
    public AuctionBuyer(Mediator mediator,
                                String name)
    {
        super(mediator, name);
    }

    @Override
    public void bid(int price)
```

```
    {
        this.price = price;
    }

    @Override
    public void cancelTheBid()
    {
        this.price = -1;
    }
}

public class Main {

    /* This program illustrate an auction. The AuctionMediator
    is responsible for adding the buyers, and after each
    buyer bid a certain amount for the item, the mediator
    know who won the auction. */
    public static void main(String[] args)
    {

        AuctionMediator med = new AuctionMediator();
        Buyer b1 = new AuctionBuyer(med, "Tal Baum");
        Buyer b2 = new AuctionBuyer(med, "Elad Shamailov");
        Buyer b3 = new AuctionBuyer(med, "John Smith");

        // Crate and add buyers
        med.addBuyer(b1);
        med.addBuyer(b2);
        med.addBuyer(b3);

        System.out.println("Welcome to the auction. Tonight " +
                        "we are selling a vacation to Vegas." +
                        " please Bid your offers.");
        System.out.println("--------------------------------" +
                                        "---------------");
        System.out.println("Waiting for the buyer's offers...");

        // Making bids
        b1.bid(1800);
        b2.bid(2000);
        b3.bid(780);
        System.out.println("---------------------------------" +
                                            "--------------");
        med.findHighestBidder();

        b2.cancelTheBid();
        System.out.print(b2.name + " Has canceled his bid!, " +
                                            "in that case ");
```

```
        med.findHighestBidder();
    }
}
```

Output:

```
Tal Baum was added to the buyers list.
Elad Shamailov was added to the buyers list.
John Smith was added to the buyers list.
Welcome to the auction. Tonight we are
 selling a vacation to Vegas. please Bid your offers.
-----------------------------------------------
Waiting for the buyer's offers...
-----------------------------------------------
The auction winner is Elad Shamailov.
He paid 2000$ for the item.
Elad Shamailov Has canceled his bid!, In that
case The auction winner is Tal Baum.
He paid 1800$ for the item.
```

Advantages:

- Simplicity
- You can replace one object in the structure with a different one without affecting the classes and the interfaces.

Disadvantages:

- The Mediator often needs to be very intimate with all the different classes, And it makes it really complex.
- Can make it difficult to maintain.

Author : http://designpattern.co.il/

Source

https://www.geeksforgeeks.org/mediator-design-pattern-2/

Chapter 39

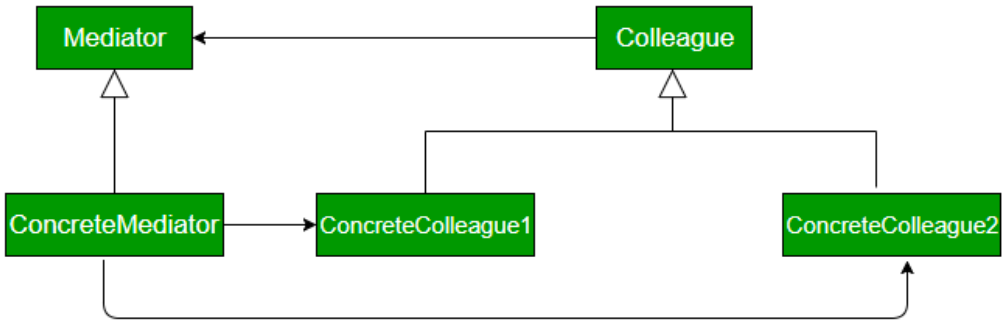
Mediator design pattern

Mediator design pattern - GeeksforGeeks

Mediator design pattern is one of the important and widely used behavioral design pattern. Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on providing a mediator between objects for communication and help in implementing lose-coupling between objects.

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it's own logic to provide way of communication.

UML Diagram Mediator design pattern



Design components

- **Mediator :**It defines the interface for communication between colleague objects.
- **ConcreteMediator :** It implements the mediator interface and coordinates communication between colleague objects.

- **Colleague :** It defines the interface for communication with other colleagues
- **ConcreteColleague :** It implements the colleague interface and communicates with other colleagues through its mediator

Let's see an example of Mediator design pattern.

```
 class ATCMediator implements IATCMediator
{
    private Flight flight;
    private Runway runway;
    public boolean land;

    public void registerRunway(Runway runway)
    {
        this.runway = runway;
    }

    public void registerFlight(Flight flight)
    {
        this.flight = flight;
    }

    public boolean isLandingOk()
    {
        return land;
    }

    @Override
    public void setLandingStatus(boolean status)
    {
        land = status;
    }
}

interface Command
{
    void land();
}

interface IATCMediator
{

    public void registerRunway(Runway runway);

    public void registerFlight(Flight flight);

    public boolean isLandingOk();
```

```
    public void setLandingStatus(boolean status);
}

class Flight implements Command
{
    private IATCMediator atcMediator;

    public Flight(IATCMediator atcMediator)
    {
        this.atcMediator = atcMediator;
    }

    public void land()
    {
        if (atcMediator.isLandingOk())
        {
            System.out.println("Successfully Landed.");
            atcMediator.setLandingStatus(true);
        }
        else
            System.out.println("Waiting for landing.");
    }

    public void getReady()
    {
        System.out.println("Ready for landing.");
    }

}

class Runway implements Command
{
    private IATCMediator atcMediator;

    public Runway(IATCMediator atcMediator)
    {
        this.atcMediator = atcMediator;
        atcMediator.setLandingStatus(true);
    }

    @Override
    public void land()
    {
        System.out.println("Landing permission granted.");
        atcMediator.setLandingStatus(true);
    }

}
```

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

        IATCMediator atcMediator = new ATCMediator();
        Flight sparrow101 = new Flight(atcMediator);
        Runway mainRunway = new Runway(atcMediator);
        atcMediator.registerFlight(sparrow101);
        atcMediator.registerRunway(mainRunway);
        sparrow101.getReady();
        mainRunway.land();
        sparrow101.land();

    }
}
```

Output:

```
Ready for landing.
Landing permission granted.
Successfully Landed.
```

Advantage

- It limits subclassing. A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behaviour requires subclassing Mediator only, Colleague classes can be reused as is.

Disadvantage

- It centralizes control. The mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain

Source

https://www.geeksforgeeks.org/mediator-design-pattern/

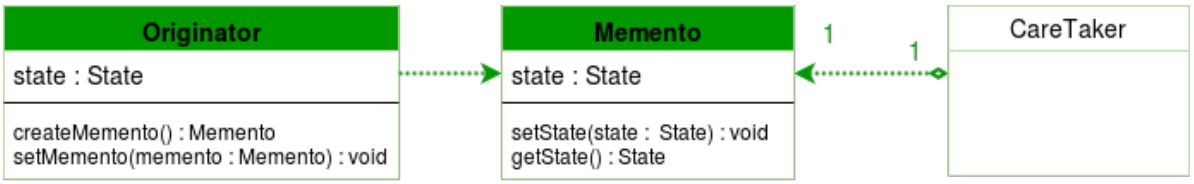
Chapter 40

Memento design pattern

Memento design pattern - GeeksforGeeks

Memento pattern is a behavioral design pattern. Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.

UML Diagram Memento design pattern



Design components

- **originator :** the object for which the state is to be saved. It creates the memento and uses it in future to undo.
- **memento :** the object that is going to maintain the state of originator. Its just a POJO.
- **caretaker :** the object that keeps track of multiple memento. Like maintaining savepoints.

A **Caretaker** would like to perform an operation on the **Originator** while having the possibility to rollback. The caretaker calls the **createMemento()** method on the originator asking the originator to pass it a memento object. At this point the originator creates a memento object saving its internal state and passes the memento to the caretaker. The caretaker maintains the memento object and performs the operation. In case of the need to undo the operation, the caretaker calls the **setMemento()** method on the originator passing the maintained memento object. The originator would accept the memento, using it to restore its previous state.

Let's see an example of Memento design pattern.

```
 import java.util.List;
import java.util.ArrayList;

class Life
{
    private String time;

    public void set(String time)
    {
        System.out.println("Setting time to " + time);
        this.time = time;
    }

    public Memento saveToMemento()
    {
        System.out.println("Saving time to Memento");
        return new Memento(time);
    }

    public void restoreFromMemento(Memento memento)
    {
        time = memento.getSavedTime();
        System.out.println("Time restored from Memento: " + time);
    }

    public static class Memento
    {
        private final String time;

        public Memento(String timeToSave)
        {
            time = timeToSave;
        }

        public String getSavedTime()
        {
            return time;
        }
    }
}

class Design
{

    public static void main(String[] args)
    {

        List<Life.Memento> savedTimes = new ArrayList<Life.Memento>();
```

```
        Life life = new Life();

        //time travel and record the eras
        life.set("1000 B.C.");
        savedTimes.add(life.saveToMemento());
        life.set("1000 A.D.");
        savedTimes.add(life.saveToMemento());
        life.set("2000 A.D.");
        savedTimes.add(life.saveToMemento());
        life.set("4000 A.D.");

        life.restoreFromMemento(savedTimes.get(0));

    }
}
```

Output:

```
Setting time to 1000 B.C.
Saving time to Memento
Setting time to 1000 A.D.
Saving time to Memento
Setting time to 2000 A.D.
Saving time to Memento
Setting time to 4000 A.D.
Time restored from Memento: 1000 B.C.
```

Advantage

- We can use Serialization to achieve memento pattern implementation that is more generic rather than Memento pattern where every object needs to have it's own Memento class implementation.

Disadvantage

- If Originator object is very huge then Memento object size will also be huge and use a lot of memory.

Source

https://www.geeksforgeeks.org/memento-design-pattern/

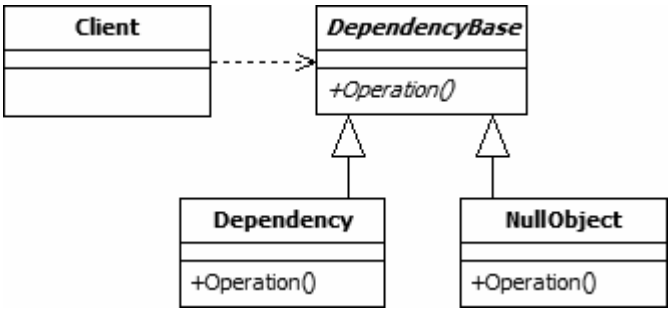
Chapter 41

Null object Design Pattern

Null object Design Pattern - GeeksforGeeks

The Null object pattern is a design pattern that simplifies the use of dependencies that can be undefined. This is achieved by using instances of a concrete class that implements a known interface, instead of null references.
We create an abstract class specifying various operations to be done, concrete classes extending this class and a null object class providing do nothing implemention of this class and will be used seemlessly where we need to check null value.

UML Diagram for Null object Design pattern



Design components

- **Client :** This class has a dependency that may or may not be required. Where no functionality is required in the dependency, it will execute the methods of a null object.
- **DependencyBase :** This abstract class is the base class for the various available dependencies that the Client may use. This is also the base class for the null object class. Where the base class provides no shared functionality, it may be replaced with an interface.
- **Dependency :** This class is a functional dependency that may be used by the Client.
- **NullObject :** This is the null object class that can be used as a dependency by the Client. It contains no functionality but implements all of the members defined by the DependencyBase abstract class.

Let's see an example of Null object design pattern.

```
 // Java program to illustrate Null
// Object Design Pattern

abstract class Emp
{
    protected String name;
    public abstract boolean isNull();
    public abstract String getName();
}

class Coder extends Emp
{
    public Coder(String name)
    {
        this.name = name;
    }
    @Override
    public String getName()
    {
        return name;
    }
    @Override
    public boolean isNull()
    {
        return false;
    }
}

class NoClient extends Emp
{
    @Override
    public String getName()
    {
        return "Not Available";
    }

    @Override
    public boolean isNull()
    {
        return true;
    }
}

class EmpData
{
```

```
    public static final String[] names = {"Lokesh", "Kushagra", "Vikram"};
    public static Emp getClient(String name)
    {
        for (int i = 0; i < names.length; i++)
        {
            if (names[i].equalsIgnoreCase(name))
            {
                return new Coder(name);
            }
        }
        return new NoClient();
    }
}

public class Main
{
    public static void main(String[] args)
    {
        Emp emp1 = EmpData.getClient("Lokesh");
        Emp emp2 = EmpData.getClient("Kushagra");
        Emp emp3 = EmpData.getClient("Vikram");
        Emp emp4 = EmpData.getClient("Rishabh");


        System.out.println(emp1.getName());
        System.out.println(emp2.getName());
        System.out.println(emp3.getName());
        System.out.println(emp4.getName());
    }
}
```

Output:

```
Lokesh
Kushagra
Vikram
Not Available
```

Advantages :

- It defines class hierarchies consisting of real objects and null objects. Null objects can be used in place of real objects when the object is expected to do nothing. Whenever client code expects a real object, it can also take a null object.
- Also makes the client code simple. Clients can treat real collaborators and null collaborators uniformly. Clients normally don't know whether they're dealing with a real or a null collaborator. This simplifies client code, because it avoids having to write testing code which handles the null collaborator specially.

Disadvantages :

- Can be difficult to implement if various clients do not agree on how the null object should do nothing as when your AbstractObject interface is not well defined.
- Can necessitate creating a new NullObject class for every new AbstractObject class.

Source

https://www.geeksforgeeks.org/null-object-design-pattern/

Chapter 42

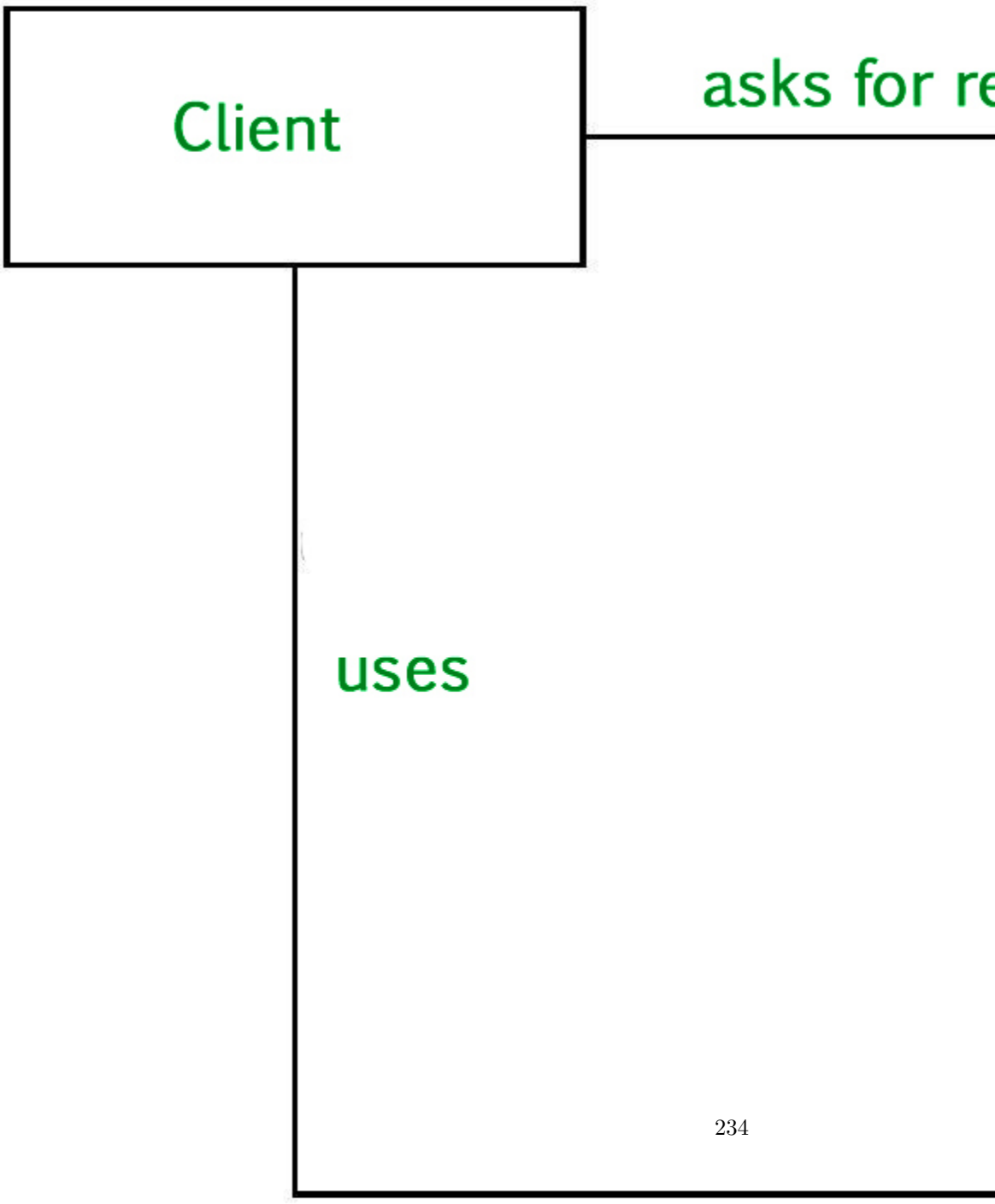
Object Pool Design Pattern

Object Pool Design Pattern - GeeksforGeeks

Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high.
Basically, an Object pool is a container which contains some amount of objects. So, when an object is taken from the pool, it is not available in the pool until it is put back.
Objects in the pool have a lifecycle:

- Creation
- Validation
- Destroy.

UML Diagram Object Pool Design Pattern


Client
asks for re
uses

- **Client :** This is the class that uses an object of the PooledObject type.
- **ReuseablePool:** The PooledObject class is the type that is expensive or slow to instantiate, or that has limited availability, so is to be held in the object pool.
- **ObjectPool :** The Pool class is the most important class in the object pool design pattern. ObjectPool maintains a list of available objects and a collection of objects that have already been requested from the pool.

Let's take the example of the database connections. It's obviously that opening too many connections might affect the performance for several reasons:

- Creating a connection is an expensive operation.
- When there are too many connections opened it takes longer to create a new one and the database server will become overloaded.

Here the object pool manages the connections and provide a way to reuse and share them. It can also limit the maximum number of objects that can be created.

```
 // Jave program to illustrate
// Object Pool Design Pattern
abstract class ObjectPool<T> {
    long deadTime;

    Hashtable<T, Long> lock, unlock;

    ObjectPool()
    {
        deadTime = 50000; // 50 seconds
        lock = new Hashtable<T, Long>();
        unlock = new Hashtable<T, Long>();
    }

    abstract T create();

    abstract boolean validate(T o);

    abstract void dead(T o);

    synchronized T takeOut()
    {
        long now = System.currentTimeMillis();
        T t;
        if (unlock.size() > 0) {
            Enumeration<T> e = unlock.keys();
            while (e.hasMoreElements()) {
                t = e.nextElement();
                if ((now - unlock.get(t)) > deadTime) {
                    // object has deadd
```

```
                    unlock.remove(t);
                    dead(t);
                    t = null;
                }
                else {
                    if (validate(t)) {
                        unlock.remove(t);
                        lock.put(t, now);
                        return (t);
                    }
                    else {
                        // object failed validation
                        unlock.remove(t);
                        dead(t);
                        t = null;
                    }
                }
            }
        }
        // no objects available, create a new one
        t = create();
        lock.put(t, now);
        return (t);
    }
    synchronized void takeIn(T t)
    {
        lock.remove(t);
        unlock.put(t, System.currentTimeMillis());
    }
}

// Three methods are abstract
// and therefore must be implemented by the subclass

class JDBCConnectionPool extends ObjectPool<Connection> {
    String dsn, usr, pwd;

    JDBCConnectionPool(String driver, String dsn, String usr, String pwd)
    {
        super();
        try {
            Class.forName(driver).newInstance();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        this.dsn = dsn;
        this.usr = usr;
```

```
        this.pwd = pwd;
    }

    Connection create()
    {
        try {
            return (DriverManager.getConnection(dsn, usr, pwd));
        }
        catch (SQLException e) {
            e.printStackTrace();
            return (null);
        }
    }

    void dead(Connection o)
    {
        try {
            ((Connection)o).close();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    boolean validate(Connection o)
    {
        try {
            return (!((Connection)o).isClosed());
        }
        catch (SQLException e) {
            e.printStackTrace();
            return (false);
        }
    }
}

class Main {
    public static void main(String args[])
    {
        // Create the ConnectionPool:
        JDBCConnectionPool pool = new JDBCConnectionPool(
            "org.hsqldb.jdbcDriver", "jdbc:hsqldb: //localhost/mydb",
            "sa", "password");

        // Get a connection:
        Connection con = pool.takeOut();
        // Return the connection:
        pool.takeIn(con);
```

```
    }
}
```

Advantages

- It offer a significant performance boost.
- It manages the connections and provides a way to reuse and share them.
- Object pool pattern is used when the rate of initializing a instance of the class is high.

When to use Object Pool Design Pattern

- When we have a work to allocates or deallocates many objects
- Also, when we know that we have a limited number of objects that will be in memory at the same time.

Reference :

https://sourcemaking.com/design_patterns/object_pool

Source

https://www.geeksforgeeks.org/object-pool-design-pattern/

Chapter 43

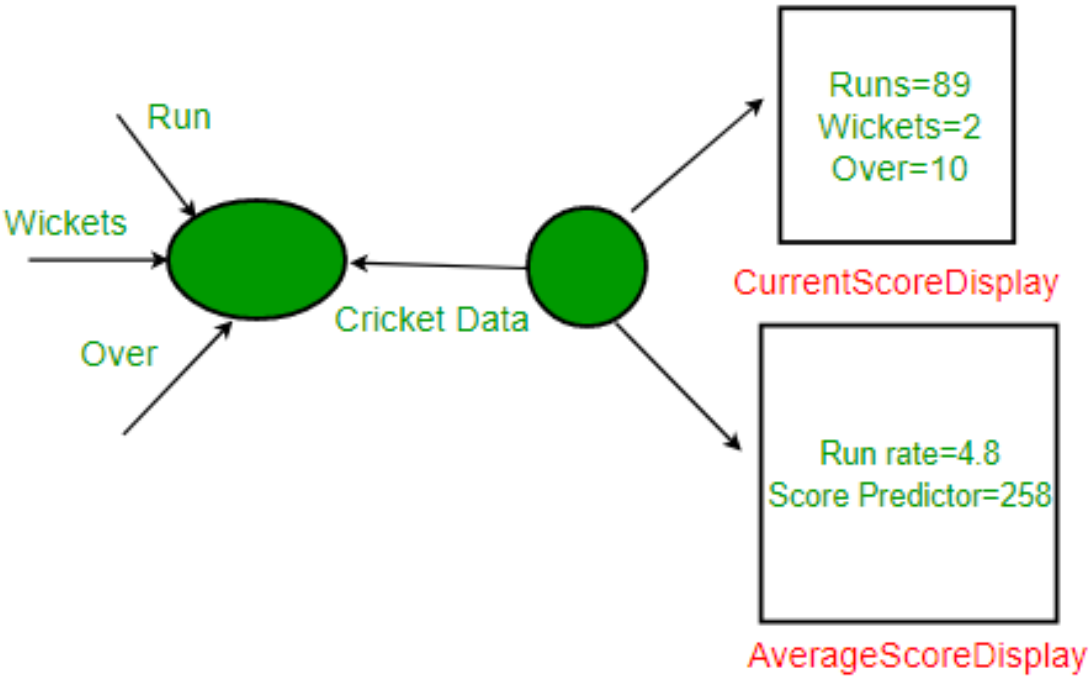
Observer Pattern | Set 1 (Introduction)

Observer Pattern | Set 1 (Introduction) - GeeksforGeeks

Let us first consider the following scenario to understand observer pattern.

Scenario:

Suppose we are building a cricket app that notifies viewers about the information such as current score, run rate etc. Suppose we have made two display elements CurrentScoreDisplay and AverageScoreDisplay. CricketData has all the data (runs, bowls etc.) and whenever data changes the display elements are notified with new data and they display the latest data ac-



cordingly.

Below is the java implementation of this design.

```
 // Java implementation of above design for Cricket App. The
// problems with this design are discussed below.

// A class that gets information from stadium and notifies
// display elements, CurrentScoreDisplay & AverageScoreDisplay
class CricketData
{
    int runs, wickets;
    float overs;
    CurrentScoreDisplay currentScoreDisplay;
    AverageScoreDisplay averageScoreDisplay;

    // Constructor
    public CricketData(CurrentScoreDisplay currentScoreDisplay,
                       AverageScoreDisplay averageScoreDisplay)
    {
        this.currentScoreDisplay = currentScoreDisplay;
        this.averageScoreDisplay = averageScoreDisplay;
    }

    // Get latest runs from stadium
```

```
    private int getLatestRuns()
    {
        // return 90 for simplicity
        return 90;
    }

    // Get latest wickets from stadium
    private int getLatestWickets()
    {
        // return 2 for simplicity
        return 2;
    }

    // Get latest overs from stadium
    private float getLatestOvers()
    {
        // return 10.2 for simplicity
        return (float)10.2;
    }

    // This method is used update displays when data changes
    public void dataChanged()
    {
        // get latest data
        runs = getLatestRuns();
        wickets = getLatestWickets();
        overs = getLatestOvers();

        currentScoreDisplay.update(runs,wickets,overs);
        averageScoreDisplay.update(runs,wickets,overs);
    }
}

// A class to display average score. Data of this class is
// updated by CricketData
class AverageScoreDisplay
{
    private float runRate;
    private int predictedScore;

    public void update(int runs, int wickets, float overs)
    {
        this.runRate = (float)runs/overs;
        this.predictedScore = (int) (this.runRate * 50);
        display();
    }

    public void display()
```

```
    {
        System.out.println("\nAverage Score Display:\n" +
                           "Run Rate: " + runRate +
                           "\nPredictedScore: " + predictedScore);
    }
}

// A class to display score. Data of this class is
// updated by CricketData
class CurrentScoreDisplay
{
    private int runs, wickets;
    private float overs;

    public void update(int runs,int wickets,float overs)
    {
        this.runs = runs;
        this.wickets = wickets;
        this.overs = overs;
        display();
    }

    public void display()
    {
        System.out.println("\nCurrent Score Display: \n" +
                           "Runs: " + runs +"\nWickets:"
                           + wickets + "\nOvers: " + overs );
    }
}

// Driver class
class Main
{
    public static void main(String args[])
    {
        // Create objects for testing
        AverageScoreDisplay averageScoreDisplay =
                                        new AverageScoreDisplay();
        CurrentScoreDisplay currentScoreDisplay =
                                        new CurrentScoreDisplay();

        // Pass the displays to Cricket data
        CricketData cricketData = new CricketData(currentScoreDisplay,
                                                  averageScoreDisplay);

        // In real app you would have some logic to call this
        // function when data changes
        cricketData.dataChanged();
```

```
    }
}
```

Output:

```
Current Score Display:
Runs: 90
Wickets:2
Overs: 10.2

Average Score Display:
Run Rate: 8.823529
PredictedScore: 441
```

Problems with above design:

- CricketData holds references to concrete display element objects even though it needs to call only the update method of these objects. It has access to too much additional information it than it requires.
- This statement "currentScoreDisplay.update(runs,wickets,overs);" violates one of the most important design principle "Program to interfaces, not implementations." as we are using concrete objects to share data rather than abstract interfaces.
- CricketData and display elements are tightly coupled.
- If in future another requirement comes in and we need another display element to be added we need to make changes to the non-varying part of our code(CricketData). This is definitely not a good design practice and application might not be able to handle changes and not easy to maintain.

How to avoid these problems?
Use Observer Pattern

Observer pattern

To understand observer pattern, first you need to understand the subject and observer objects.

The relation between subject and observer can easily be understood as an analogy to magazine subscription.

- A magazine publisher(subject) is in the business and publishes magazines (data).
- If you(user of data/observer) are interested in the magazine you subscribe(register), and if a new edition is published it gets delivered to you.
- If you unsubscribe(unregister) you stop getting new editions.
- Publisher doesn't know who you are and how you use the magazine, it just delivers it to you because you are a subscriber(loose coupling).

Definition:

The Observer Pattern defines a one to many dependency between objects so that one object changes state, all of its dependents are notified and updated automatically.

Explanation:

- One to many dependency is between Subject(One) and Observer(Many).
- There is dependency as Observers themselves don't have access to data. They are dependent on Subject to provide them data.

Class diagram:

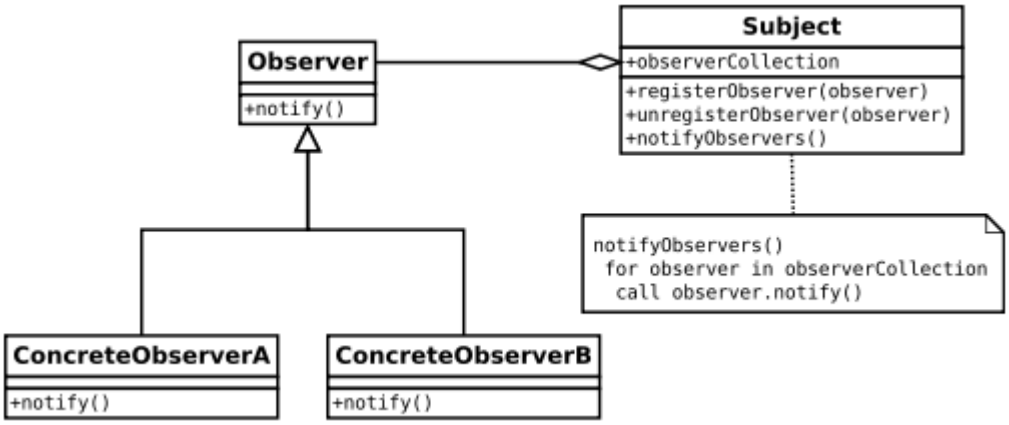


Image Source : Wikipedia

- Here Observer and Subject are interfaces(can be any abstract super type not necessarily java interface).
- All observers who need the data need to implement observer interface.
- notify() method in observer interface defines the action to be taken when the subject provides it data.
- The subject maintains an observerCollection which is simply the list of currently registered(subscribed) observers.
- registerObserver(observer) and unregisterObserver(observer) are methods to add and remove observers respectively.
- notifyObservers() is called when the data is changed and the observers need to be supplied with new data.

Advantages:
Provides a loosely coupled design between objects that interact. Loosely coupled objects are flexible with changing requirements. Here loose coupling means that the interacting objects should have less information about each other.

Observer pattern provides this loose coupling as:

- Subject only knows that observer implement Observer interface.Nothing more.
- There is no need to modify Subject to add or remove observers.
- We can reuse subject and observer classes independently of each other.

Disadvantages:

- Memory leaks caused by Lapsed listener problem because of explicit register and unregistering of observers.

When to use this pattern?
You should consider using this pattern in your application when multiple objects are dependent on the state of one object as it provides a neat and well tested design for the same.

Real Life Uses:

- It is heavily used in GUI toolkits and event listener. In java the button(subject) and onClickListener(observer) are modelled with observer pattern.
- Social media, RSS feeds, email subscription in which you have the option to follow or subscribe and you receive latest notification.
- All users of an app on play store gets notified if there is an update.

Observer Pattern | Set 2 (Implementation)

Source

https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/

Chapter 44

Observer Pattern | Set 2 (Implementation)

Observer Pattern | Set 2 (Implementation) - GeeksforGeeks

We strongly recommend to refer below Set 1 before moving on to this post.

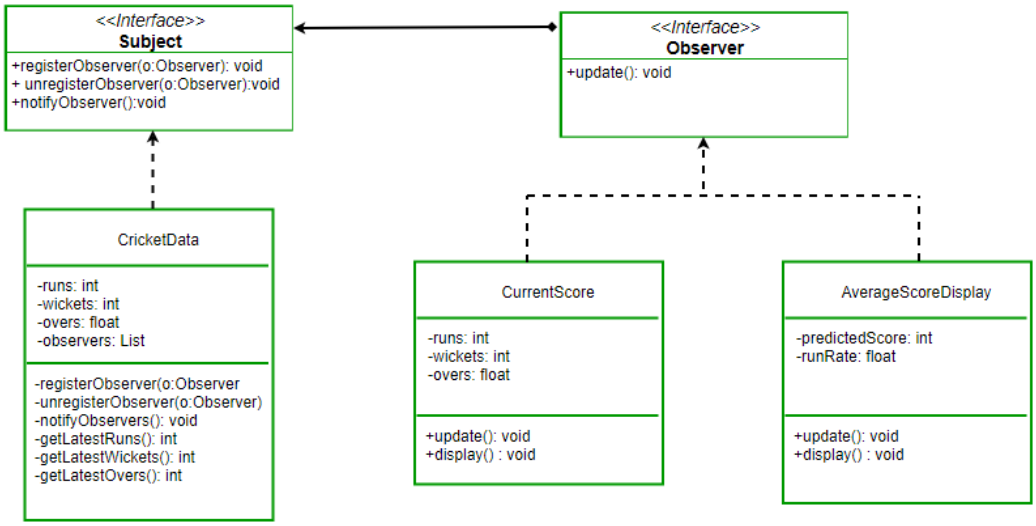
Observer Pattern -Introduction

In Set 1, we discussed below problem, a solution for the problem without Observer pattern and problems with the solution.

Suppose we are building a cricket app that notifies viewers about the information such as current score, run rate etc. Suppose we have made two display elements CurrentScoreDisplay and AverageScoreDisplay. CricketData has all the data (runs, bowls etc.) and whenever data changes the display elements are notified with new data and they display the latest data accordingly

Applying Observer pattern to above problem:
Let us see how we can improve the design of our application using observer pattern. If we observe the flow of data we can easily see that the CricketData and display elements follow subject-observers relationship.

New Class Diagram:



Java Implementation:

```
 // Java program to demonstrate working of
// onserver pattern
import java.util.ArrayList;
import java.util.Iterator;

// Implemented by Cricket data to communicate
// with observers
interface Subject
{
    public void registerObserver(Observer o);
    public void unregisterObserver(Observer o);
    public void notifyObservers();
}

class CricketData implements Subject
{
    int runs;
    int wickets;
    float overs;
    ArrayList<Observer> observerList;

    public CricketData() {
        observerList = new ArrayList<Observer>();
    }

    @Override
    public void registerObserver(Observer o) {
        observerList.add(o);
```

```
}

@Override
public void unregisterObserver(Observer o) {
    observerList.remove(observerList.indexOf(o));
}

@Override
public void notifyObservers()
{
    for (Iterator<Observer> it =
          observerList.iterator(); it.hasNext();)
    {
        Observer o = it.next();
        o.update(runs,wickets,overs);
    }
}

// get latest runs from stadium
private int getLatestRuns()
{
    // return 90 for simplicity
    return 90;
}

// get latest wickets from stadium
private int getLatestWickets()
{
    // return 2 for simplicity
    return 2;
}

// get latest overs from stadium
private float getLatestOvers()
{
    // return 90 for simplicity
    return (float)10.2;
}

// This method is used update displays
// when data changes
public void dataChanged()
{
    //get latest data
    runs = getLatestRuns();
    wickets = getLatestWickets();
    overs = getLatestOvers();
```

```
        notifyObservers();
    }
}

// This interface is implemented by all those
// classes that are to be updated whenever there
// is an update from CricketData
interface Observer
{
    public void update(int runs, int wickets,
                       float overs);
}

class AverageScoreDisplay implements Observer
{
    private float runRate;
    private int predictedScore;

    public void update(int runs, int wickets,
                        float overs)
    {
        this.runRate =(float)runs/overs;
        this.predictedScore = (int)(this.runRate * 50);
        display();
    }

    public void display()
    {
        System.out.println("\nAverage Score Display: \n"
                           + "Run Rate: " + runRate +
                           "\nPredictedScore: " +
                           predictedScore);
    }
}

class CurrentScoreDisplay implements Observer
{
    private int runs, wickets;
    private float overs;

    public void update(int runs, int wickets,
                        float overs)
    {
        this.runs = runs;
        this.wickets = wickets;
        this.overs = overs;
        display();
    }
```

```
    public void display()
    {
        System.out.println("\nCurrent Score Display:\n"
                           + "Runs: " + runs +
                           "\nWickets:" + wickets +
                           "\nOvers: " + overs );
    }
}

// Driver Class
class Main
{
    public static void main(String args[])
    {
        // create objects for testing
        AverageScoreDisplay averageScoreDisplay =
                           new AverageScoreDisplay();
        CurrentScoreDisplay currentScoreDisplay =
                           new CurrentScoreDisplay();

        // pass the displays to Cricket data
        CricketData cricketData = new CricketData();

        // register display elements
        cricketData.registerObserver(averageScoreDisplay);
        cricketData.registerObserver(currentScoreDisplay);

        // in real app you would have some logic to
        // call this function when data changes
        cricketData.dataChanged();

        //remove an observer
        cricketData.unregisterObserver(averageScoreDisplay);

        // now only currentScoreDisplay gets the
        // notification
        cricketData.dataChanged();
    }
}
```

Output:

```
Average Score Display:
Run Rate: 8.823529
PredictedScore: 441

Current Score Display:
```

```
Runs: 90
Wickets:2
Overs: 10.2

Current Score Display:
Runs: 90
Wickets:2
Overs: 10.2
```

Note: Now we can add/delete as many observers without changing the subject.

References:

1. https://en.wikipedia.org/wiki/Observer_pattern
2. Head First Design Patterns book (highly recommended)

Source

https://www.geeksforgeeks.org/observer-pattern-set-2-implementation/

Chapter 45

Prototype Design Pattern

Prototype Design Pattern - GeeksforGeeks

Prototype allows us to hide the complexity of making new instances from the client. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required. This approach saves costly resources and time, especially when the object creation is a heavy process.

The prototype pattern is a creational design pattern. Prototype patterns is required, when object creation is time consuming, and costly operation, so we create object with existing object itself. One of the best available way to create object from existing objects are **clone() method**. Clone is the simplest approach to implement prototype pattern. However, it is your call to decide how to copy existing object based on your business model.

Prototype Design Participants

1) **Prototype** : This is the prototype of actual object.

2) **Prototype registry** : This is used as registry service to have all prototypes accessible using simple string parameters.

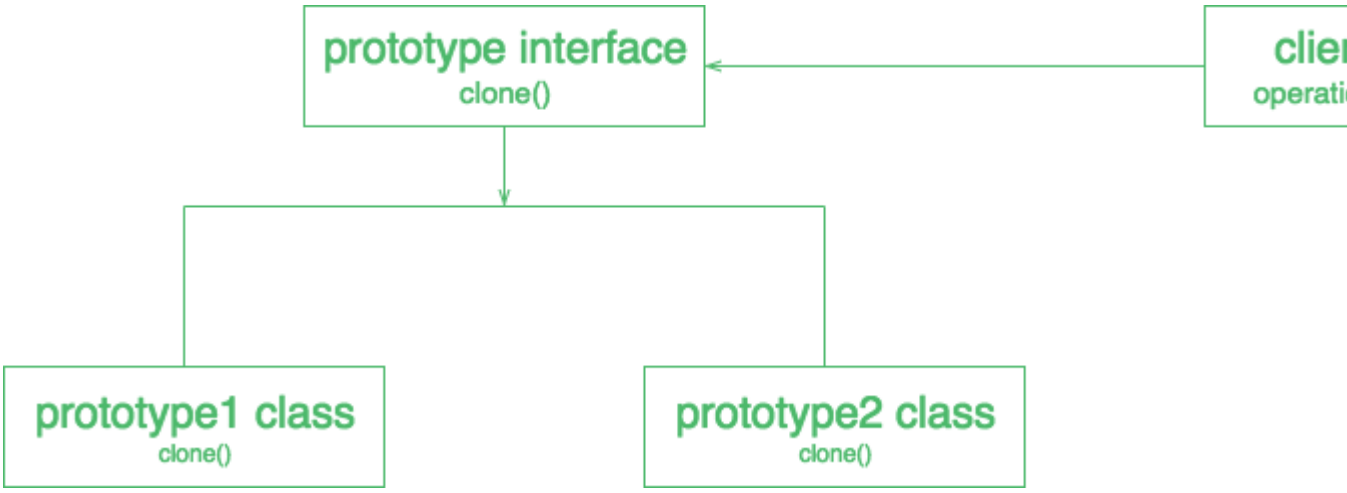
3) **Client** : Client will be responsible for using registry service to access prototype instances.

When to use the Prototype Design Pattern

When a system should be independent of how its products are created, composed, and represented and
When the classes to instantiate are specified at run-time.
For example,
1) By dynamic loading or To avoid building a class hierarchy of factories that parallels the class hierarchy of products or

2) When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

The UML Diagram of the Prototype Design Pattern




```
 // A Java program to demonstrate working of
// Prototype Design Pattern with example
// of a ColorStore class to store existing objects.

import java.util.HashMap;
import java.util.Map;


abstract class Color implements Cloneable
{

    protected String colorName;

    abstract void addColor();

    public Object clone()
    {
        Object clone = null;
        try
        {
            clone = super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
        return clone;
    }
}
```

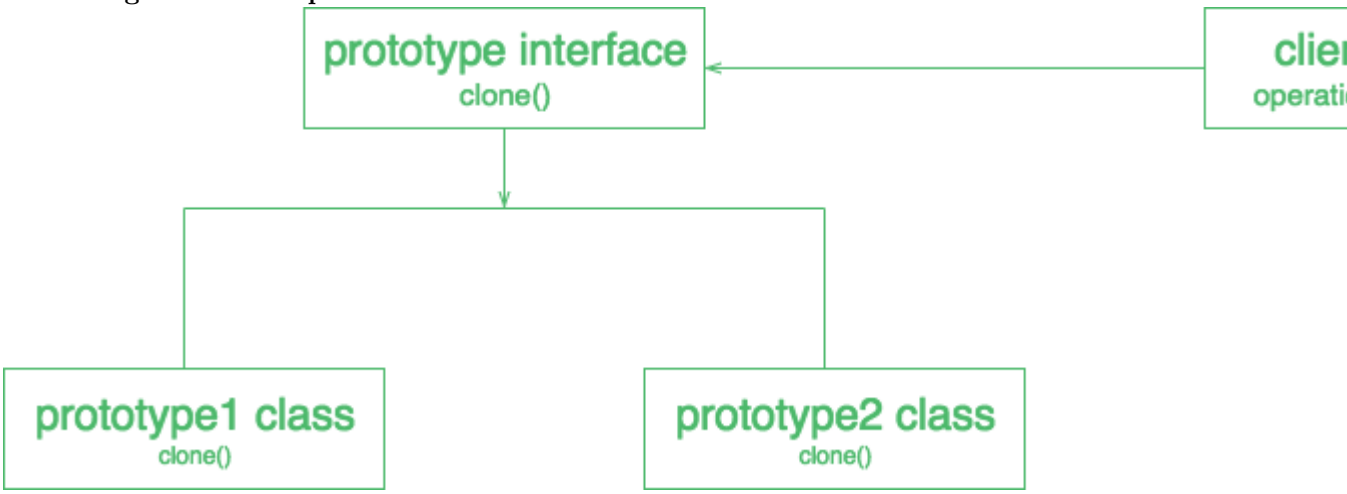
```
class blueColor extends Color
{
    public blueColor()
    {
        this.colorName = "blue";
    }

    @Override
    void addColor()
    {
        System.out.println("Blue color added");
    }

}

class blackColor extends Color{

    public blackColor()
    {
        this.colorName = "black";
    }

    @Override
    void addColor()
    {
        System.out.println("Black color added");
    }
}

class ColorStore {

    private static Map<String, Color> colorMap = new HashMap<String, Color>();

    static
    {
        colorMap.put("blue", new blueColor());
        colorMap.put("black", new blackColor());
    }

    public static Color getColor(String colorName)
    {
        return (Color) colorMap.get(colorName).clone();
    }
}


// Driver class
class Prototype
```

```
{
    public static void main (String[] args)
    {
        ColorStore.getColor("blue").addColor();
        ColorStore.getColor("black").addColor();
        ColorStore.getColor("black").addColor();
        ColorStore.getColor("blue").addColor();
    }
}
```

Output :

```
Blue color added
Black color added
Black color added
Blue color added
```

UML diagram of example:



Advantages of Prototype Design Pattern

- **Adding and removing products at run-time** – Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.
- **Specifying new objects by varying values** – Highly dynamic systems let you define new behavior through object composition by specifying values for an object's variables and not by defining new classes.
- **Specifying new objects by varying structure** – Many applications build objects from parts and subparts. For convenience, such applications often let you instantiate complex, user-defined structures to use a specific subcircuit again and again.

- **Reduced subclassing** – Factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all.

Disadvantages of Prototype Design Pattern

- Overkill for a project that uses very few objects and/or does not have an underlying emphasis on the extension of prototype chains.
- It also hides concrete product classes from the client
- Each subclass of Prototype must implement the clone() operation which may be difficult, when the classes under consideration already exist. Also implementing clone() can be difficult when their internals include objects that don't support copying or have circular references.

Source

https://www.geeksforgeeks.org/prototype-design-pattern/

Chapter 46

Proxy Design Pattern

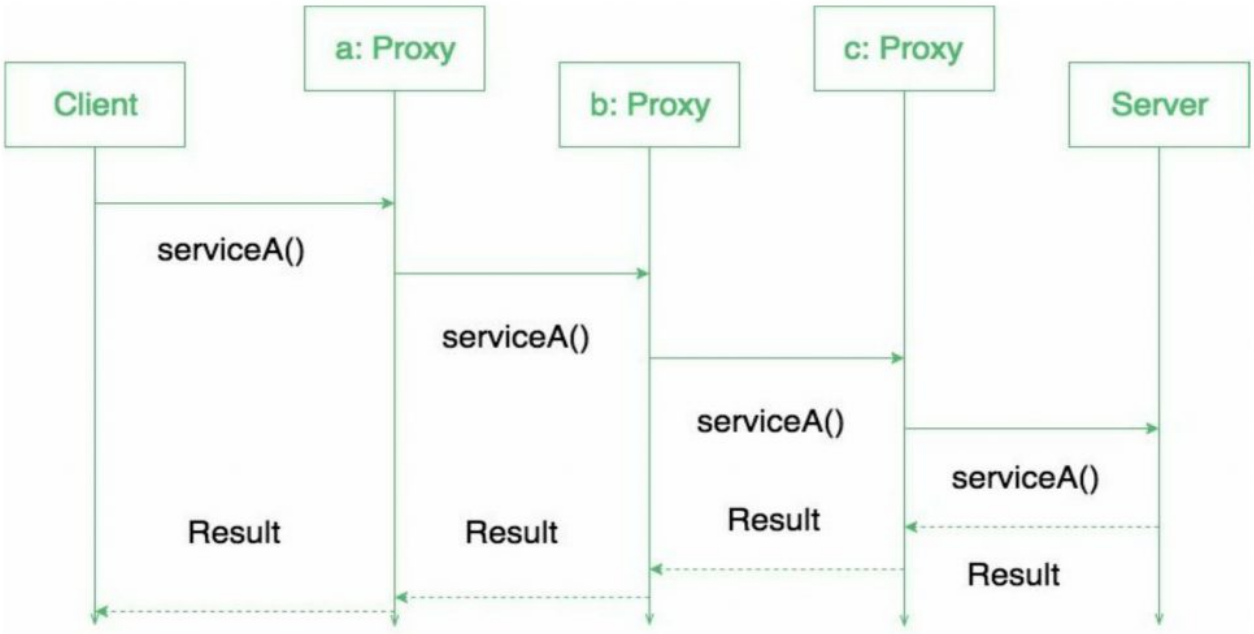
Proxy Design Pattern - GeeksforGeeks

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. They are closely related in structure, but not purpose, to Adapters and Decorators.

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**".

Behavior

As in the decorator pattern, proxies can be chained together. The client, and each proxy, believes it is delegating messages to the real server:



When to use this pattern?

Proxy pattern is used when we need to create a wrapper to cover the main object's complexity from the client.

Types of proxies

Remote proxy:
They are responsible for representing the object located remotely. Talking to the real object might involve marshalling and unmarshalling of data and talking to the remote object. All that logic is encapsulated in these proxies and the client application need not worry about them.

Virtual proxy:

These proxies will provide some default and instant results if the real object is supposed to take some time to produce results. These proxies initiate the operation on real objects and provide a default result to the application. Once the real object is done, these proxies push the actual data to the client where it has provided dummy data earlier.

Protection proxy:

If an application does not have access to some resource then such proxies will talk to the objects in applications that have access to that resource and then get the result back.

Smart Proxy:

A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. An example can be to check if the real object is locked before it is accessed to ensure that no other object can change it.

Some Examples

A very simple real life scenario is our college internet, which restricts few site access. The proxy first checks the host you are connecting to, if it is not part of restricted site list, then it connects to the real internet. This example is based on Protection proxies.

Lets see how it works :

Interface of Internet

```
 package com.saket.demo.proxy;

public interface Internet
{
    public void connectTo(String serverhost) throws Exception;
}
```

RealInternet.java

```
 package com.saket.demo.proxy;

public class RealInternet implements Internet
{
    @Override
    public void connectTo(String serverhost)
    {
        System.out.println("Connecting to "+ serverhost);
    }
}
```

ProxyInternet.java

```
 package com.saket.demo.proxy;

import java.util.ArrayList;
import java.util.List;


public class ProxyInternet implements Internet
{
    private Internet internet = new RealInternet();
    private static List<String> bannedSites;

    static
    {
```

```
        bannedSites = new ArrayList<String>();
        bannedSites.add("abc.com");
        bannedSites.add("def.com");
        bannedSites.add("ijk.com");
        bannedSites.add("lnm.com");
    }

    @Override
    public void connectTo(String serverhost) throws Exception
    {
        if(bannedSites.contains(serverhost.toLowerCase()))
        {
            throw new Exception("Access Denied");
        }

        internet.connectTo(serverhost);
    }

}
```

Client.java

```
 package com.saket.demo.proxy;

public class Client
{
    public static void main (String[] args)
    {
        Internet internet = new ProxyInternet();
        try
        {
            internet.connectTo("geeksforgeeks.org");
            internet.connectTo("abc.com");
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

As one of the site is mentioned in the banned sites, So
Running the program will give the output :

```
Connecting to geeksforgeeks.org
Access Denied
```

Benefits:

- One of the advantages of Proxy pattern is security.
- This pattern avoids duplication of objects which might be huge size and memory intensive. This in turn increases the performance of the application.
- The remote proxy also ensures about security by installing the local code proxy (stub) in the client machine and then accessing the server with help of the remote code.

Drawbacks/Consequences:

This pattern introduces another layer of abstraction which sometimes may be an issue if the RealSubject code is accessed by some of the clients directly and some of them might access the Proxy classes. This might cause disparate behaviour.

Interesting points:

- There are few differences between the related patterns. Like Adapter pattern gives a different interface to its subject, while Proxy patterns provides the same interface from the original object but the decorator provides an enhanced interface. Decorator pattern adds additional behaviour at runtime.
- Proxy used in Java API: java.rmi.*;

Source

https://www.geeksforgeeks.org/proxy-design-pattern/

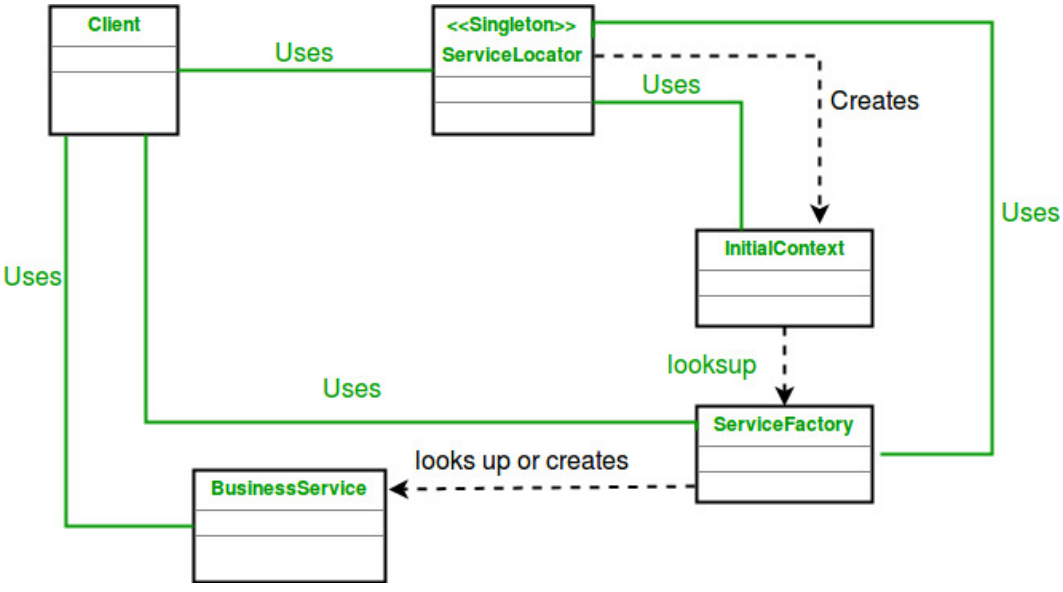
Chapter 47

Service Locator Pattern

Service Locator Pattern - GeeksforGeeks

The service locator pattern is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the "service locator" which on request returns the information necessary to perform a certain task.
The ServiceLocator is responsible for returning instances of services when they are requested for by the service consumers or the service clients.

UML Diagram Service Locator Pattern



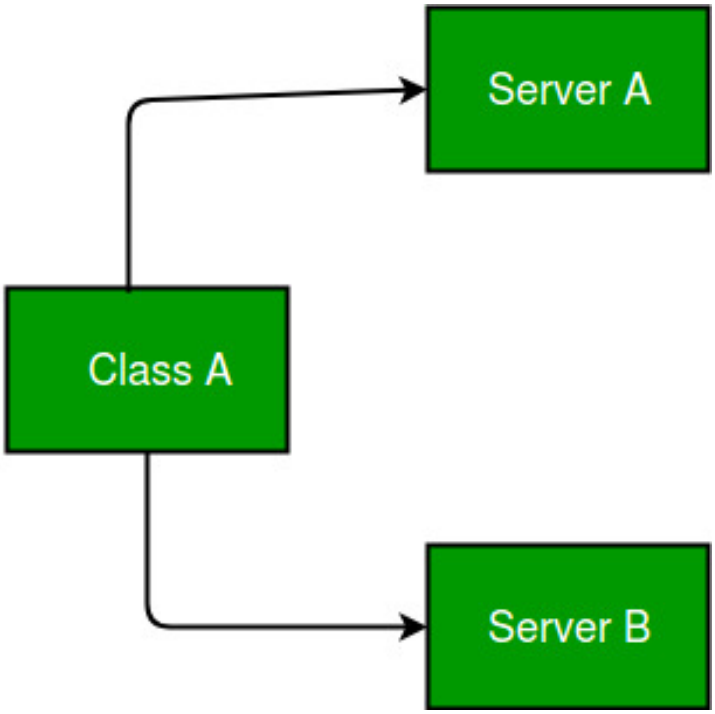
Design components

- **Service Locator :** The Service Locator abstracts the API lookup services, vendor dependencies, lookup complexities, and business object creation, and provides a simple

interface to clients. This reduces the client's complexity. In addition, the same client or other clients can reuse the Service Locator.

- **InitialContext :** The InitialContext object is the start point in the lookup and creation process. Service providers provide the context object, which varies depending on the type of business object provided by the Service Locator's lookup and creation service.
- **ServiceFactory :** The ServiceFactory object represents an object that provides life cycle management for the BusinessService objects. The ServiceFactory object for enterprise beans is an EJBHome object.
- **BusinessService :** The BusinessService is a role that is fulfilled by the service the client is seeking to access. The BusinessService object is created or looked up or removed by the ServiceFactory. The BusinessService object in the context of an EJB application is an enterprise bean.

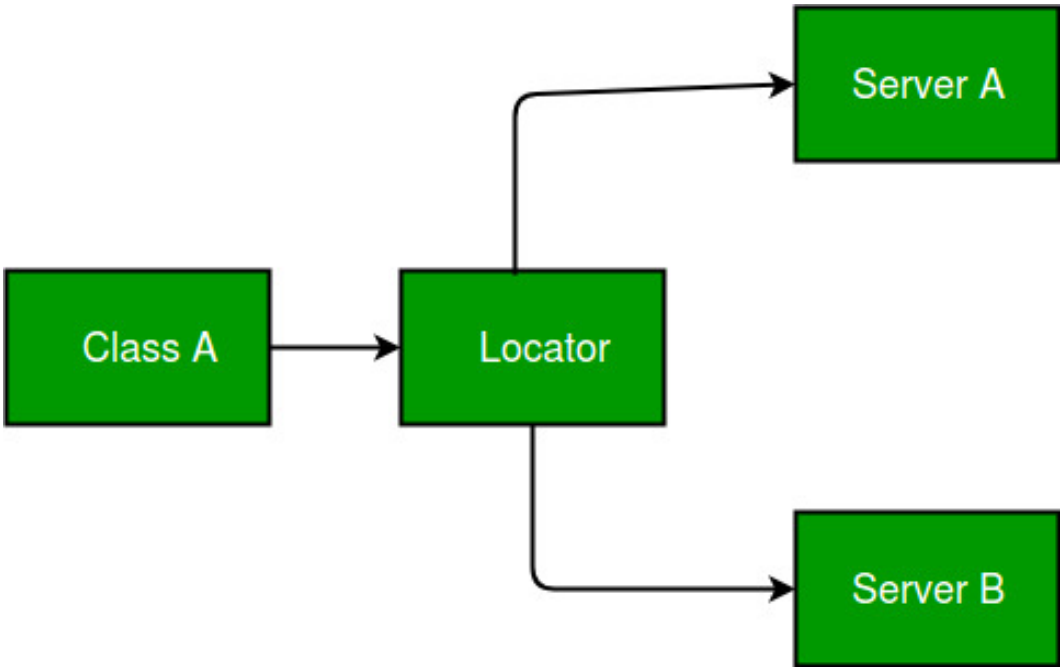
Suppose classes with dependencies on services whose concrete types are specified at compile time.



In the above diagram, ClassA has compile time dependencies on ServiceA and ServiceB.But this situation has drawbacks.

- If we want to replace or update the dependencies we must change the classes source code and recompile the solution.
- The concrete implementation of the dependencies must be available at compile time.

By using the Service Locator pattern :



In simple words, Service Locator pattern does not describe how to instantiate the services. It describes a way to register services and locate them.

Let's see an example of Service Locator Pattern.

```
 // Java program to
// illustrate Service Design Service
// Locator Pattern

import java.util.ArrayList;
import java.util.List;

// Service interface
// for getting name and
// Executing it.

interface Service {
    public String getName();
    public void execute();
}

// Service one implementing Locator
class ServiceOne implements Service {
    public void execute()
    {
        System.out.println("Executing ServiceOne");
    }
```

```
    @Override
    public String getName()
    {
        return "ServiceOne";
    }
}

// Service two implementing Locator
class ServiceTwo implements Service {
    public void execute()
    {
        System.out.println("Executing ServiceTwo");
    }

    @Override
    public String getName()
    {
        return "ServiceTwo";
    }
}

// Checking the context
// for ServiceOne and ServiceTwo
class InitialContext {
    public Object lookup(String name)
    {
        if (name.equalsIgnoreCase("ServiceOne")) {
            System.out.println("Creating a new ServiceOne object");
            return new ServiceOne();
        }
        else if (name.equalsIgnoreCase("ServiceTwo")) {
            System.out.println("Creating a new ServiceTwo object");
            return new ServiceTwo();
        }
        return null;
    }
}

class Cache {
    private List<Service> services;

    public Cache()
    {
        services = new ArrayList<Service>();
    }

    public Service getService(String serviceName)
```

```
    {
        for (Service service : services) {
            if (service.getName().equalsIgnoreCase(serviceName)) {
                System.out.println("Returning cached "
                                   + serviceName + " object");
                return service;
            }
        }
        return null;
    }

    public void addService(Service newService)
    {
        boolean exists = false;
        for (Service service : services) {
            if (service.getName().equalsIgnoreCase(newService.getName())) {
                exists = true;
            }
        }
        if (!exists) {
            services.add(newService);
        }
    }
}

// Locator class
class ServiceLocator {
    private static Cache cache;

    static
    {
        cache = new Cache();
    }

    public static Service getService(String name)
    {
        Service service = cache.getService(name);

        if (service != null) {
            return service;
        }

        InitialContext context = new InitialContext();
        Service ServiceOne = (Service)context.lookup(name);
        cache.addService(ServiceOne);
        return ServiceOne;
    }
}
```

```
// Driver class
class ServiceLocatorPatternDemo {
    public static void main(String[] args)
    {
        Service service = ServiceLocator.getService("ServiceOne");
        service.execute();

        service = ServiceLocator.getService("ServiceTwo");
        service.execute();

        service = ServiceLocator.getService("ServiceOne");
        service.execute();

        service = ServiceLocator.getService("ServiceTwo");
        service.execute();
    }
}
```

Output:

```
Creating a new ServiceOne object
Executing ServiceOne
Creating a new ServiceTwo object
Executing ServiceTwo
Returning cached ServiceOne object
Executing ServiceOne
Returning cached ServiceTwo object
Executing ServiceTwo
```

Advantages :

- Applications can optimize themselves at run-time by selectively adding and removing items from the service locator.
- Large sections of a library or application can be completely separated. The only link between them becomes the registry.

Disadvantages :

- The registry makes the code more difficult to maintain (opposed to using Dependency injection), because it becomes unclear when you would be introducing a breaking change.
- The registry hides the class dependencies causing run-time errors instead of compile-time errors when dependencies are missing.

Strategies

The following strategies are used to implement service Locator Pattern :

EJB Service Locator Strategy : This strategy uses EJBHome object for enterprise bean components and this EJBHome is cached in the ServiceLocator for future use when the client needs the home object again.

JMS Queue Service Locator Strategy : This strategy is applicable to point to point messaging requirements. The following the strategies under JMS Queue Service Locator Strategy.

- JMS Queue Service Locator Strategy
- JMS Topic Service Locator Strategy

Type Checked Service Locator Strategy : This strategy has trade-offs. It reduces the flexibility of lookup, which is in the Services Property Locator strategy, but add the type checking of passing in a constant to the ServiceLocator.getHome() method.

Source

https://www.geeksforgeeks.org/service-locator-pattern/

Chapter 48

Singleton Class in Java

Singleton Class in Java - GeeksforGeeks

In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.
After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.
To design a singleton class:

1. Make constructor as private.
2. Write a static method that has return type object of this singleton class. Here, the concept of Lazy initialization in used to write this static method.

Normal class vs Singleton class: Difference in normal and singleton class in terms of instantiation is that, For normal class we use constructor, whereas for singleton class we use getInstance() method (Example code:I). In general, to avoid confusion we may also use the class name as method name while defining this method (Example code:II).

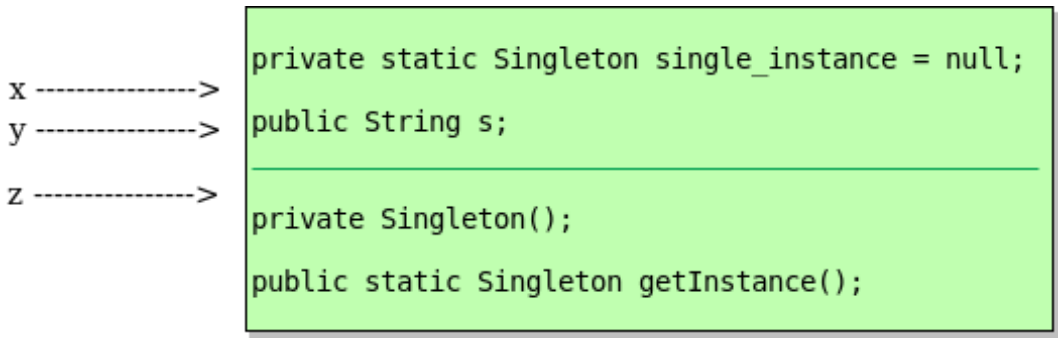
Implementing Singleton class with getInstance() method

```
 // Java program implementing Singleton class
// with getInstance() method
class Singleton
{
    // static variable single_instance of type Singleton
    private static Singleton single_instance = null;

    // variable of type String
    public String s;

    // private constructor restricted to this class itself
```

```
    private Singleton()
    {
        s = "Hello I am a string part of Singleton class";
    }

    // static method to create instance of Singleton class
    public static Singleton getInstance()
    {
        if (single_instance == null)
            single_instance = new Singleton();

        return single_instance;
    }
}

// Driver Class
class Main
{
    public static void main(String args[])
    {
        // instantiating Singleton class with variable x
        Singleton x = Singleton.getInstance();

        // instantiating Singleton class with variable y
        Singleton y = Singleton.getInstance();

        // instantiating Singleton class with variable z
        Singleton z = Singleton.getInstance();

        // changing variable of instance x
        x.s = (x.s).toUpperCase();

        System.out.println("String from x is " + x.s);
        System.out.println("String from y is " + y.s);
        System.out.println("String from z is " + z.s);
        System.out.println("\n");

        // changing variable of instance z
        z.s = (z.s).toLowerCase();

        System.out.println("String from x is " + x.s);
        System.out.println("String from y is " + y.s);
        System.out.println("String from z is " + z.s);
    }
}
```

Output:

```
String from x is HELLO I AM A STRING PART OF SINGLETON CLASS
String from y is HELLO I AM A STRING PART OF SINGLETON CLASS
String from z is HELLO I AM A STRING PART OF SINGLETON CLASS


String from x is hello i am a string part of singleton class
String from y is hello i am a string part of singleton class
String from z is hello i am a string part of singleton class
```



Explanation: In the Singleton class, when we first time call getInstance() method, it creates an object of the class with name single_instance and return it to the variable. Since single_instance is static, it is changed from null to some object. Next time, if we try to call getInstance() method, since single_instance is not null, it is returned to the variable, instead of instantiating the Singleton class again. This part is done by if condition.

In the main class, we instantiate the singleton class with 3 objects x, y, z by calling static method getInstance(). But actually after creation of object x, variables y and z are pointed to object x as shown in the diagram. Hence, if we change the variables of object x, that is reflected when we access the variables of objects y and z. Also if we change the variables of object z, that is reflected when we access the variables of objects x and y.

Implementing Singleton class with method name as that of class name

```
 // Java program implementing Singleton class
// with method name as that of class
class Singleton
{
    // static variable single_instance of type Singleton
    private static Singleton single_instance=null;

    // variable of type String
    public String s;

    // private constructor restricted to this class itself
    private Singleton()
    {
        s = "Hello I am a string part of Singleton class";
```

```
    }

    // static method to create instance of Singleton class
    public static Singleton Singleton()
    {
        // To ensure only one instance is created
        if (single_instance == null)
        {
            single_instance = new Singleton();
        }
        return single_instance;
    }
}

// Driver Code
class Main
{
    public static void main(String args[])
    {
        // instantiating Singleton class with variable x
        Singleton x = Singleton.Singleton();

        // instantiating Singleton class with variable y
        Singleton y = Singleton.Singleton();

        // instantiating Singleton class with variable z
        Singleton z = Singleton.Singleton();

        // changing variable of instance x
        x.s = (x.s).toUpperCase();

        System.out.println("String from x is " + x.s);
        System.out.println("String from y is " + y.s);
        System.out.println("String from z is " + z.s);
        System.out.println("\n");

        // changing variable of instance x
        z.s = (z.s).toLowerCase();

        System.out.println("String from x is " + x.s);
        System.out.println("String from y is " + y.s);
        System.out.println("String from z is " + z.s);
    }
}
```

Output:

```
String from x is HELLO I AM A STRING PART OF SINGLETON CLASS
String from y is HELLO I AM A STRING PART OF SINGLETON CLASS
String from z is HELLO I AM A STRING PART OF SINGLETON CLASS


String from x is hello i am a string part of singleton class
String from y is hello i am a string part of singleton class
String from z is hello i am a string part of singleton class
```

Explanation: In the Singleton class, when we first time call Singleton() method, it creates an object of class Singleton with name single_instance and return it to the variable. Since single_instance is static, it is changed from null to some object. Next time if we try to call Singleton() method, since single_instance is not null, it is returned to the variable, instead of instantiating the Singleton class again.

Source

https://www.geeksforgeeks.org/singleton-class-java/

Chapter 49

Singleton Design Pattern | Implementation

Singleton Design Pattern | Implementation - GeeksforGeeks

Singleton Design Pattern | Introduction

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.
Let's see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.

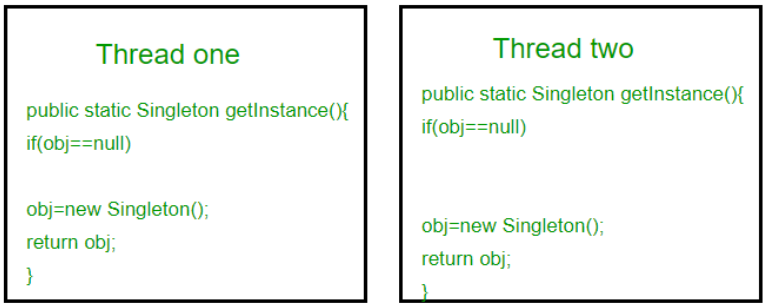
Method 1: Classic Implementation

```
 // Classical Java implementation of singleton
// design pattern
class Singleton
{
    private static Singleton obj;

    // private constructor to force use of
    // getInstance() to create Singleton object
    private Singleton() {}

    public static Singleton getInstance()
```

```
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

Here we have declared getInstance() static so that we can call it without instantiating the class. The first time getInstance() is called it creates a new singleton object and after that it just returns the same object. Note that Singleton obj is not created until we need it and call getInstance() method. This is called lazy instantiation.

The main problem with above method is that it is not thread safe. Consider the following execution sequence.



This execution sequence creates two objects for singleton. Therefore this classic implementation is not thread safe.

Method 2: make getInstance() synchronized

```
 // Thread Synchronized Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj;

    private Singleton() {}

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

Here using synchronized makes sure that only one thread at a time can execute getInstance(). The main disadvantage of this is method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However if performance of getInstance() is not critical for your application this method provides a clean and simple solution.

Method 3: Eager Instantiation

```
 // Static initializer based Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj = new Singleton();

    private Singleton() {}

    public static Singleton getInstance()
    {
        return obj;
    }
}
```

Here we have created instance of singleton in static initializer. JVM executes static initializer when the class is loaded and hence this is guaranteed to be thread safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

Method 4 (Best): Use "Double Checked Locking"
If you notice carefully once an object is created synchronization is no longer useful because now obj will not be null and any sequence of operations will lead to consistent results.
So we will only acquire lock on the getInstance() once, when the obj is null. This way we only synchronize the first way through, just what we want.

```
 // Double Checked Locking based Java implementation of
// singleton design pattern
class Singleton
{
    private volatile static Singleton obj;

    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null)
```

```
        {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj==null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

We have declared the obj volatile which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

References:
Head First Design Patterns book (Highly recommended)
https://en.wikipedia.org/wiki/Singleton_pattern

Source

https://www.geeksforgeeks.org/singleton-design-pattern/

Chapter 50

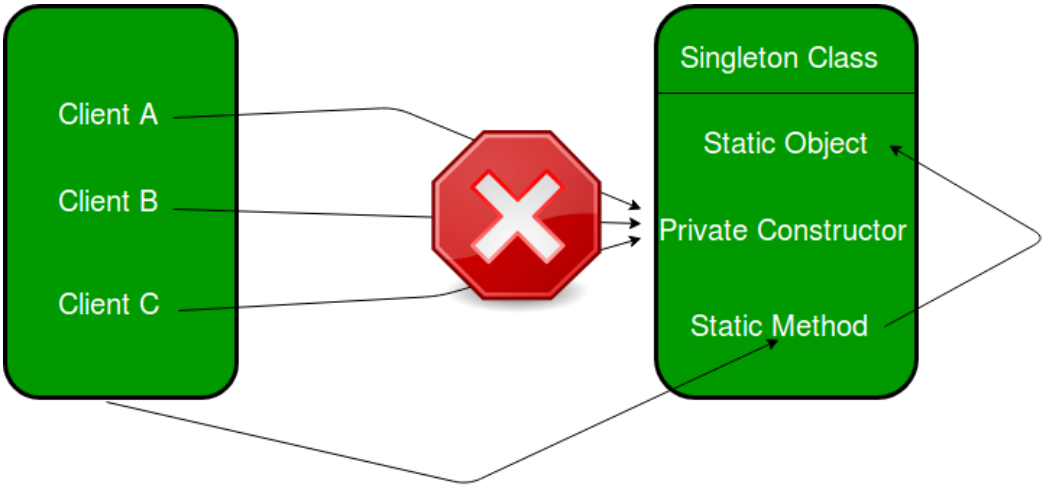
Singleton Design Pattern | Introduction

Singleton Design Pattern | Introduction - GeeksforGeeks

Singleton is a part of **Gang of Four design pattern** and it is categorized under **creational** design patterns. In this article, we are going to take a deeper look into the usage of the Singleton pattern. It is one of the most simple design pattern in terms of the modelling but on the other hand this is one of the most controversial pattern in terms of complexity of usage.

Singleton pattern is a design pattern which restricts a class to instantiate its multiple objects. It is nothing but a way of defining a class. Class is defined in such a way that only one instance of class is created in the complete execution of program or project. It is used where only a single instance of class is required to control the action throughout the execution. A singleton class shouldn't have multiple instances in any case and at any cost. Singleton classes are used for logging, driver objects, caching and thread pool, database connections.

Singleton Design Pattern



Implementation of Singleton class

An implementation of singleton class should have following properties:

1. **It should have only one instance :** This is done by providing instance of class from within the class. Outer classes or subclasses should be prevented to create the instance. This is done by making the constructor private in java so that no class can access the constructor and hence cannot instantiate it.
2. **Instance should be globally accessible :** Instance of singleton class should be globally accessible so that each class can use it. In java it is done by making the access-specifier of instance public.

```
 //A singleton class should have public visiblity
//so that complete application can use
public class GFG {

  //static instance of class globally accessible
  public static GFG instance = new GFG();
  private GFG() {
    // private constructor so that class
    //cannot be instantiated from outside
    //this class
  }
}
```

Detailed Article: Implementation of Singleton Design Pattern in Java

Initialization Types of Singleton

1. **Early initialization :** In this method, class is initialized whether it is to be used or not. Main advantage of this method is its simplicity. You initiate the class at the time of class loading. Its drawback is that class is always initialized whether it is being used or not.

2. **Lazy initialization :** In this method, class in initialized only when it is required. It can save you from instantiating the class when you don't need it. Generally lazy initialization is used when we create a singleton class.

Examples of Singleton class

1. **java.lang.Runtime :** Java provides a class Runtime in its lang package which is singleton in nature. Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime() method.
   An application cannot instantiate this class so multiple objects can't be created for this class. Hence Runtime is a singleton class.

2. **java.awt.Desktop :** The Desktop class allows a Java application to launch associated applications registered on the native desktop to handle a URI or a file.
   Supported operations include:

   - launching the user-default browser to show a specified URI;
     launching the user-default mail client with an optional mailto URI;
   - launching a registered application to open, edit or print a specified file.
   - This class provides methods corresponding to these operations. The methods look for the associated application registered on the current platform, and launch it to handle a URI or file. If there is no associated application or the associated application fails to be launched, an exception is thrown.
   - Each operation is an action type represented by the Desktop.Action class.

   This class also cannot be instantiated from application. Hence it is also a singleton class.

Applications of Singleton classes

There is a lot of applications of singleton pattern like cache-memory, database connection, drivers, logging. Some major of them are :-

1. **Hardware interface access:** The use of singleton depends on the requirements. Singleton classes are also used to prevent concurrent access of class. Practically singleton can be used in case external hardware resource usage limitation required e.g. Hardware printers where the print spooler can be made a singleton to avoid multiple concurrent accesses and creating deadlock.
2. **Logger :** Singleton classes are used in log file generations. Log files are created by logger class object. Suppose an application where the logging utility has to produce one log file based on the messages received from the users. If there is multiple client application using this logging utility class they might create multiple instances of this class and it can potentially cause issues during concurrent access to the same logger file. We can use the logger utility class as a singleton and provide a global point of reference, so that each user can use this utility and no 2 users access it at same time.

3. **Configuration File:** This is another potential candidate for Singleton pattern because this has a performance benefit as it prevents multiple users to repeatedly access and read the configuration file or properties file. It creates a single instance of the configuration file which can be accessed by multiple calls concurrently as it will provide static config data loaded into in-memory objects. The application only reads from the configuration file at the first time and there after from second call onwards the client applications read the data from in-memory objects.
4. **Cache:** We can use the cache as a singleton object as it can have a global point of reference and for all future calls to the cache object the client application will use the in-memory object.

Important points

- Singleton classes can have only one instance and that instance should be globally accessible.
- java.lang.Runtime and java.awt.Desktop are 2 singleton classes provided by JVM.
- Singleton Design pattern is a type of creational design pattern.
- Outer classes should be prevented to create instance of singleton class.

References:-
https://en.wikipedia.org/wiki/Singleton_pattern
https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html
https://docs.oracle.com/javase/7/docs/api/java/awt/Desktop.html

Source

https://www.geeksforgeeks.org/singleton-design-pattern-introduction/

Chapter 51

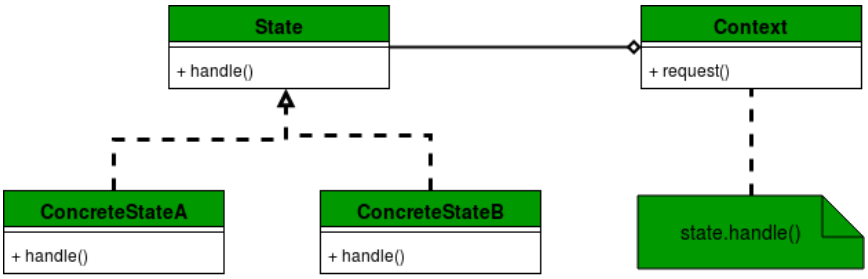
State Design Pattern

State Design Pattern - GeeksforGeeks

State pattern is one of the behavioral design pattern. State design pattern is used when an Object changes its behavior based on its internal state.

If we have to change behavior of an object based on its state, we can have a state variable in the Object and use if-else condition block to perform different actions based on the state. State pattern is used to provide a systematic and lose-coupled way to achieve this through Context and State implementations.

UML Diagram of State Design Pattern



- **Context:** Defines an interface to client to interact. It maintains references to concrete state object which may be used to define current state of object.
- **State:** Defines interface for declaring what each concrete state should do.
- **ConcreteState:** Provides implementation for methods defined in State.

Example of State Design Pattern
In below example, we have implemented a mobile state scenario . With respect to alerts, a mobile can be in different states. For example, vibration and silent. Based on this alert state, behavior of the mobile changes when an alert is to be done.

```
 // Java program to demonstrate working of
// State Design Pattern

interface MobileAlertState
{
    public void alert(AlertStateContext ctx);
}

class AlertStateContext
{
    private MobileAlertState currentState;

    public AlertStateContext()
    {
        currentState = new Vibration();
    }

    public void setState(MobileAlertState state)
    {
        currentState = state;
    }

    public void alert()
    {
        currentState.alert(this);
    }
}

class Vibration implements MobileAlertState
{
    @Override
    public void alert(AlertStateContext ctx)
    {
         System.out.println("vibration...");
    }

}

class Silent implements MobileAlertState
{
    @Override
    public void alert(AlertStateContext ctx)
    {
        System.out.println("silent...");
    }

}
```

```
class StatePattern
{
    public static void main(String[] args)
    {
        AlertStateContext stateContext = new AlertStateContext();
        stateContext.alert();
        stateContext.alert();
        stateContext.setState(new Silent());
        stateContext.alert();
        stateContext.alert();
        stateContext.alert();
    }
}
```

Output:

```
vibration...
vibration...
silent...
silent...
silent...
```

Advantages of State Design Pattern

- With State pattern, the benefits of implementing polymorphic behavior are evident, and it is also easier to add states to support additional behavior.
- In the State design pattern, an object's behavior is the result of the function of its state, and the behavior gets changed at runtime depending on the state. This removes the dependency on the if/else or switch/case conditional logic. For example, in the TV remote scenario, we could have also implemented the behavior by simply writing one class and method that will ask for a parameter and perform an action (switch the TV on/off) with an if/else block.
- The State design pattern also improves Cohesion since state-specific behaviors are aggregated into the ConcreteState classes, which are placed in one location in the code.

Disadvantages of State Design Pattern

- The State design pattern can be used when we need to change state of object at runtime by inputting in it different subclasses of some State base class. This circumstance is advantage and disadvantage in the same time, because we have a clear separate State classes with some logic and from the other hand the number of classes grows up.

Source

https://www.geeksforgeeks.org/state-design-pattern/

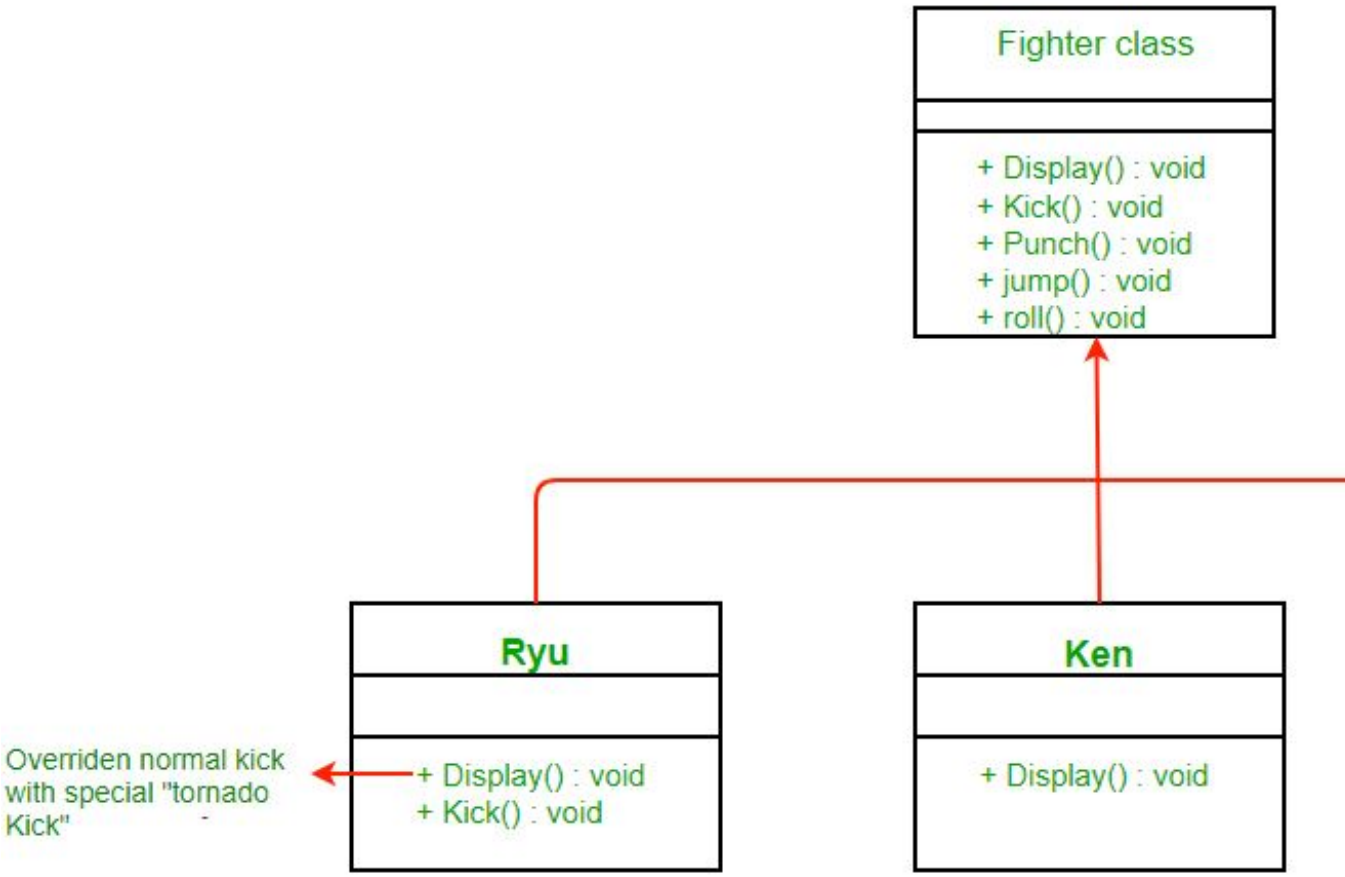
Chapter 52

Strategy Pattern | Set 1 (Introduction)

Strategy Pattern | Set 1 (Introduction) - GeeksforGeeks

As always we will learn this pattern by defining a problem and using strategy pattern to solve it. Suppose we are building a game "Street Fighter". For simplicity assume that a character may have four moves that is kick, punch, roll and jump. Every character has kick and punch moves, but roll and jump are optional. How would you model your classes? Suppose initially you use inheritance and abstract out the common features in a **Fighter** class and let other characters subclass **Fighter** class.

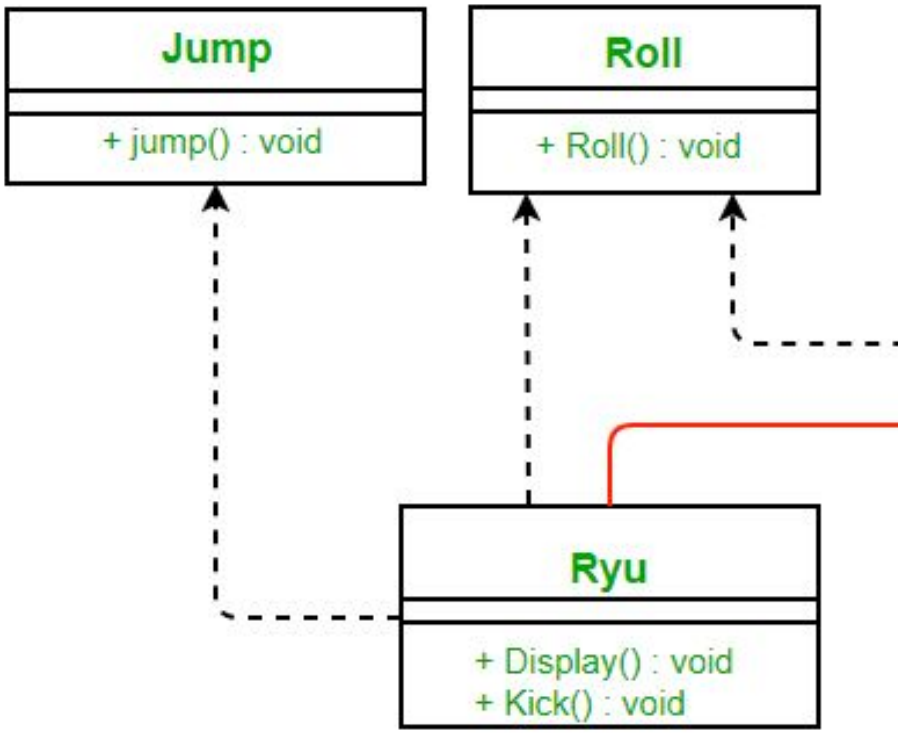
Fighter class will we have default implementation of normal actions. Any character with specialized move can override that action in its subclass. Class diagram would be as follows:



What are the problems with above design?

What if a character doesn't perform jump move? It still inherits the jump behavior from superclass. Although you can override jump to do nothing in that case but you may have to do so for many existing classes and take care of that for future classes too. This would also make maintenance difficult. So we can't use inheritance here.

What about an Interface?



Take a look at the following design:

It's much cleaner. We took out some actions (which some characters might not perform) out of **Fighter**class and made interfaces for them. That way only characters that are supposed to jump will implement the **JumpBehavior.**

What are the problems with above design?

The main problem with the above design is code reuse. Since there is no default implementation of jump and roll behavior we may have code duplicity. You may have to rewrite the same jump behavior over and over in many subclasses.

How can we avoid this?

What if we made **JumpBehavior** and **RollBehavior** classes instead of interface? Well then we would have to use multiple inheritance that is not supported in many languages due to many problems associated with it.

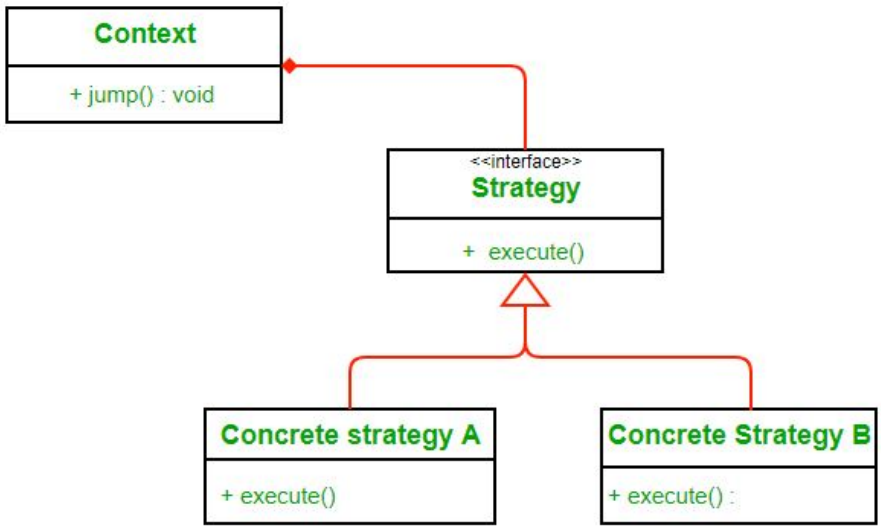
Here strategy pattern comes to our rescue. We will learn what the strategy pattern is and then apply it to solve our problem.

Definition:

Wikipedia defines strategy pattern as:

*"In computer programming, the **strategy pattern** (also known as the **policy pattern**) is a software design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern*

- *defines a family of algorithms,*
- *encapsulates each algorithm, and*
- *makes the algorithms interchangeable within that family."*



***Class* Diagram:**

Here we rely on composition instead of inheritance for reuse. **Context** is composed of a **Strategy**. Instead of implementing a behavior the **Context** delegates it to **Strategy**. The context would be the class that would require changing behaviors. We can change behavior dynamically. **Strategy** is implemented as interface so that we can change behavior without affecting our context.

We will have a clearer understanding of strategy pattern when we will use it to solve our problem.

Advantages:

1. A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
2. By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
3. The application can switch strategies at run-time.
4. Strategy enables the clients to choose the required algorithm, without using a "switch" statement or a series of "if-else" statements.
5. Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

Disadvantages:

1. The application must be aware of all the strategies to select the right one for the right situation.
2. Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
3. In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.

References:

- Head First Design Patterns
- http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_517_Fall_2007/wiki1b_8_sa
- https://en.wikipedia.org/wiki/Strategy_pattern

Source

https://www.geeksforgeeks.org/strategy-pattern-set-1/

Chapter 53

Strategy Pattern | Set 2 (Implementation)

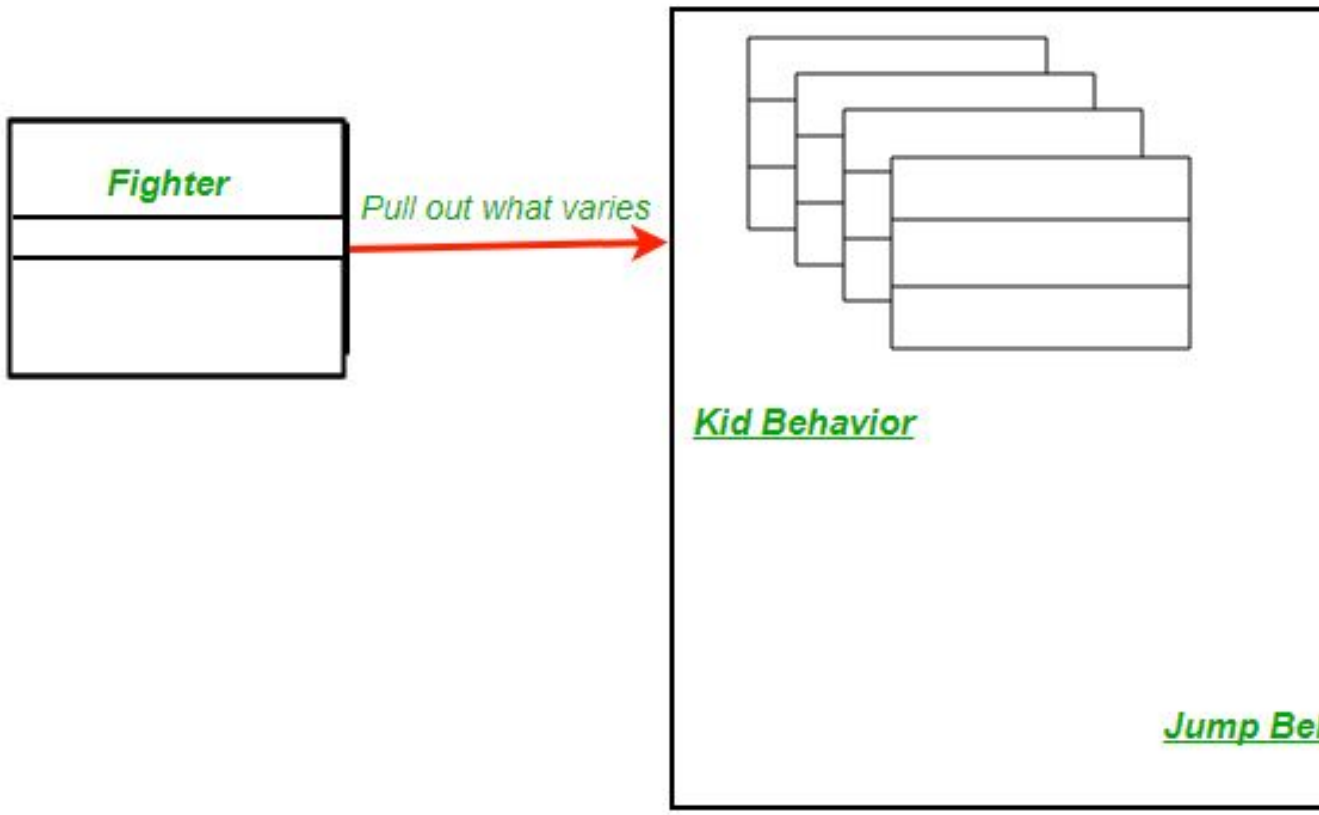
Strategy Pattern | Set 2 (Implementation) - GeeksforGeeks

We have discussed a fighter example and introduced Strategy Pattern in set 1.

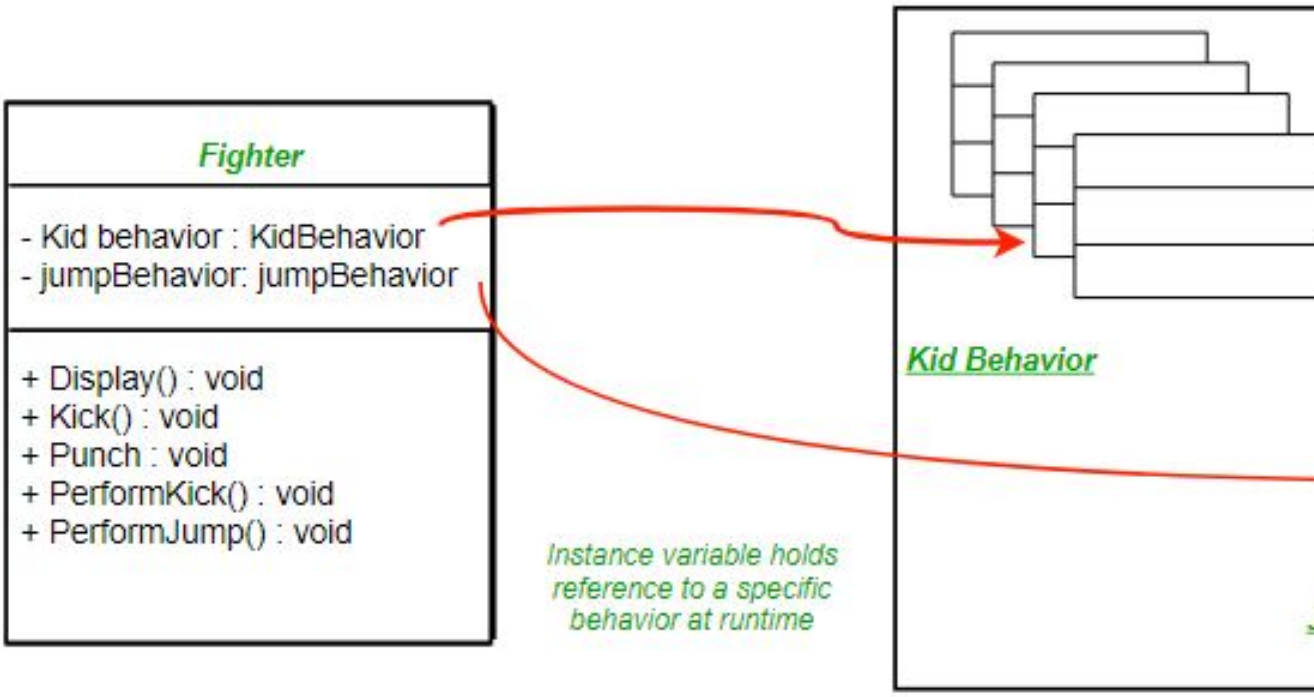
Strategy Pattern | Set 1 (Introduction)

In this post, we apply Strategy Pattern to the Fighter Problem and discuss implementation.

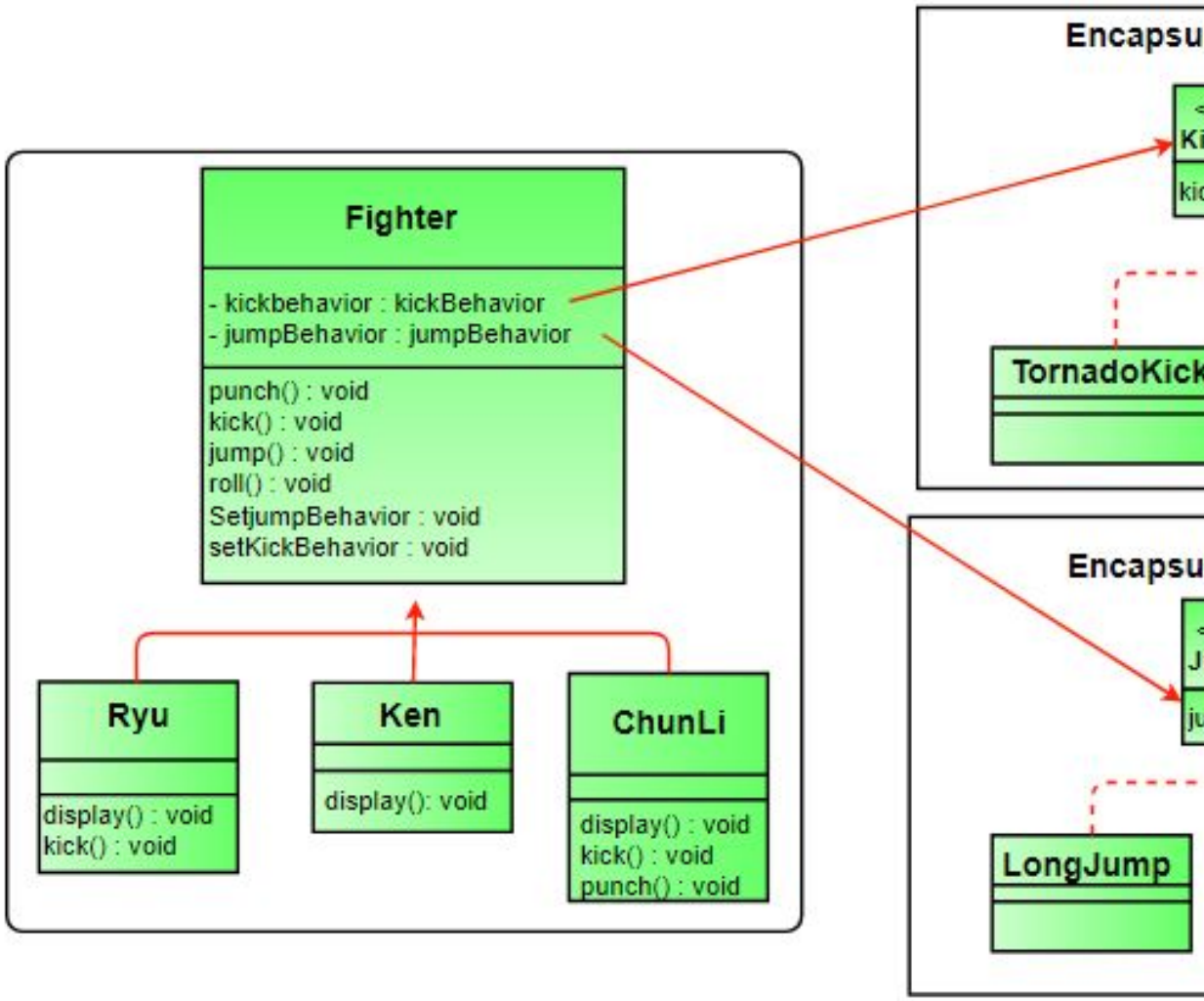
The first step is to identify the behaviors that may vary across different classes in future and separate them from the rest. For our example let them be kick and jump behaviors. To separate these behaviors we will pull both methods out of **Fighter** class and create a new set of classes to represent each behavior.



The Fighter class will now delegate its kick and jump behavior instead of using kick and jump methods defined in the Fighter class or its subclass.



After reworking the final class diagram would be (Click on image for better view):



Comparing our design to the definition of strategy pattern encapsulated kick and jump behaviors are two families of algorithms. And these algorithms are interchangeable as evident in implementation.

Below is the Java implementation of the same.

```
 // Java program to demonstrate implementation of
// Strategy Pattern

// Abstract as you must have a specific fighter
```

```
abstract class Fighter
{
    KickBehavior kickBehavior;
    JumpBehavior jumpBehavior;

    public Fighter(KickBehavior kickBehavior,
                   JumpBehavior jumpBehavior)
    {
        this.jumpBehavior = jumpBehavior;
        this.kickBehavior = kickBehavior;
    }
    public void punch()
    {
        System.out.println("Default Punch");
    }
    public void kick()
    {
        // delegate to kick behavior
        kickBehavior.kick();
    }
    public void jump()
    {

        // delegate to jump behavior
        jumpBehavior.jump();
    }
    public void roll()
    {
        System.out.println("Default Roll");
    }
    public void setKickBehavior(KickBehavior kickBehavior)
    {
        this.kickBehavior = kickBehavior;
    }
    public void setJumpBehavior(JumpBehavior jumpBehavior)
    {
        this.jumpBehavior = jumpBehavior;
    }
    public abstract void display();
}

// Encapsulated kick behaviors
interface KickBehavior
{
    public void kick();
}
class LightningKick implements KickBehavior
{
```

```
    public void kick()
    {
        System.out.println("Lightning Kick");
    }
}
class TornadoKick implements KickBehavior
{
    public void kick()
    {
        System.out.println("Tornado Kick");
    }
}

// Encapsulated jump behaviors
interface JumpBehavior
{
    public void jump();
}
class ShortJump implements JumpBehavior
{
    public void jump()
    {
        System.out.println("Short Jump");
    }
}
class LongJump implements JumpBehavior
{
    public void jump()
    {
        System.out.println("Long Jump");
    }
}

// Characters
class Ryu extends Fighter
{
    public Ryu(KickBehavior kickBehavior,
               JumpBehavior jumpBehavior)
    {
        super(kickBehavior,jumpBehavior);
    }
    public void display()
    {
        System.out.println("Ryu");
    }
}
class Ken extends Fighter
{
```

```
    public Ken(KickBehavior kickBehavior,
               JumpBehavior jumpBehavior)
    {
        super(kickBehavior,jumpBehavior);
    }
    public void display()
    {
        System.out.println("Ken");
    }
}
class ChunLi extends Fighter
{
    public ChunLi(KickBehavior kickBehavior,
                  JumpBehavior jumpBehavior)
    {
        super(kickBehavior,jumpBehavior);
    }
    public void display()
    {
        System.out.println("ChunLi");
    }
}

// Driver class
class StreetFighter
{
    public static void main(String args[])
    {
        // let us make some behaviors first
        JumpBehavior shortJump = new ShortJump();
        JumpBehavior LongJump = new LongJump();
        KickBehavior tornadoKick = new TornadoKick();

        // Make a fighter with desired behaviors
        Fighter ken = new Ken(tornadoKick,shortJump);
        ken.display();

        // Test behaviors
        ken.punch();
        ken.kick();
        ken.jump();

        // Change behavior dynamically (algorithms are
        // interchangeable)
        ken.setJumpBehavior(LongJump);
        ken.jump();
    }
}
```

Output :

```
Ken
Default Punch
Tornado Kick
Short Jump
Long Jump
```

References:
Head First Design Patterns

Source

https://www.geeksforgeeks.org/strategy-pattern-set-2/

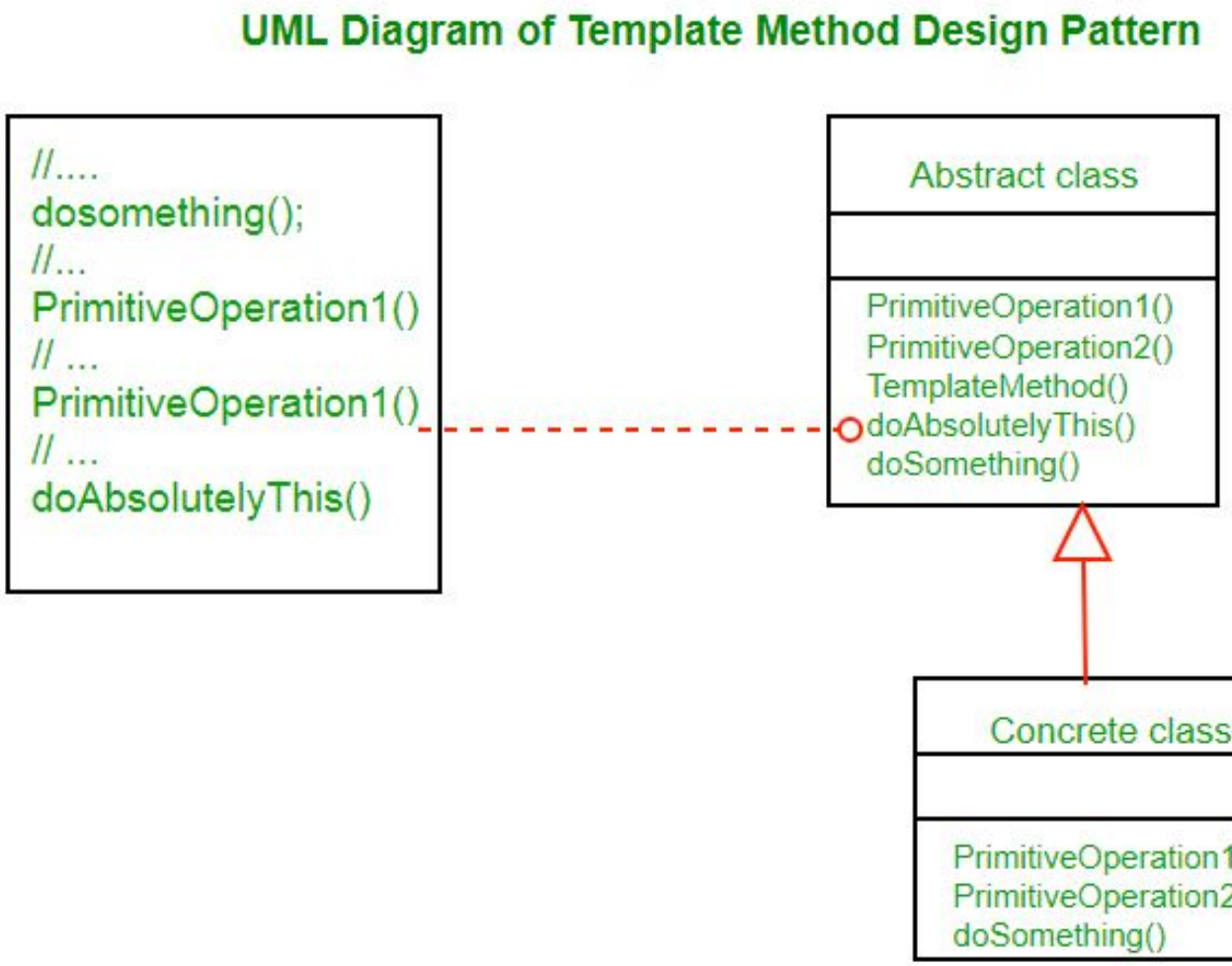
Chapter 54

Template Method Design Pattern

Template Method Design Pattern - GeeksforGeeks

Template method design pattern is to define an algorithm as skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm is preserved by the parent class.
Template means Preset format like HTML templates which has fixed preset format.Similarly in template method pattern,we have a preset structure method called template method which consists of steps.This steps can be abstract method which will be implemented by its subclasses.

This behavioral design pattern is one of the easiest to understand and implement. This design pattern is used popularly in framework development. This helps to avoid code duplication also.



Source : Wikipedia

- **AbstractClass** contains the templateMethod() which should be made final so that it cannot be overridden. This template method makes use of other operations available in order to run the algorithm but is decoupled for the actual implementation of these methods. All operations used by this template method are made abstract, so their implementation is deferred to subclasses.
- **ConcreteClass** implements all the operations required by the templateMethod that were defined as abstract in the parent class. There can be many different Concrete-Classes.

Lets see an example of the template method pattern.

```
abstract class OrderProcessTemplate
```

```
{
    public boolean isGift;

    public abstract void doSelect();

    public abstract void doPayment();

    public final void giftWrap()
    {
        try
        {
            System.out.println("Gift wrap successfull");
        }
        catch (Exception e)
        {
            System.out.println("Gift wrap unsuccessful");
        }
    }

    public abstract void doDelivery();

    public final void processOrder(boolean isGift)
    {
        doSelect();
        doPayment();
        if (isGift) {
            giftWrap();
        }
        doDelivery();
    }
}


class NetOrder extends OrderProcessTemplate
{
    @Override
    public void doSelect()
    {
        System.out.println("Item added to online shopping cart");
        System.out.println("Get gift wrap preference");
        System.out.println("Get delivery address.");
    }

    @Override
    public void doPayment()
    {
        System.out.println
                    ("Online Payment through Netbanking, card or Paytm");
```

```
    }

    @Override
    public void doDelivery()
    {
        System.out.println
                    ("Ship the item through post to delivery address");
    }

}

class StoreOrder extends OrderProcessTemplate
{

    @Override
    public void doSelect()
    {
        System.out.println("Customer chooses the item from shelf.");
    }

    @Override
    public void doPayment()
    {
        System.out.println("Pays at counter through cash/POS");
    }

    @Override
    public void doDelivery()
    {
        System.out.println("Item deliverd to in delivery counter.");
    }

}

class TemplateMethodPatternClient
{
    public static void main(String[] args)
    {
        OrderProcessTemplate netOrder = new NetOrder();
        netOrder.processOrder(true);
        System.out.println();
        OrderProcessTemplate storeOrder = new StoreOrder();
        storeOrder.processOrder(true);
    }
}
```

Output :

```
Item added to online shopping cart
Get gift wrap preference
Get delivery address.
Online Payment through Netbanking, card or Paytm
Gift wrap successfull
Ship the item through post to delivery address

Customer chooses the item from shelf.
Pays at counter through cash/POS
Gift wrap successfull
Item deliverd to in delivery counter.
```

The above example deals with order processing flow. The OrderProcessTemplate class is an abstract class containing the algorithm skeleton. As shown on note, processOrder() is the method that contains the process steps. We have two subclasses NetOrder and StoreOrder which has the same order processing steps.

So the overall algorithm used to process an order is defined in the base class and used by the subclasses. But the way individual operations are performed vary depending on the subclass.

When to use template method

The template method is used in frameworks, where each implements the invariant parts of a domain's architecture, leaving "placeholders" for customization options.

The template method is used for the following reasons :

- Let subclasses implement varying behavior (through method overriding)
- Avoid duplication in the code , the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in the subclasses.
- Control at what points subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.

Reference :
Wikipedia

Source

https://www.geeksforgeeks.org/template-method-design-pattern/

Chapter 55

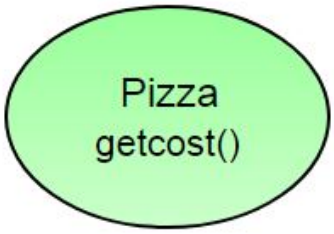
The Decorator Pattern | Set 2 (Introduction and Design)

The Decorator Pattern | Set 2 (Introduction and Design) - GeeksforGeeks

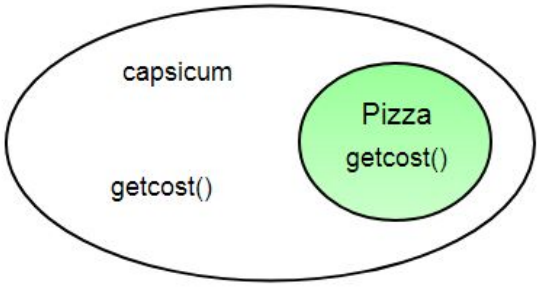
As we saw our previous designs using inheritance didn't work out that well. In this article, decorator pattern is discussed for the design problem in previous set.

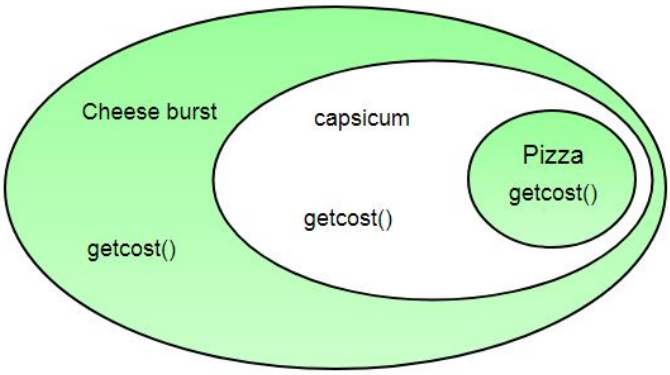
So what we do now is take a pizza and "decorate" it with toppings at runtime:

1. Take a pizza object.

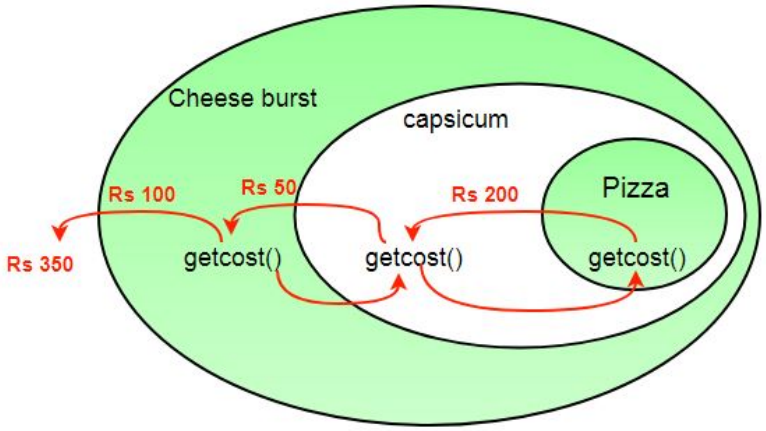


2. "Decorate" it with a Capsicum object.





3. "Decorate" it with a CheeseBurst object.
4. Call getCost() and use delegation instead of inheritance to calculate the toppings cost.



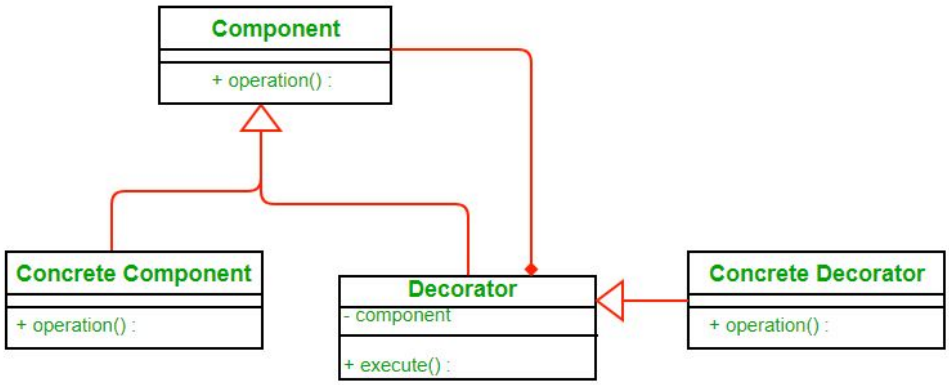
What we get in the end is a pizza with cheeseburst and capsicum toppings. Visualize the "decorator" objects like wrappers. Here are some of the properties of decorators:

- Decorators have the same super type as the object they decorate.
- You can use multiple decorators to wrap an the object.

- Since decorators have same type as object, we can pass around decorated object instead of original.
- We can decorate objects at runtime.

Definition:

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Class Diagram: Image src: Wikipedia

- Each component can be used on its own or may be wrapped by a decorator.
- Each decorator has an instance variable that holds the reference to component it decorates(HAS-A relationship).
- The ConcreteComponent is the object we are going to dynamically decorate.

Advantages:

- The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime**.**
- The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.
- Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

Disadvantages:

- Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.
- It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

References:

- Head First Design Patterns(Book)
- https://neillmorgan.wordpress.com/2010/02/07/decorator-pattern-pros-and-cons/
- https://en.wikipedia.org/wiki/Decorator_pattern
- http://stackoverflow.com/questions/4842978/decorator-pattern-versus-sub-classing

In the next post, we will be discussing implementation of decorator pattern.

Source

https://www.geeksforgeeks.org/the-decorator-pattern-set-2-introduction-and-design/

Chapter 56

Unified Modeling Language (UML) | Activity Diagrams

Unified Modeling Language (UML) | Activity Diagrams - GeeksforGeeks

We use **Activity Diagrams** to illustrate the flow of control in a system and refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.

UML models basically three types of diagrams, namely, structure diagrams, interaction diagrams, and behavior diagrams. An activity diagram is a **behavioral diagram** i.e. it depicts the behavior of a system.

An activity diagram portrays the control flow from a start point to a finish point showing the various decision paths that exist while the activity is being executed. We can depict both sequential processing and concurrent processing of activities using an activity diagram. They are used in business and process modelling where their primary use is to depict the dynamic aspects of a system.

An activity diagram is very **similar to a flowchart**. So let us understand if an activity diagrams or a flowcharts are any different :

Flowcharts were typically invented earlier than activity diagrams. Non programmers use Flow charts to model workflows. For example: A manufacturer uses a flow chart to explain and illustrate how a particular product is manufactured. We can call a flowchart a primitive version of an activity diagram. Business processes where decision making is involved is expressed using a flow chart.

So, programmers use activity diagrams (advanced version of a flowchart) to depict workflows. An activity diagram is **used by developers** to understand the flow of programs on a high level. It also enables them to figure out constraints and conditions that cause particular events. A flow chart converges into being an activity diagram if complex decisions are being made.

Brevity is the soul of wit. We need to convey a lot of information with clarity and make sure it is short. So an activity diagram helps people on both sides i.e. Businessmen and Developers to interact and understand systems.

A question arises:

Do we need to use both the diagram and the textual documentation?
Different individuals have different preferences in which they understand something. For example: To understand a concept, some people might prefer a written tutorial with images while others would prefer a video lecture.
So we generally use both the diagram and the textual documentation to make our system description as clear as possible. We also need to be sensitive to the needs of the audience that we are catering to at times.

Difference between a Use case diagram and an Activity diagram

An activity diagram is used to model the workflow depicting conditions, constraints, sequential and concurrent activities. On the other hand, the purpose of a Use Case is to just depict the functionality i.e. what the system does and not how it is done. So in simple terms, an activity diagram shows 'How' while a Use case shows 'What' for a particular system.
The levels of abstraction also vary for both of them. An activity diagram can be used to illustrate a business process (high level implementation) to a stand alone algorithm (ground level implementation). However, Use cases have a low level of abstraction. They are used to show a **high level** of implementation only.

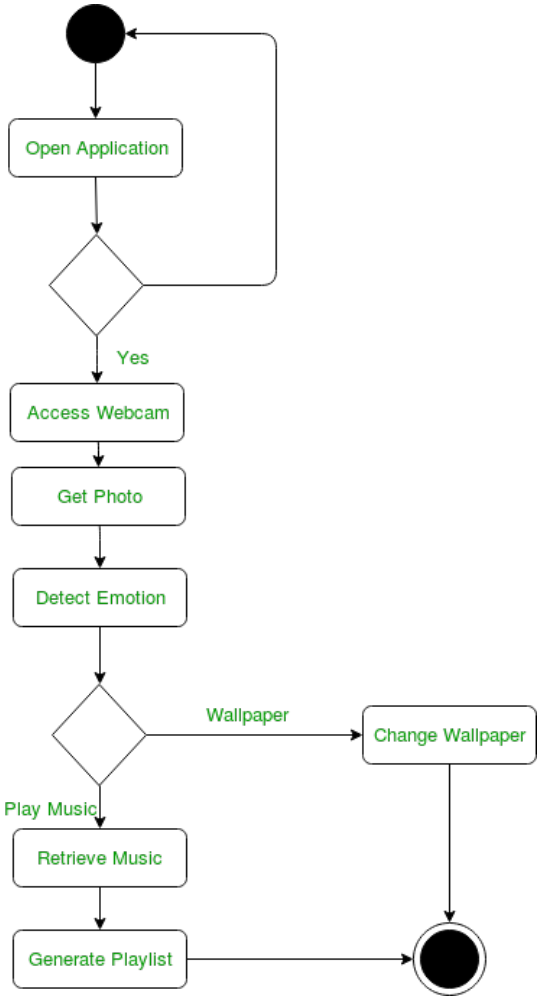


Figure – an activity diagram for an emotion based music player

The above figure depicts an activity diagram for an emotion based music player which can also be used to change the wallpaper.

The various components used in the diagram and the standard notations are explained below.

1. **Initial State** – The starting state before an activity takes place is depicted using the initial state.

   Figure – notation for initial state or start state

   A process can have only one initial state unless we are depicting nested activities. We use a black filled circle to depict the initial state of a system. For objects, this is the state when they are instantiated. The Initial State from the UML Activity Diagram marks the entry point and the initial Activity State.

For example – Here the initial state is the state of the system before the application is opened.

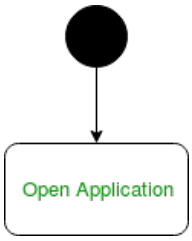


Figure – initial state symbol being used

2. **Action or Activity State** – An activity represents execution of an action on objects or by objects. We represent an activity using a rectangle with rounded corners. Basically any action or event that takes place is represented using an activity.

Figure – notation for an activity state

For example – Consider the previous example of opening an application opening the application is an activity state in the activity diagram.

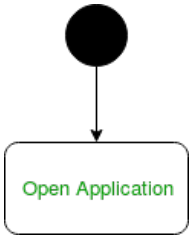


Figure – activity state symbol being used

3. **Action Flow or Control flows** – Action flows or Control flows are also referred to as paths and edges. They are used to show the transition from one activity state to another.

Figure – notation for control Flow

An activity state can have multiple incoming and outgoing action flows. We use a line with an arrow head to depict a Control Flow. If there is a constraint to be adhered to while making the transition it is mentioned on the arrow.

Consider the example – Here both the states transit into one final state using action flow symbols i.e. arrows.

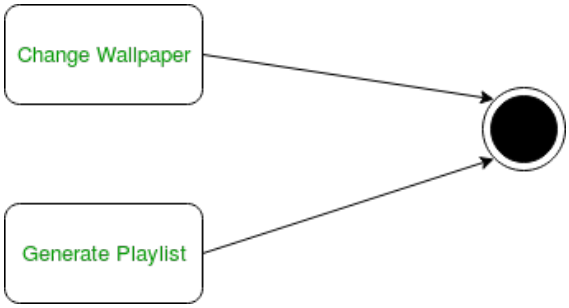


Figure – using action flows for transitions

4. **Decision node and Branching** – When we need to make a decision before deciding the flow of control, we use the decision node.

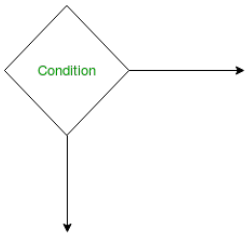


Figure – notation for decision node

The outgoing arrows from the decision node can be labelled with conditions or guard expressions.It always includes two or more output arrows.

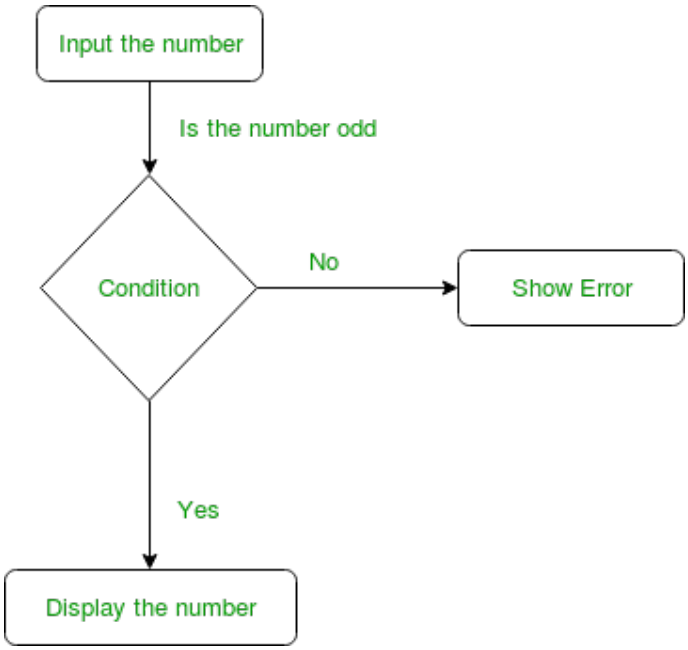


Figure – an activity diagram using decision node

5. **Guards** – A Guard refers to a statement written next to a decision node on an arrow sometimes within square brackets.

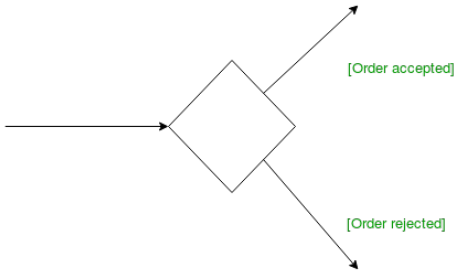


Figure – guards being used next to a decision node

The statement must be true for the control to shift along a particular direction. Guards help us know the constraints and conditions which determine the flow of a process.

6. **Fork** – Fork nodes are used to support concurrent activities.

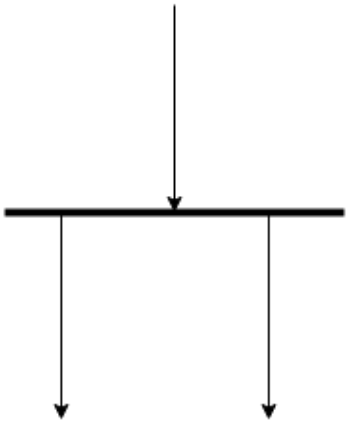


Figure – fork notation

When we use a fork node when both the activities get executed concurrently i.e. no decision is made before splitting the activity into two parts. Both parts need to be executed in case of a fork statement.
We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent activity state and outgoing arrows towards the newly created activities.
For example: In the example below, the activity of making coffee can be split into two concurrent activities and hence we use the fork notation.

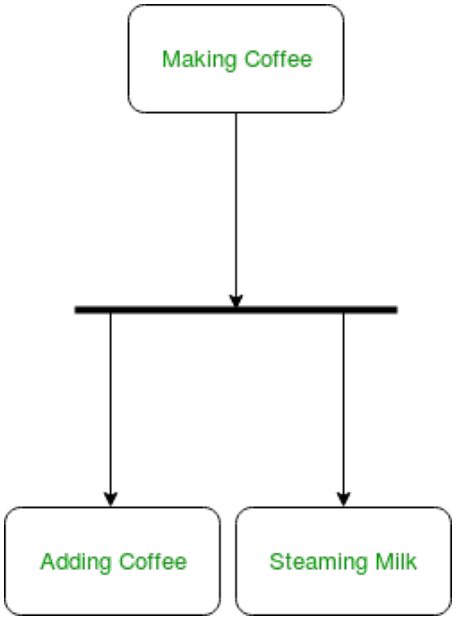


Figure – a diagram using fork

7. **Join** – Join nodes are used to support concurrent activities converging into one. For join notations we have two or more incoming edges and one outgoing edge.

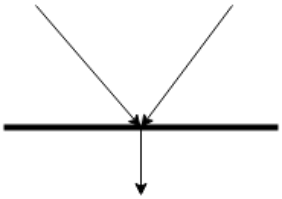


Figure – join notation

For example – When both activities i.e. steaming the milk and adding coffee get completed, we converge them into one final activity.

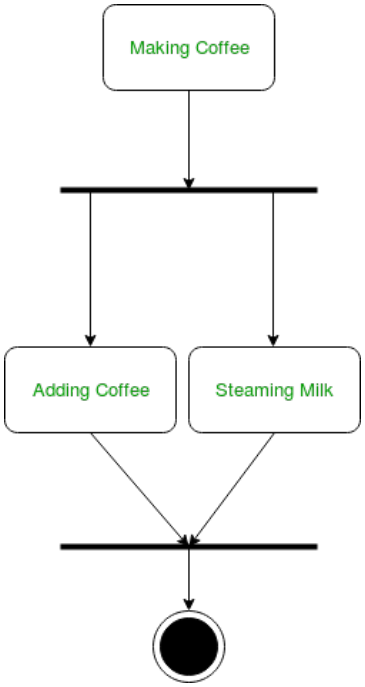


Figure – a diagram using join notation

8. **Merge or Merge Event** – Scenarios arise when activities which are not being executed concurrently have to be merged. We use the merge notation for such scenarios. We can merge two or more activities into one if the control proceeds onto the next activity irrespective of the path chosen.

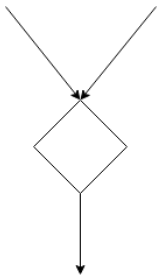


Figure – merge notation

For example – In the diagram below: we can't have both sides executing concurrently, but they finally merge into one. A number can't be both odd and even at the same time.

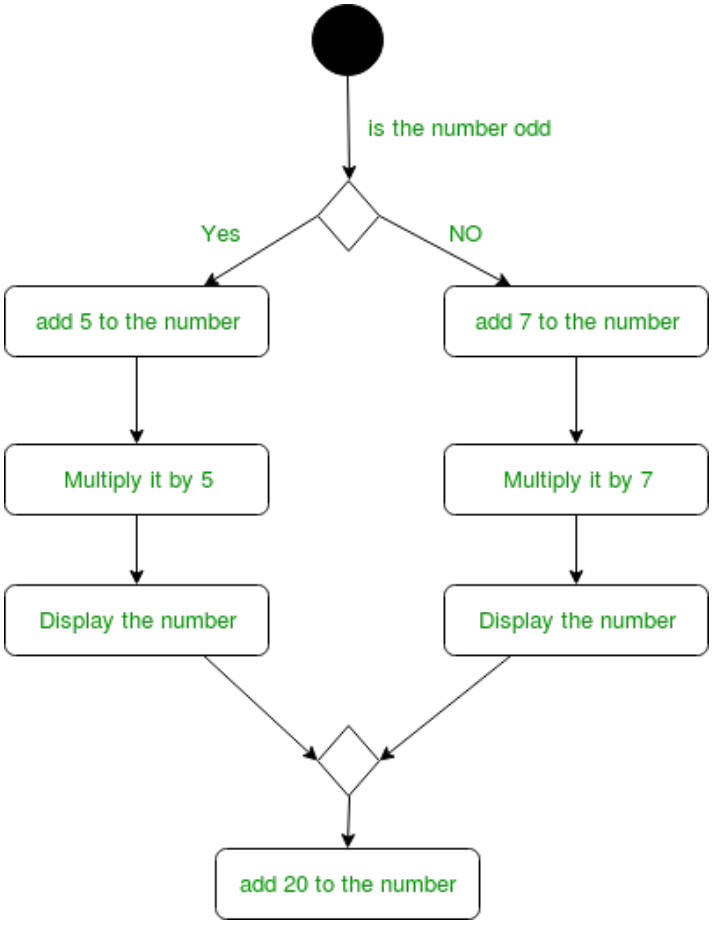


Figure – an activity diagram using merge notation

9. **Swimlanes** – We use swimlanes for grouping related activities in one column. Swimlanes group related activities into one column or one row. Swimlanes can be vertical and horizontal. Swimlanes are used to add modularity to the activity diagram. It is not mandatory to use swimlanes. They usually give more clarity to the activity diagram. It's similar to creating a function in a program. It's not mandatory to do so, but, it is a recommended practice.

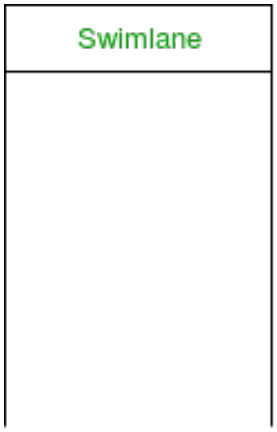


Figure – swimlanes notation
We use a rectangular column to represent a swimlane as shown in the figure above.

For example – Here different set of activities are executed based on if the number is odd or even. These activities are grouped into a swimlane.

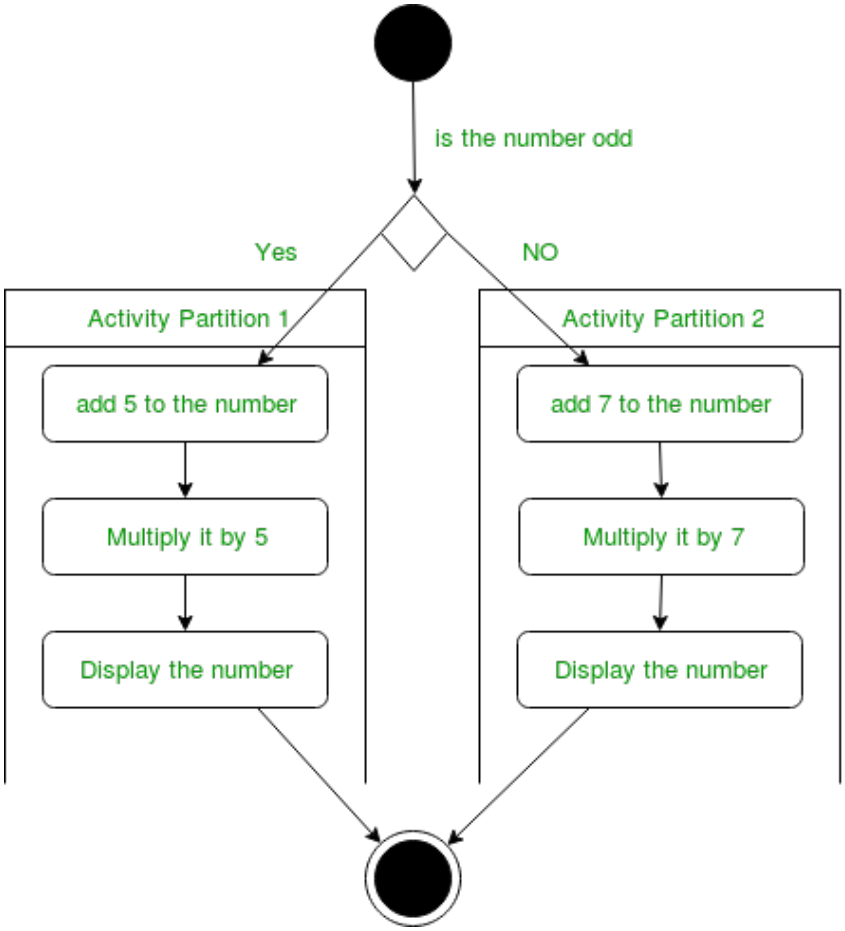


Figure – an activity diagram making use of swimlanes

10. **Time Event** –

Figure – time event notation

We can have a scenario where an event takes some time to complete. We use an hourglass to represent a time event.
For example – Let us assume that the processing of an image takes takes a lot of time. Then it can be represented as shown below.

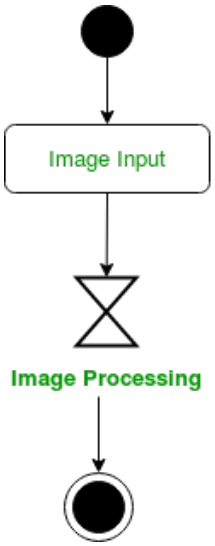


Figure – an activity diagram using time event

11. **Final State or End State** – The state which the system reaches when a particular process or activity ends is known as a Final State or End State. We use a filled circle within a circle notation to represent the final state in a state machine diagram. A system or a process can have multiple final states.

Figure – notation for final state

1. Identify the initial state and the final states.
2. Identify the intermediate activities needed to reach the final state from he initial state.
3. Identify the conditions or constraints which cause the system to change control flow.
4. Draw the diagram with appropriate notations.

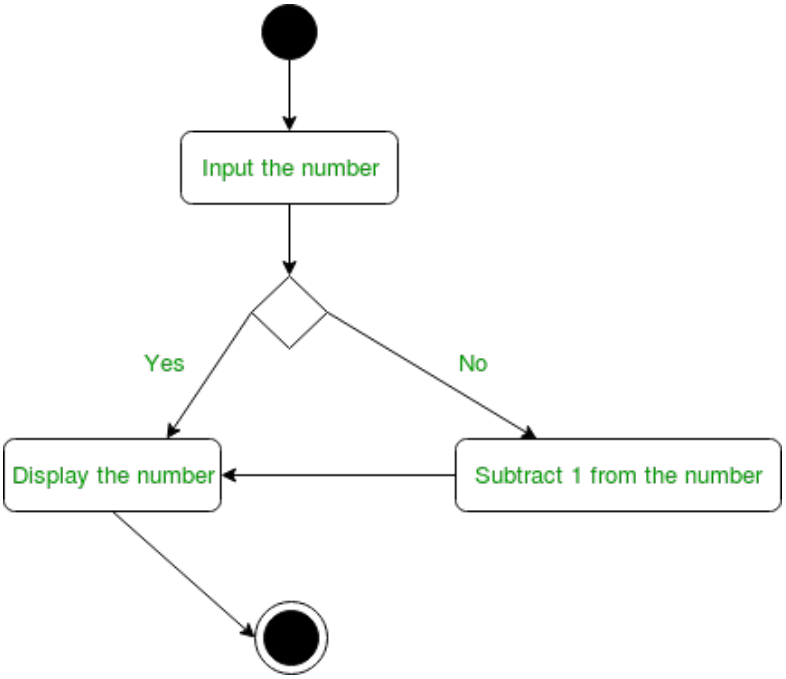


Figure – an activity diagram

The above diagram prints the number if it is odd otherwise it subtracts one from the number and displays it.

- Dynamic modelling of the system or a process.
- Illustrate the various steps involved in a UML use case.
- Model software elements like methods,operations and functions.
- We can use Activity diagrams to depict concurrent activities easily.
- Show the constraints, conditions and logic behind algorithms.

References –
Activity diagrams – IBM
Activity Diagram – sparxsystems

Source

https://www.geeksforgeeks.org/unified-modeling-language-uml-activity-diagrams/

Chapter 57

Unified Modeling Language (UML) | An Introduction

Unified Modeling Language (UML) | An Introduction - GeeksforGeeks

Unified Modeling Language (UML) is a general purpose modelling language. The main aim of UML is define a standard way to **visualize** the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.

UML is **not a programming language**, it is rather a visual language. We use UML diagrams to portray the **behavior and structure** of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO) published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

Do we really need UML?

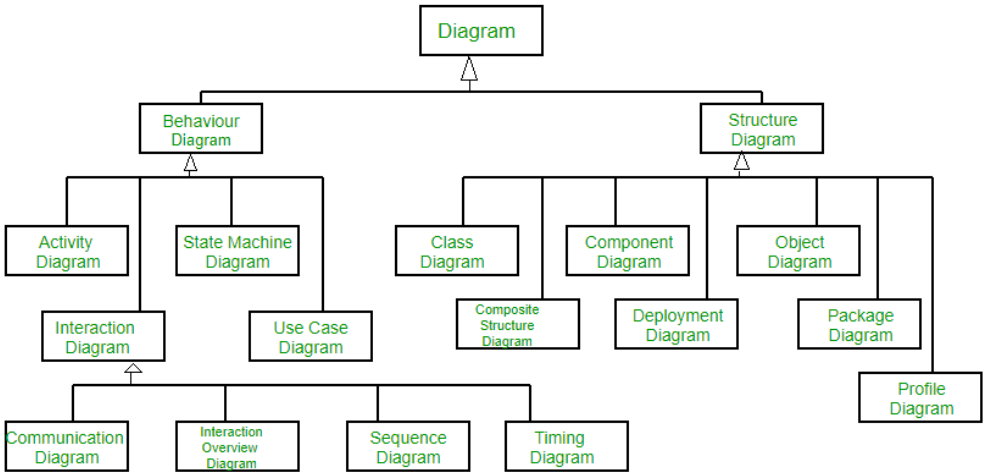
- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

UML is linked with **object oriented** design and analysis. UML makes the use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:

1. **Structural Diagrams** – Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.

2. **Behavior Diagrams** – Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

The image below shows the hierarchy of diagrams according to UML 2.2



1. **Class** – A class defines the blue print i.e. structure and functions of an object.
2. **Objects** – Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.
3. **Inheritance** – Inheritance is a mechanism by which child classes inherit the properties of their parent classes.
4. **Abstraction** – Mechanism by which implementation details are hidden from user.
5. **Encapsulation** – Binding data together and protecting it from the outer world is referred to as encapsulation.
6. **Polymorphism** – Mechanism by which functions or entities are able to exist in different forms.

Additions in UML 2.0 –

- Software development methodologies like agile have been incorporated and scope of original UML specification has been broadened.
- Originally UML specified 9 diagrams. UML 2.x has increased the number of diagrams from 9 to 13. The four diagrams that were added are : timing diagram, communication diagram, interaction overview diagram and composite structure diagram. UML 2.x renamed statechart diagrams to state machine diagrams.
- UML 2.x added the ability to decompose software system into components and subcomponents.

1. **Class Diagram** – The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes,their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.
2. **Composite Structure Diagram** – We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.
3. **Object Diagram** – An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.
4. **Component Diagram** – Component diagrams are used to represent the how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.
5. **Deployment Diagram** – Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.
6. **Package Diagram** – We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organise UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organise class and use case diagrams.

1. **State Machine Diagrams** – A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams** . These terms are often used interchangeably.So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.
2. **Activity Diagrams** – We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the

execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram.An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.

3. **Use Case Diagrams** – Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.
4. **Sequence Diagram** – A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.
5. **Communication Diagram** – A Communication Diagram(known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams,however, communication diagrams represent objects and links in a free form.
6. **Timing Diagram** – Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.
7. **Interaction Overview Diagram** – An Interaction Overview Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams.

Reference –
Unified Modeling Language – Wikipedia
Unified Modeling Language – IBM

Source

https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction/

Chapter 58

Unified Modeling Language (UML) | Class Diagrams

Unified Modeling Language (UML) | Class Diagrams - GeeksforGeeks

What is UML?
It is the general purpose modeling language used to visualize the system. It is a graphical language that is standard to the software industry for specifying, visualizing, constructing and documenting the artifacts of the software systems, as well as for business modeling.

Benefits of UML:

- Simplifies complex software design, can also implement OOPs like concept which is widely used.
- It reduces thousands of words of explanation in a few graphical diagrams that may reduce time consumption to understand.
- It makes communication more clear and real.
- It helps to acquire the entire system in a view.
- It becomes very much easy for the software programmer to implement the actual demand once they have the clear picture of the problem.

Types of UML: The UML diagrams are divided into two parts: Structural UML diagrams and Behavioral UML diagrams which are listed below:

1. Structural UML diagrams
    - Class diagram
    - Package diagram
    - Object diagram
    - Component diagram
    - Composite structure diagram
    - Deployment diagram
2. Behavioral UML diagrams

- Activity diagram
- Sequence diagram
- Use case diagram
- State diagram
- Communication diagram
- Interaction overview diagram
- Timing diagram

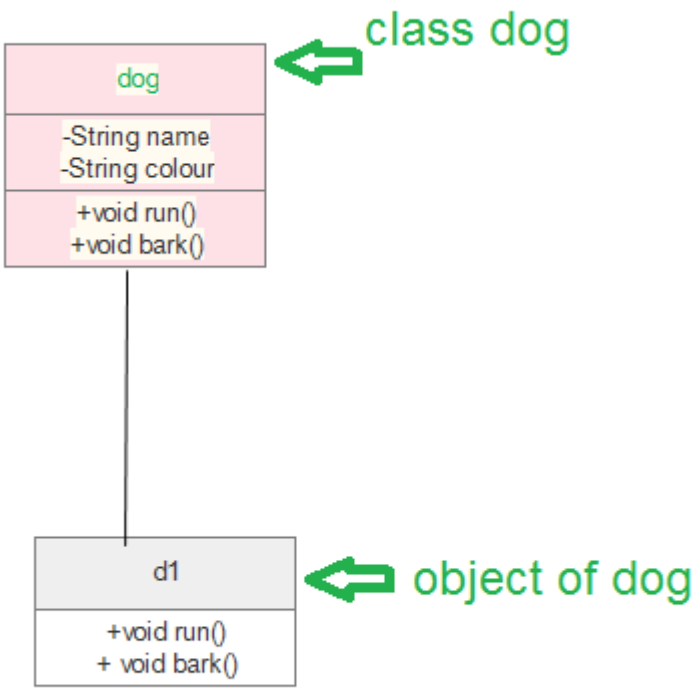
UML class diagrams: Class diagrams are the main building blocks of every object oriented methods. The class diagram can be used to show the classes, relationships, interface, association, and collaboration. UML is standardized in class diagrams. Since classes are the building block of an application that is based on OOPs, so as the class diagram has appropriate structure to represent the classes, inheritance, relationships, and everything that OOPs have in its context. It describes various kinds of objects and the static relationship in between them.
The main purpose to use class diagrams are:

- This is the only UML which can appropriately depict various aspects of OOPs concept.
- Proper design and analysis of application can be faster and efficient.
- It is base for deployment and component diagram.

There are several software available which can be used online and offline to draw these diagrams Like Edraw max, lucid chart etc. There are several points to be kept in focus while drawing the class diagram. These can be said as its syntax:

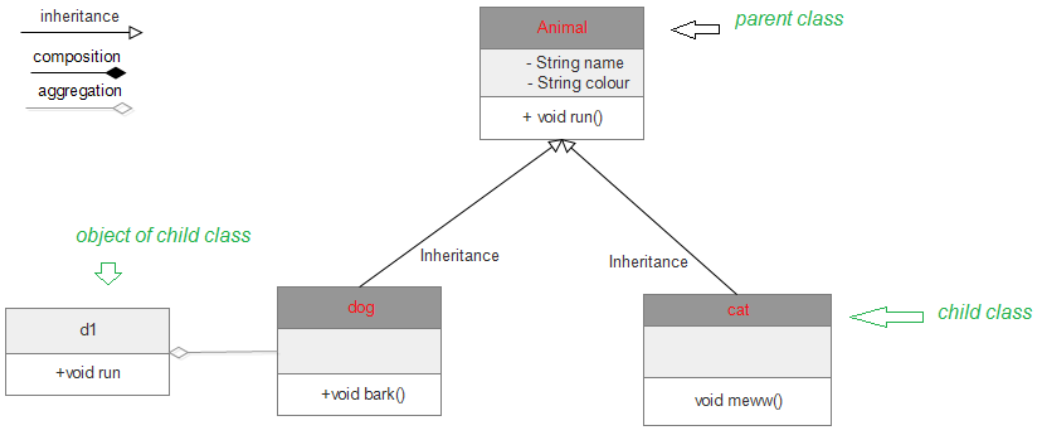
- Each class is represented by a rectangle having a subdivision of three compartments name, attributes and operation.
- There are three types of modifiers which are used to decide the visibility of attributes and operations.
  - + is used for public visibility(for everyone)
  - # is used for protected visibility (for friend and derived)
  - – is used for private visibility (for only me)

Below is the example of Animal class (parent) having two child class as dog and cat both have object d1, c1 inheriting properties of the parent class.




```
 import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        dog d1 = new dog();
        d1.bark();
        d1.run();
        cat c1 = new cat();
        c1.meww();
    }
}

class Animal {
    public void run()
    {
        String name;
        String colour;

        System.out.println("animal is running");
    }
}
```

```
class dog extends Animal {
    public void bark()
    {
        System.out.println("wooh!wooh! dog is barking");
    }
    public void run()
    {
        System.out.println("dog is running");
    }
}

class cat extends Animal {
    public void meww()
    {
        System.out.println("meww! meww!");
    }
}
```



Process to design class diagram: In Edraw max (or any other platform where class diagrams can be drawn) follow the steps:

- Open a blank document in the class diagram section.
- From the library select the class diagram and click on create option.
- Prepare the model of the class in the opened template page.
- After editing according to requirement save it.

There are several diagram components which can be efficiently used while making/editing the model. These are as follows:

- Class { name, attribute, method}
- Objects

- Interface
- Relationships {inheritance, association, generalization}
- Associations {bidirectional, unidirectional}

Class diagrams are one of the most widely used diagrams in the fields of software engineering as well as businesses modelling.

Source

https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/

Chapter 59

Unified Modeling Language (UML) | Object Diagrams

Unified Modeling Language (UML) | Object Diagrams - GeeksforGeeks

An **Object Diagram** can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behavior of the system at a particular instant. Object diagrams are vital to portray and understand functional requirements of a system.
In other words, "An object diagram in the Unified Modeling Language (UML), is a diagram that shows a **complete or partial view** of the structure of a modeled system **at a specific time.**"

Difference between an Object and a Class Diagram –
An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.

What is a classifier?

In UML a classifier refers to a group of elements that have some common features like methods, attributes and operations. A classifier can be thought of as an abstract metaclass which draws a boundary for a group of instances having common static and dynamic features. For example, we refer a class, an object, a component, or a deployment node as classifiers in UML since they define a common set of properties.

An Object Diagram is a structural diagram which uses notation similar to that of class diagrams. We are able to design object diagrams by instantiating classifiers.

Object Diagrams use **real world examples** to depict the nature and structure of the system at a particular **point in time**. Since we are able to use data available within objects, Object diagrams provide a **clearer view** of the relationships that exist **between objects**.

Figure – a class and its corresponding object

1. **Objects or Instance specifications** – When we instantiate a classifier in a system, the object we create represents an entity which exists in the system.We can represent the changes in object over time by creating multiple instance specifications. We use a rectangle to represent an object in an Object Diagram. An object is generally linked to other objects in an object diagram.

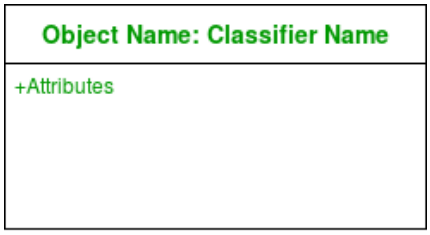


Figure – notation for an Object

For example – In the figure below, two objects of class Student are linked to an object of class College.

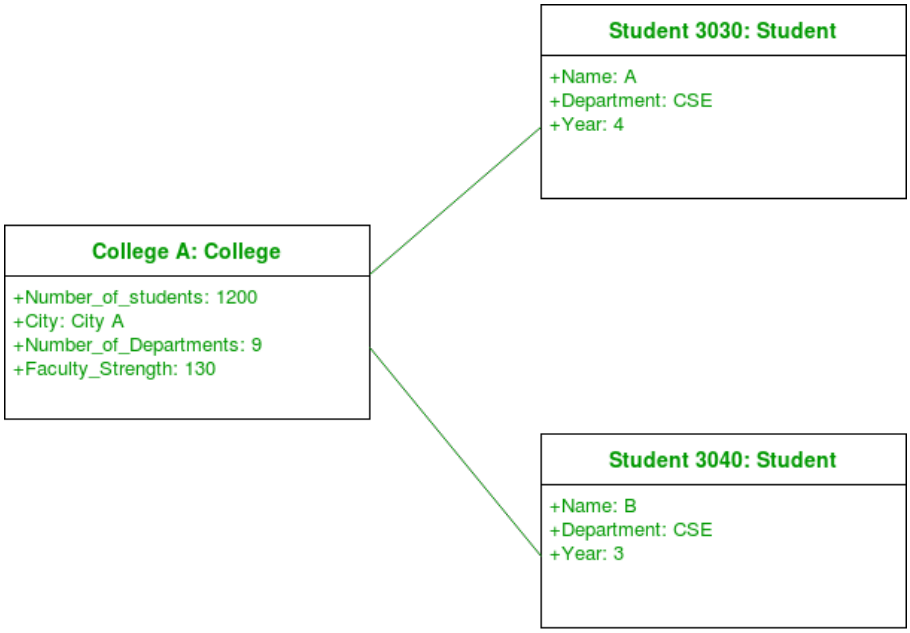


Figure – an object diagram using a link and 3 objects

2. **Links** – We use a link to represent a relationship between two objects.

Figure – notation for a link

We represent the number of participants on the link for each end of the link. We use the term association for a relationship between two classifiers. The term link is used to specify a relationship between two instance specifications or objects. We use a solid line to represent a link between two objects.

Notation	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
	Zero or more
1..*	One or more
7	Seven only
0..2	Zero or two
4..7	Four to seven

3. **Dependency Relationships** – We use a dependency relationship to show when one element depends on another element.

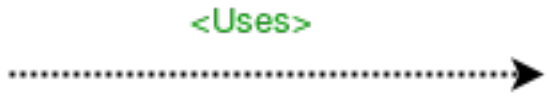


Figure – notation for dependency relationship

Class diagrams, component diagrams, deployment and object diagrams use dependency relationships. A dependency is used to depict the relationship between dependent and independent entities in the system.Any change in the definition or structure of one element may cause changes to the other. This is a unidirectional kind of relationship between two objects.
Dependency relationships are of various types specified with keywords (sometimes within angular brackets").

Abstraction,Binding, Realization, Substitution and Usage are the types of dependency relationships used in UML.
For example – In the figure below, an object of Player class is dependent (or uses) an object of Bat class.

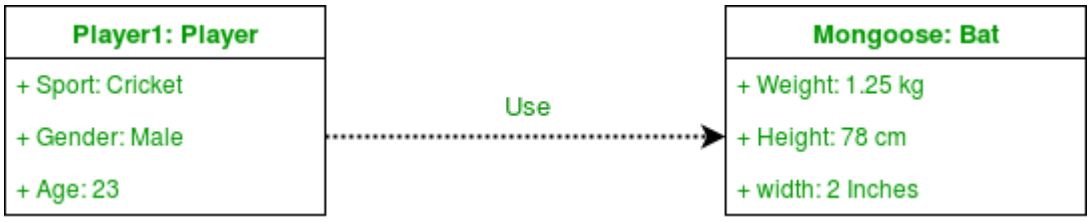


Figure – an object diagram using a dependency relationship

4. **Association** – Association is a reference relationship between two objects (or classes).

Figure – notation for association

Whenever an object uses another it is called an association.We use association when one object references members of the other object. Association can be uni-directional or bi-directional. We use an arrow to represent association.
For example – The object of Order class is associated with an object of Customer class.

Figure – an object diagram using association

5. **Aggregation** – Aggregation represents a "has a" relationship.

Figure – notation for aggregation

Aggregation is a specific form of association.on relationship; aggregation is more specific than ordinary association. It is an association that represents a part-whole or part-of relationship. It is a kind of parent -child relationship however it isn't inheritance. Aggregation occurs when the lifecycle of the contained objects does not strongly depend on the lifecycle of container objects.

Figure – an object diagram using aggregation

For example – A library has an aggregation relationship with books. Library has books or books are a part of library. The existence of books is independent of the existence of the library.While implementing, there isn't a lot of difference between aggregation and association. We use a hollow diamond on the containing object with a line which joins it to the contained object.

6. **Composition** – Composition is a type of association where the child cannot exist independent of the other.

Figure – notation for composition

Composition is also a special type of association. It is also a kind of parent child relationship but it is not inheritance. Consider the example of a boy Gurkaran: Gurkaran is composed of legs and arms. Here Gurkaran has a composition relationship with his

legs and arms. Here legs and arms cant exist without the existence of their parent object. So whenever independent existence of the child is not possible we use a composition relationship. We use a filled diamond on the containing object with a line which joins it to the contained object.

Figure – an object diagram using composition

For Example – In the figure below, consider the object Bank1. Here an account cannot exist without the existence of a bank.

Figure – a bank is composed of accounts

Association and dependency are often confused in their usage. A source of confusion was the use of transient links in UML 1. Meta-models are now handled differently in UML 2 and the issue has been resolved.

There are a large number of dependencies in a system. We only represent the ones which are essential to convey for understanding the system. We need to understand that every association implies a dependency itself. We , however, prefer not to draw it separately. An association implies a dependency similar to a way in which generalization does.

1. Draw all the necessary class diagrams for the system.
2. Identify the crucial points in time where a system snapshot is needed.
3. Identify the objects which cover crucial functionality of the system.
4. Identify the relationship between objects drawn.

- Model the static design(similar to class diagrams) or structure of a system using prototypical instances and real data.
- Helps us to understand the functionalities that the system should deliver to the users.
- Understand relationships between objects.
- Visualise, document, construct and design a static frame showing instances of objects and their relationships in the dynamic story of life of a system.
- Verify the class diagrams for completeness and accuracy by using Object Diagrams as specific test cases.
- Discover facts and dependencies between specific instances and depicting specific examples of classifiers.

Source

https://www.geeksforgeeks.org/unified-modeling-language-uml-object-diagrams/

Chapter 60

Unified Modeling Language (UML) | Sequence Diagrams

Unified Modeling Language (UML) | Sequence Diagrams - GeeksforGeeks

In this post we discuss Sequence Diagrams. **Unified Modelling Language (UML)** is a modeling language in the field of software engineering which aims to set standard ways to visualize the design of a system. UML guides the creation of multiple types of diagrams such as interaction , structure and behaviour diagrams.
A **sequence diagram** is the most commonly used **interaction** diagram.

Interaction diagram –
An interaction diagram is used to show the **interactive behavior** of a system. Since visualizing the interactions in a system can be a cumbersome task, we use different types of interaction diagrams to capture various features and aspects of interaction in a system.

Sequence Diagrams –
A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

1. **Actors** – An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.

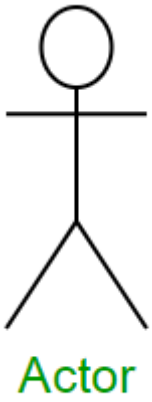


Figure – notation symbol for actor

We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.
For example – Here the user in seat reservation system is shown as an actor where it exists outside the system and is not a part of the system.

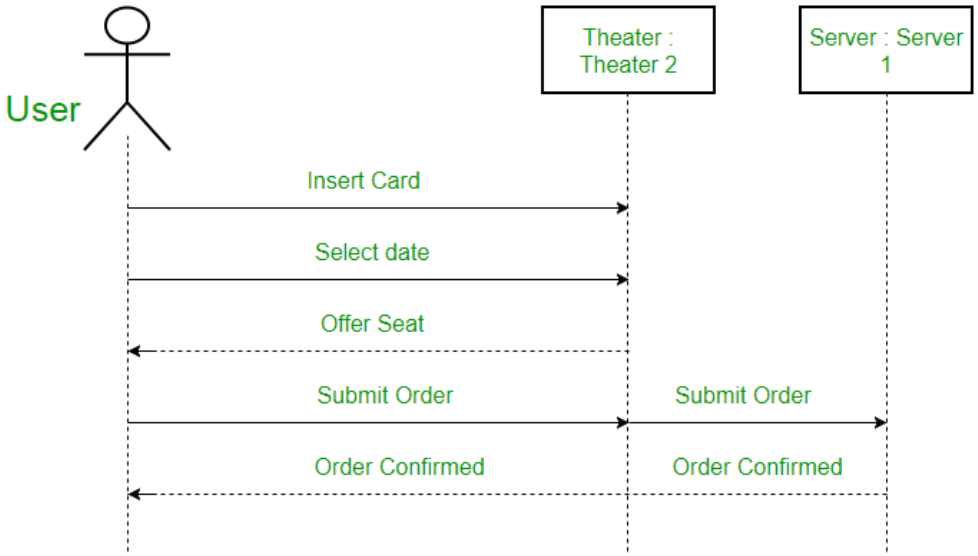


Figure – an actor interacting with a seat reservation system

2. **Lifelines** – A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram. The

standard in UML for naming a lifeline follows the following format – Instance Name : Class Name

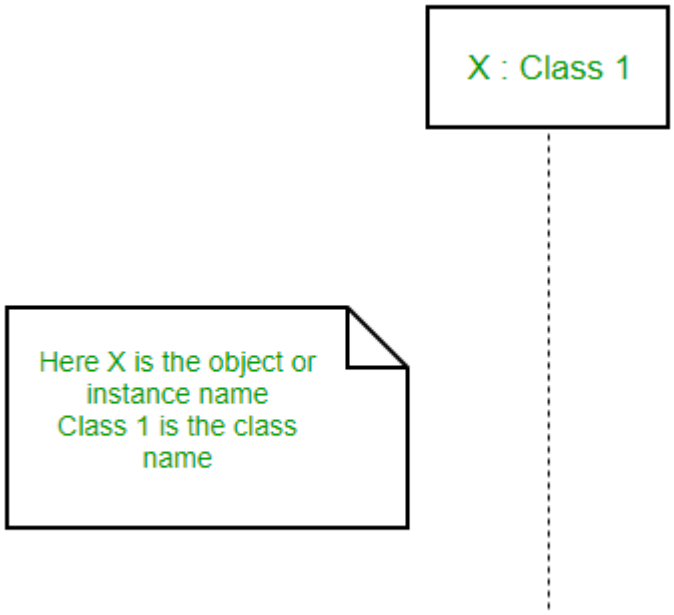


Figure – lifeline

We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the stem) as shown above. If we want to model an unnamed instance, we follow the same pattern except now the portion of lifeline's name is left blank.

Difference between a lifeline and an actor – A lifeline always portrays an object internal to the system whereas actors are used to depict objects external to the system. The following is an example of a sequence diagram:

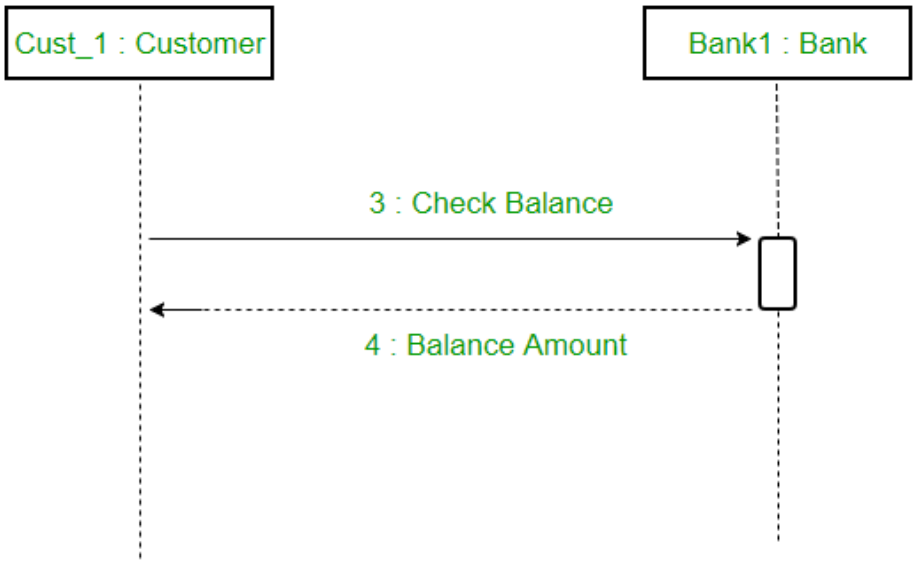


Figure – a sequence diagram

3. **Messages** – Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline. We represent messages using arrows. Lifelines and messages form the core of a sequence diagram.
   Messages can be broadly classified into the following **categories** :

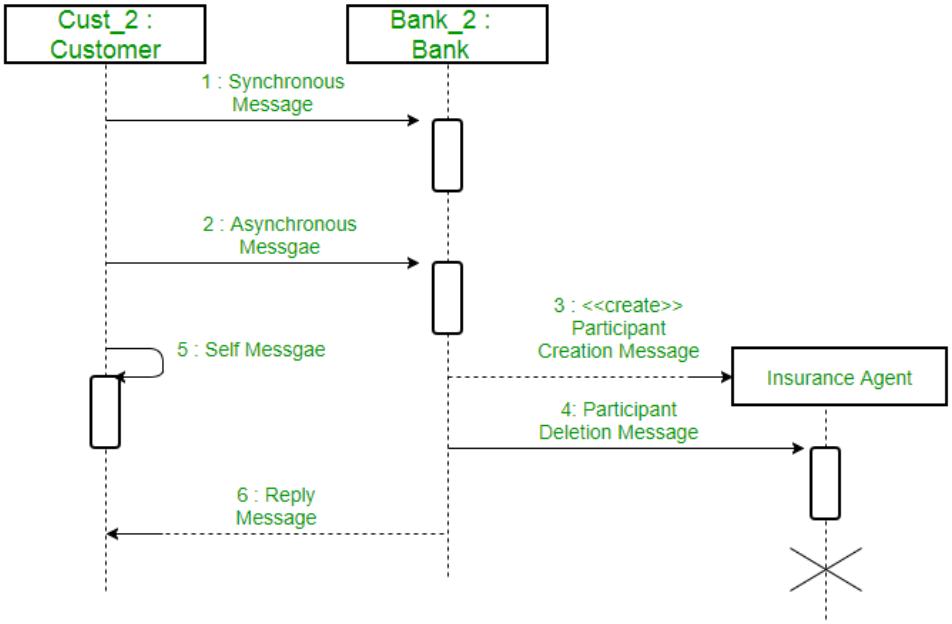


Figure – a sequence diagram with different types of messages

- **Synchronous messages** – A synchronous message waits for a reply before the interaction can move forward. The sender waits until the receiver has completed the processing of the message. The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message. A large number of calls in object oriented programming are synchronous. We use a solid arrow head to represent a synchronous message.

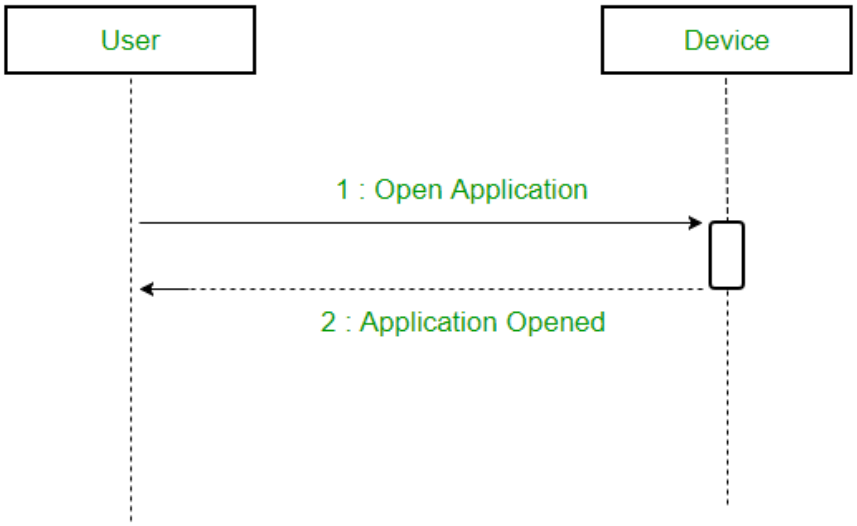
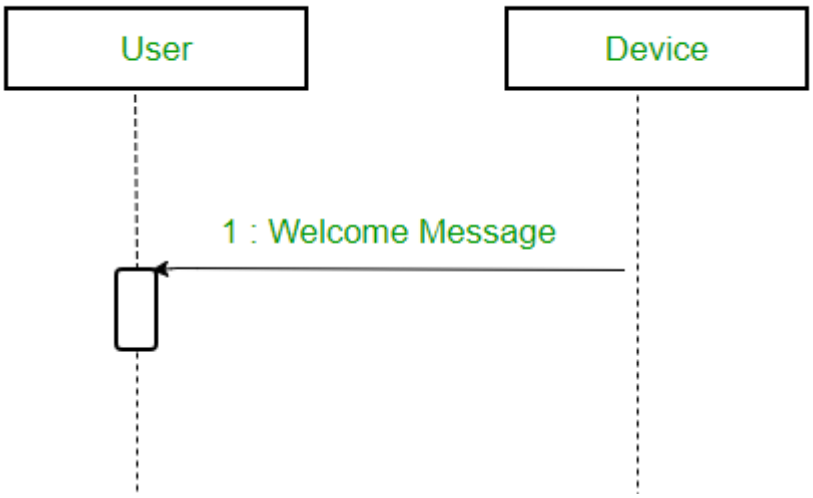


Figure – a sequence diagram using a synchronous message

- **Asynchronous Messages** – An asynchronous message does not wait for a reply from the receiver. The interaction moves forward irrespective of the receiver processing the previous message or not. We use a lined arrow head to represent an asynchronous message.



- **Create message** – We use a Create message to instantiate a new object in the sequence diagram. There are situations when a particular message call requires the creation of an object. It is represented with a dotted arrow and create word labelled on it to specify that it is the create Message symbol.
  For example – The creation of a new order on a e-commerce website would require a new object of Order class to be created.

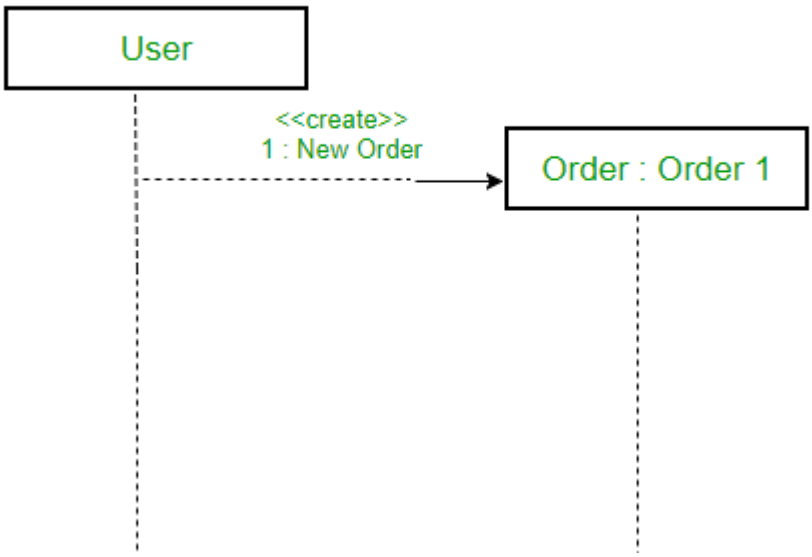


Figure – a situation where create message is used

- **Delete Message** – We use a Delete Message to delete an object. When an object is deallocated memory or is destroyed within the system we use the Delete Message symbol. It destroys the occurrence of the object in the system.It is represented by an arrow terminating with a x.

For example – In the scenario below when the order is received by the user, the object of order class can be destroyed.

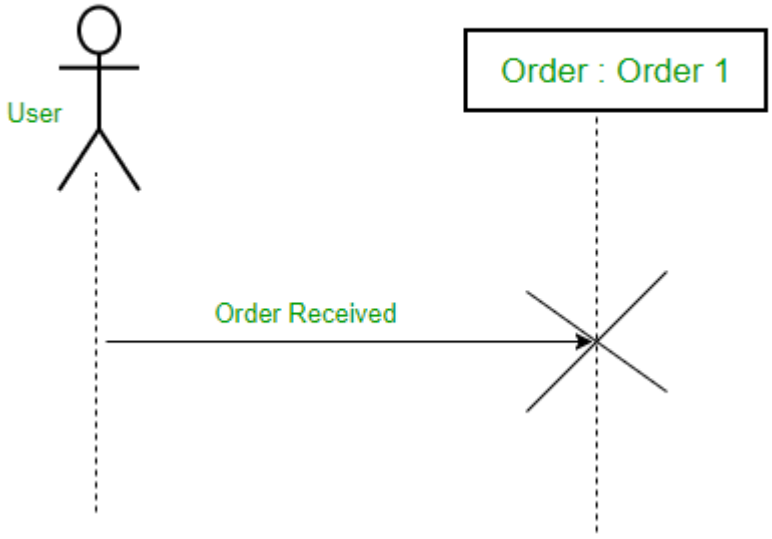


Figure – a scenario where delete message is used

- **Self Message** – Certain scenarios might arise where the object needs to send a message to itself. Such messages are called Self Messages and are represented with a U shaped arrow.

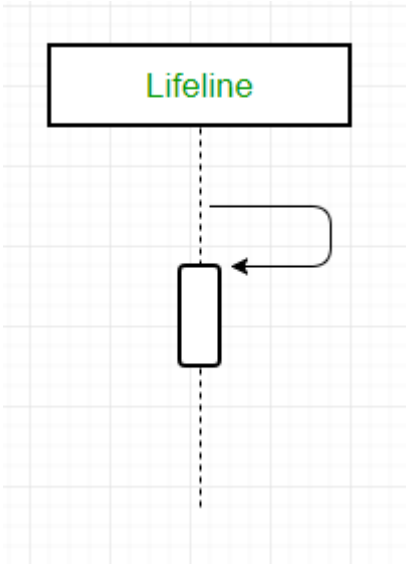


Figure – self message

For example – Consider a scenario where the device wants to access its webcam.

Such a scenario is represented using a self message.

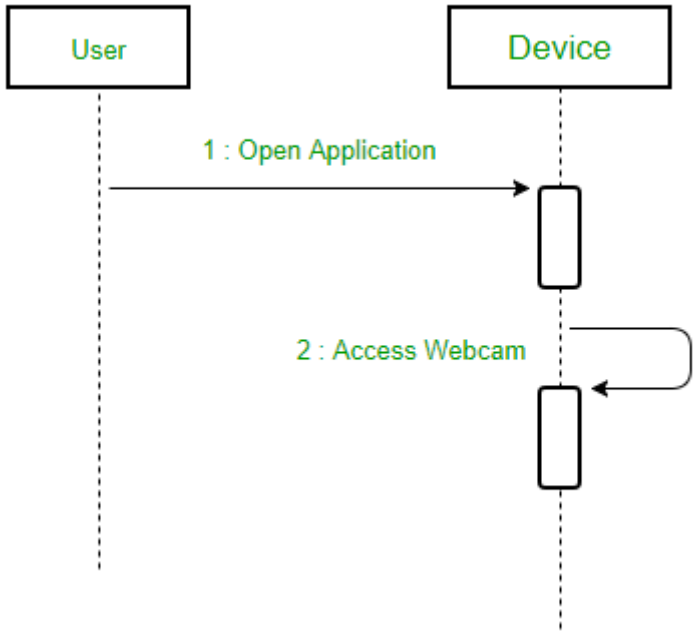


Figure – a scenario where a self message is used

- **Reply Message** – Reply messages are used to show the message being sent from the receiver to the sender. We represent a return/reply message using an open arrowhead with a dotted line. The interaction moves forward only when a reply message is sent by the receiver.

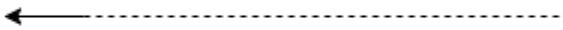


Figure – reply message

For example – Consider the scenario where the device requests a photo from the user. Here the message which shows the photo being sent is a reply message.

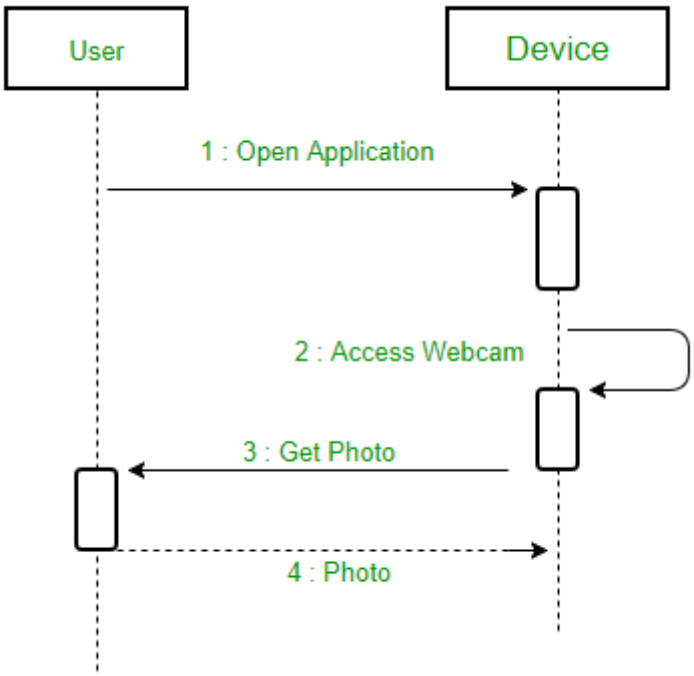


Figure – a scenario where a reply message is used

- **Found Message** – A Found message is used to represent a scenario where an unknown source sends the message. It is represented using an arrow directed towards a lifeline from an end point. For example: Consider the scenario of a hardware failure.

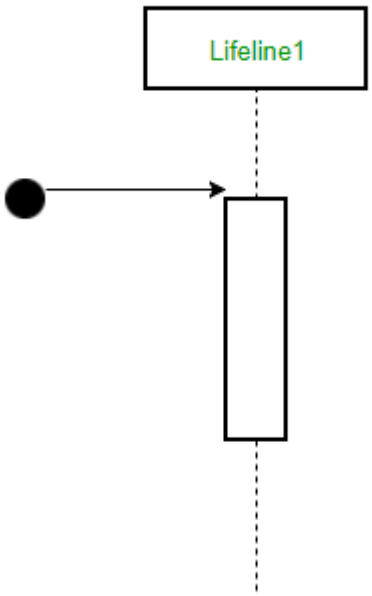


Figure – found message

It can be due to multiple reasons and we are not certain as to what caused the hardware failure.

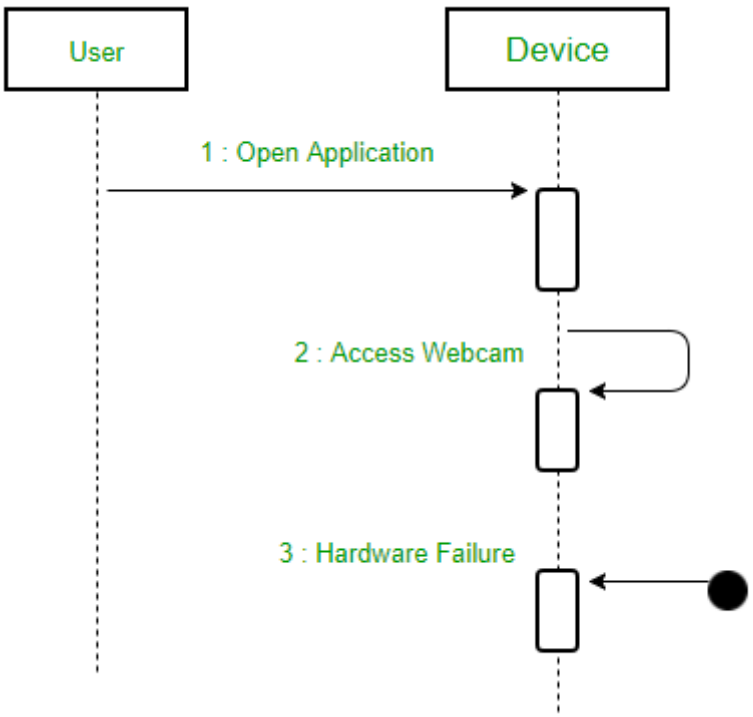


Figure – a scenario where found message is used

- **Lost Message** – A Lost message is used to represent a scenario where the recipient is not known to the system. It is represented using an arrow directed towards an end point from a lifeline. For example: Consider a scenario where a warning is generated.

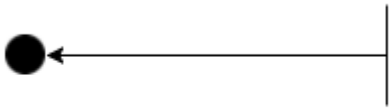


Figure – lost message

The warning might be generated for the user or other software/object that the lifeline is interracting with. Since the destination is not known before hand, we use the Lost Message symbol.

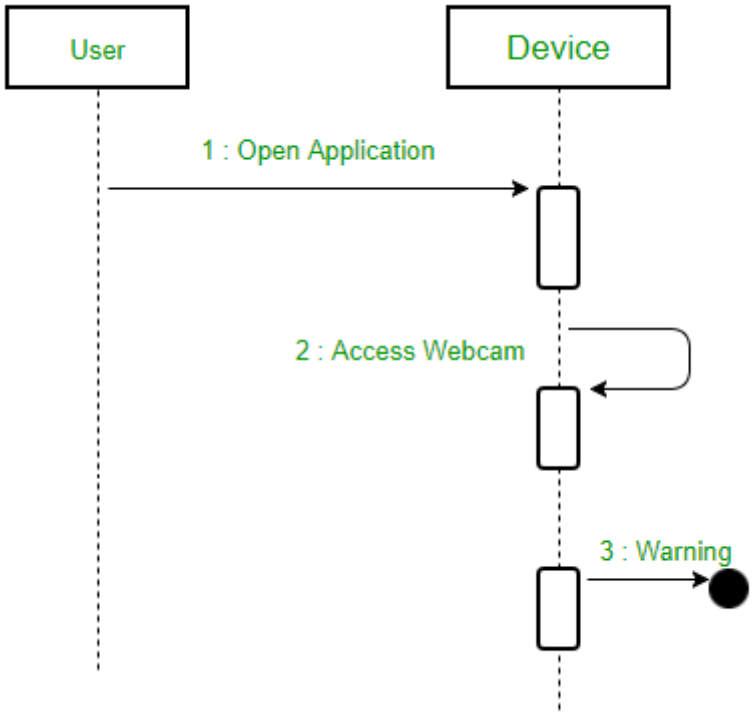


Figure – a scenario where lost message is used

4. **Guards** – To model conditions we use guards in UML. They are used when we need to restrict the flow of messages on the pretext of a condition being met. Guards play an important role in letting software developers know the constraints attached to a system or a particular process.

   For example: In order to be able to withdraw cash, having a balance greater than zero is a condition that must be met as shown below.

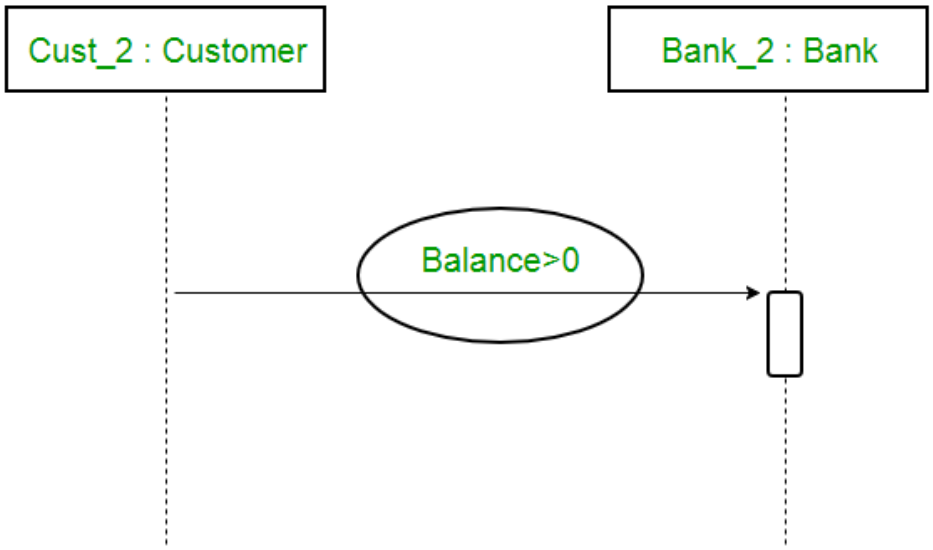


Figure – sequence diagram using a guard

A sequence diagram for an emotion based music player –

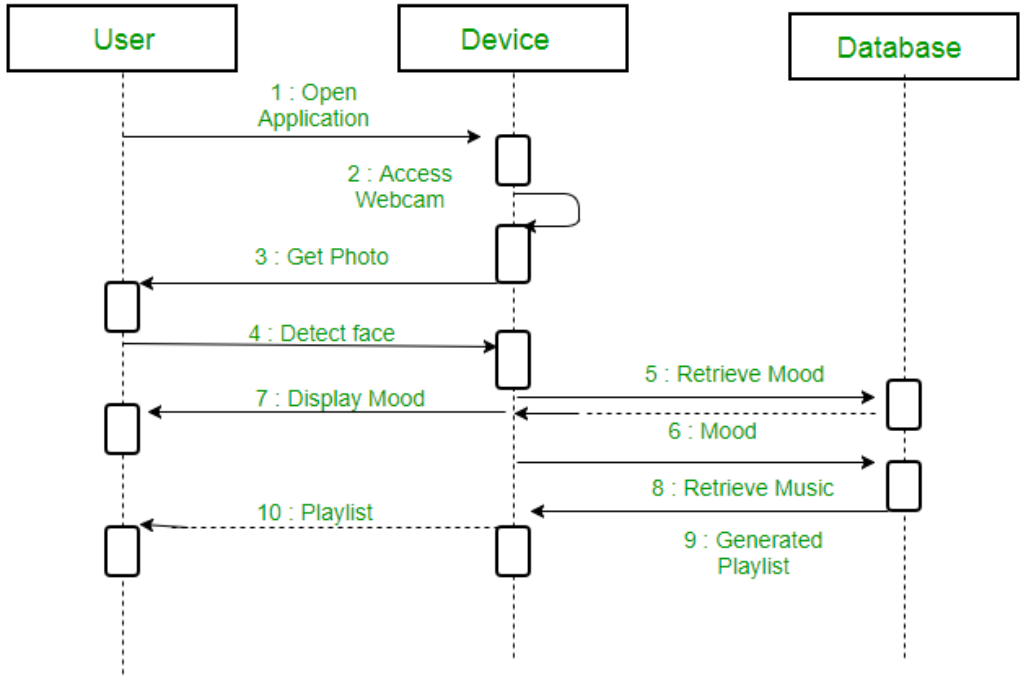


Figure – a sequence diagram for an emotion based music player

The above sequence diagram depicts the sequence diagram for an emotion based music player:

1. Firstly the application is opened by the user.
2. The device then gets access to the web cam.
3. The webcam captures the image of the user.
4. The device uses algorithms to detect the face and predict the mood.
5. It then requests database for dictionary of possible moods.
6. The mood is retrieved from the database.
7. The mood is displayed to the user.
8. The music is requested from the database.
9. The playlist is generated and finally shown to the user.

- Used to model and visualise the logic behind a sophisticated function, operation or procedure.
- They are also used to show details of UML use case diagrams.
- Used to understand the detailed functionality of current or future systems.
- Visualise how messages and tasks move between objects or components in a system.

References –
The sequence diagram – IBM
Sequence Diagram – sparxsystems

Source

https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/

Chapter 61

Unified Modeling Language (UML) | State Diagrams

Unified Modeling Language (UML) | State Diagrams - GeeksforGeeks

A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time. It's a **behavioral** diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and **Statechart Diagrams**. These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli. We can say that each and every class has a state but we don't model every class using State diagrams. We prefer to model the states with three or more states.

Uses of statechart diagram –

- We use it to state the events responsible for change in state (we do not show what processes cause those events).
- We use it to model the dynamic behavior of the system .
- To understand the reaction of objects/classes to internal or external stimuli.

Firstly let us understand what are **Behavior diagrams**? There are two types of diagrams in UML :

1. **Structure Diagrams** – Used to model the static structure of a system, for example- class diagram, package diagram, object diagram, deployment diagram etc.
2. **Behavior diagram** – Used to model the dynamic change in the system over time. They are used to model and construct the functionality of a system. So, a behavior diagram simply guides us through the functionality of the system using Use case diagrams, Interaction diagrams, Activity diagrams and State diagrams.

Difference between state diagram and flowchart –
The basic purpose of a **state diagram** is to portray various changes in state of the class and

not the processes or commands causing the changes. However, a **flowchart** on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.

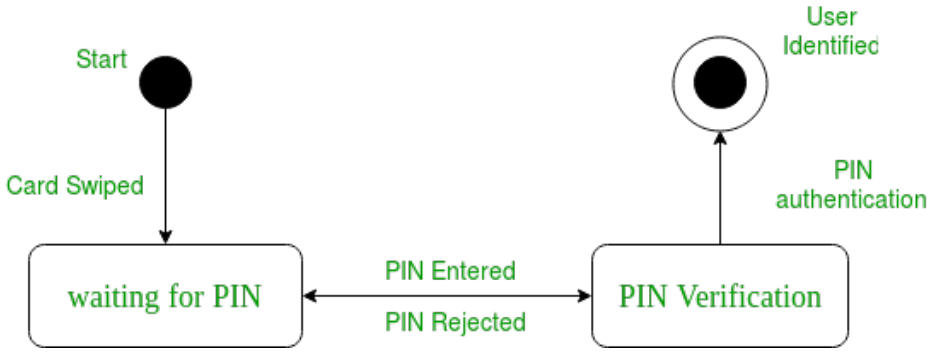


Figure – a state diagram for user verification

The state diagram above shows the different states in which the verification sub-system or class exist for a particular system.

1. **Initial state** – We use a black filled circle represent the initial state of a System or a class.

   Figure – initial state notation

2. **Transition** – We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.

   Figure – transition

3. **State** – We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time.

   Figure – state notation

4. **Fork** – We use a rounded solid rectangular bar to represent a Fork notation with incomig arrow from the parent state and outgoing arrows towards the newly created

states. We use the fork notation to represent a state splitting into two or more concurrent states.

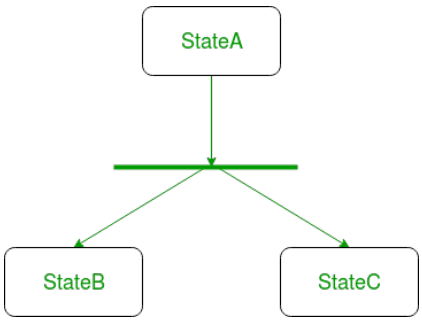


Figure – a diagram using the fork notation

5. **Join** – We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.

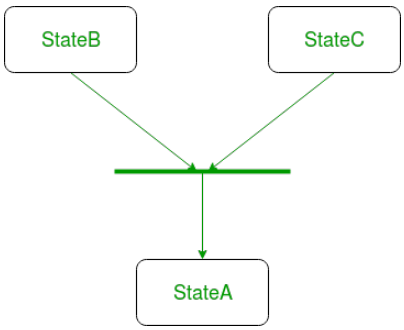


Figure – a diagram using join notation

6. **Self transition** – We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.

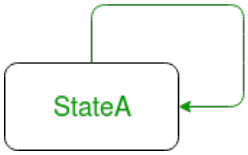


Figure – self transition notation

7. **Composite state** – We use a rounded rectangle to represent a composite state also.We represent a state with internal activities using a composite state.

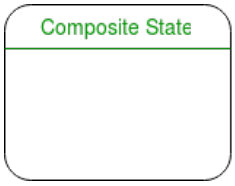


Figure – a state with internal activities

8. **Final state** – We use a filled circle within a circle notation to represent the final state in a state machine diagram.

Figure – final state notation

1. Identify the initial state and the final terminating states.
2. Identify the possible states in which the object can exist (boundary values corresponding to different attributes guide us in identifying different states).
3. Label the events which trigger these transitions.

Example – state diagram for an online order –

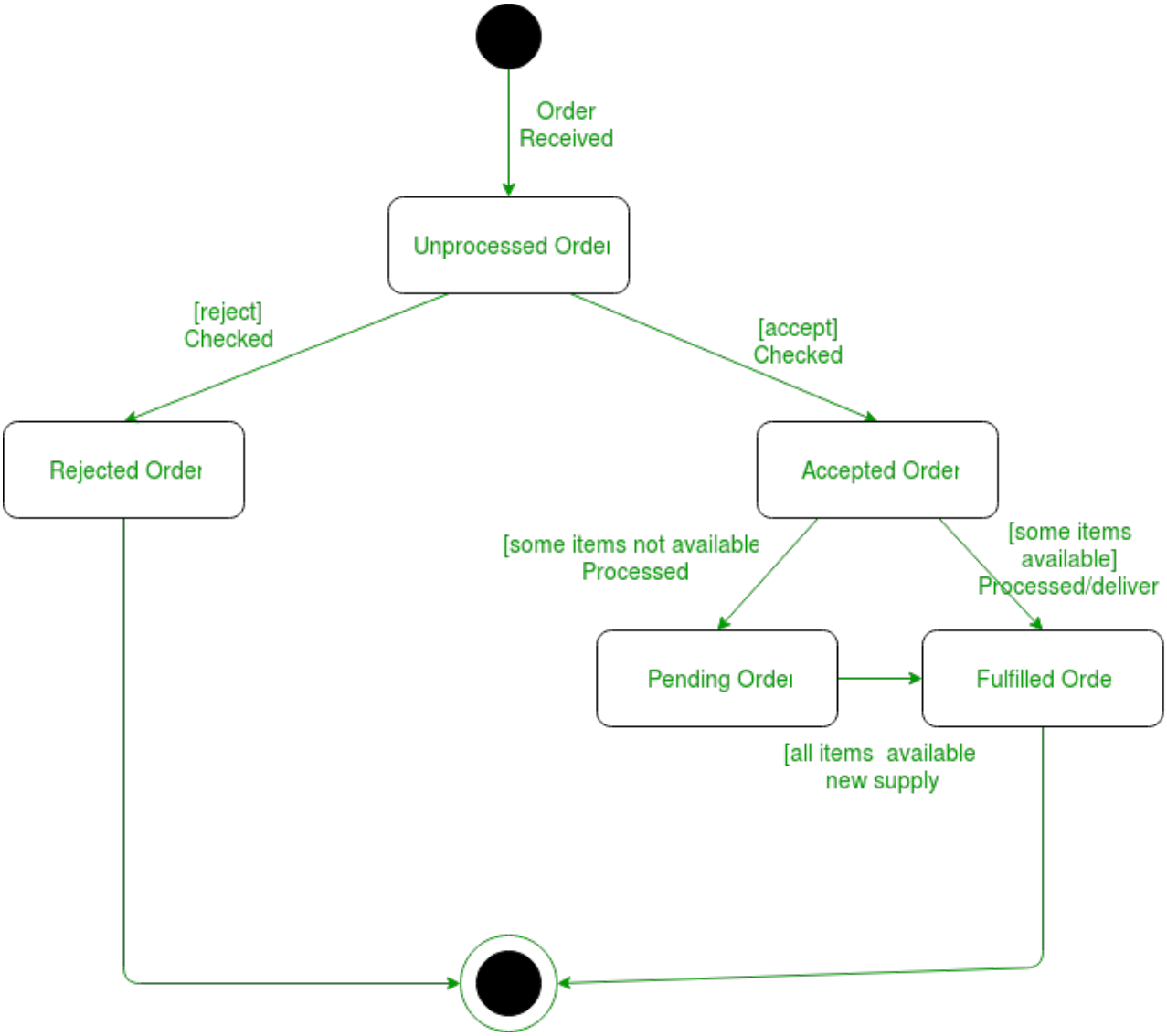


Figure – state diagram for an online order

The UMl diagrams we draw depend on the system we aim to represent. Here is just an example of how an online ordering system might look like :

1. On the event of an order being received, we transit from our initial state to Unprocessed order state.
2. The unprocessed order is then checked.
3. If the order is rejected, we transit to the Rejected Order state.
4. If the order is accepted and we have the items available we transit to the fulfilled order state.
5. However if the items are not available we transit to the Pending Order state.

6. After the order is fulfilled, we transit to the final state. In this example, we merge the two states i.e. Fulfilled order and Rejected order into one final state.

Note – Here we could have also treated fulfilled order and rejected order as final states separately.

Reference –
State Diagram – IBM

Source

https://www.geeksforgeeks.org/unified-modeling-language-uml-state-diagrams/

Chapter 62

Visitor design pattern

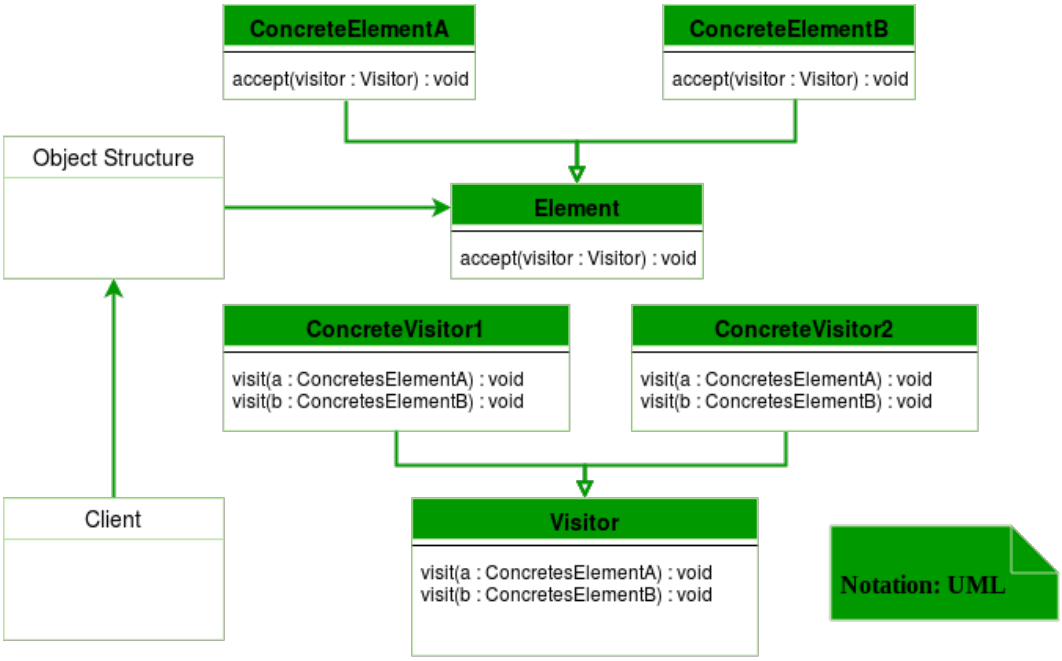
Visitor design pattern - GeeksforGeeks

Visitor design pattern is one of the behavioral design pattern. It is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

The visitor pattern consists of two parts:

- a method called **Visit()** which is implemented by the visitor and is called for every element in the data structure
- visitable classes providing **Accept()** methods that accept a visitor

UML Diagram Visitor design pattern



Design components

- **Client :** The Client class is a consumer of the classes of the visitor design pattern. It has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing.
- **Visitor :** This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.
- **ConcreteVisitor :** For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations.
- **Visitable :** is an interface which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object.
- **ConcreteVisitable :** Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

Let's see an example of Visitor design pattern.

```
 interface ItemElement
{
    public int accept(ShoppingCartVisitor visitor);
}

class Book implements ItemElement
{
    private int price;
    private String isbnNumber;
```

```
    public Book(int cost, String isbn)
    {
        this.price=cost;
        this.isbnNumber=isbn;
    }

    public int getPrice()
    {
        return price;
    }

    public String getIsbnNumber()
    {
        return isbnNumber;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }

}

class Fruit implements ItemElement
{
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm)
    {
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg()
    {
        return pricePerKg;
    }

    public int getWeight()
    {
        return weight;
    }
```

```
    public String getName()
    {
        return this.name;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }

}

interface ShoppingCartVisitor
{

    int visit(Book book);
    int visit(Fruit fruit);
}

class ShoppingCartVisitorImpl implements ShoppingCartVisitor
{

    @Override
    public int visit(Book book)
    {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50)
        {
            cost = book.getPrice()-5;
        }
        else
            cost = book.getPrice();

        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);
        return cost;
    }

    @Override
    public int visit(Fruit fruit)
    {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = "+cost);
        return cost;
    }

}
```

```
class ShoppingCartClient
{

    public static void main(String[] args)
    {
        ItemElement[] items = new ItemElement[]{new Book(20, "1234"),new Book(100, "5678"),
                new Fruit(10, 2, "Banana"), new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }

    private static int calculatePrice(ItemElement[] items)
    {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items)
        {
            sum = sum + item.accept(visitor);
        }
        return sum;
    }

}
```

Output:

```
Book ISBN::1234 cost =20
Book ISBN::5678 cost =95
Banana cost = 20
Apple cost = 25
Total Cost = 160
```

Here, in the implementation if accept() method in all the items are same but it can be different, for example there can be logic to check if item is free then don't call the visit() method at all.

Advantages :

- If the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.
- Adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

Disadvantages :

- We should know the return type of visit() methods at the time of designing otherwise we will have to change the interface and all of its implementations.
- If there are too many implementations of visitor interface, it makes it hard to extend.

Source

https://www.geeksforgeeks.org/visitor-design-pattern/