Assignment 1: Factory Design Pattern

What is Factory Design Pattern?

Factory Method is a creational design pattern that provides an **interface** for creating objects in a superclass, but allows subclasses to **alter** the type of objects that will be created. This pattern is particularly useful when a class doesn't know what sub-classes will be required to create an object, or when a class wants to delegate the responsibility of **object creation** to its subclasses.

Context: *Car*

In the context of creating different types of cars, the factory design pattern can be used to create different cars without the client code needing to know the specific class of the object being created.

For example, let's say we have a car factory class that is responsible for creating different cars such as Alcazar, Swift, Creta, etc. Each car has its own class and implementation but we can create them through a factory method in the factory class. The client code can use this factory method to create objects of different car without having to know the specific class of the object being created.

Parent Class: Car

Child Class 1: Alcazar

Child Class 2: Swift

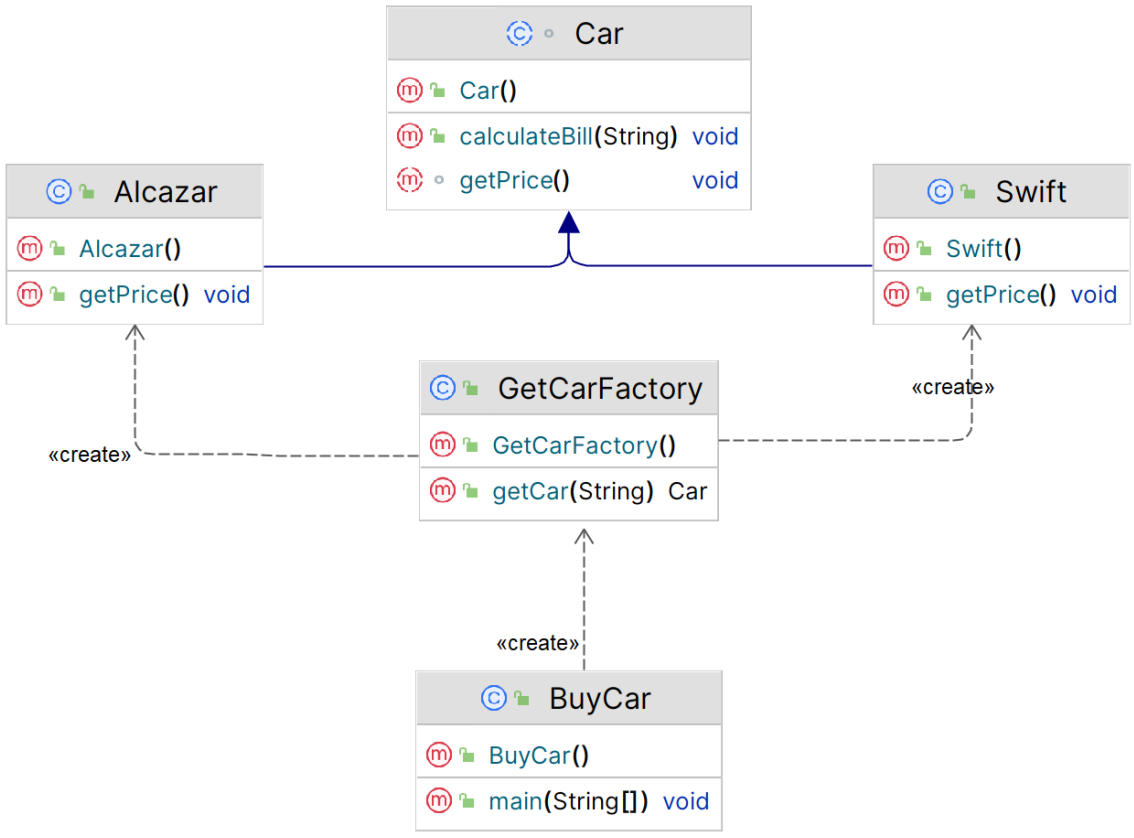


Figure: Inheritance Diagram

Implementation (Code)

```
abstract class Car {
    protected double rate;
    public String color;
    abstract void getPrice();
    public void calculateBill(String color){
        double colorValue;
        colorValue = switch (color) {
            case ("Brown") -> 40000;
            case ("White") -> 60000;
            case ("Red") -> 75000;
            case ("Black") -> 90000;
            default -> 38000;
        };
        System.out.println(rate + colorValue);
    }
}



public class Alcazar extends Car {
    public void getPrice() {
        rate = 1500000;
    }
}



public class Swift extends Car {
    public void getPrice() {
        rate = 500000;
    }
}



public class GetCarFactory {
    public Car getCar(String carName){
        if (carName == null){
            return null;
        }
        if (carName.equalsIgnoreCase("Alcazar")){
            return new Alcazar();
        }
        else if (carName.equalsIgnoreCase("Swift")){
            return new Swift();
        }
        return null;
    }
}
```

```
import java.io.*;
public class BuyCar {
    public static void main(String[] args) throws IOException {
        GetCarFactory carFactory = new GetCarFactory();

        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

        // Name
        System.out.print("Which car you want to buy [Alcazar, Swift]: ");
        String carName =br.readLine();

        System.out.println("Ok! \nPlease Enter the Specifications you need: ");

        // Color
        System.out.print("Color [ Black,Red,White,Brown ] : ");
        String color = br.readLine();

        Car c = carFactory.getCar(carName);

        System.out.print("Your Bill amount for " + carName + " is ");
        c.getPrice();
        c.calculateBill(color);

    }
}
```

Output

```
Which car you want to buy [Alcazar, Swift]: Alcazar
Ok!
Please Enter the Specifications you need :
Color [ Black,Red,White,Brown ] : Black
Your Bill amount for Alcazar is 1590000.0
```

Applicability

1. Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.
2. Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.
3. Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

Assignment 2: Builder Design Pattern

What is Builder Design Pattern?

Builder is a creational design pattern that lets you **construct complex objects** step by step. The pattern allows you to produce different types and representations of an object using the **same construction code**.

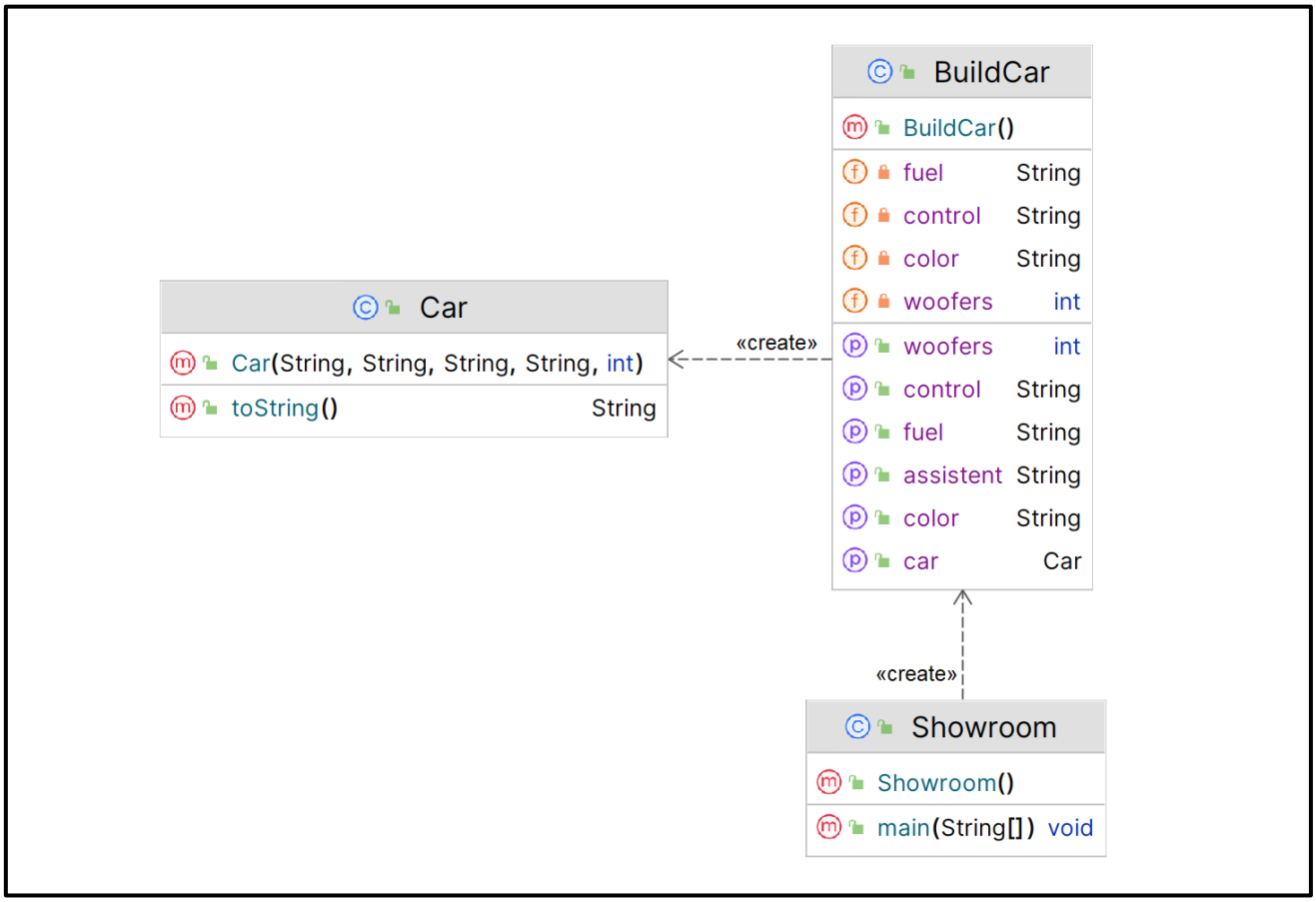
Intend

- ✓ Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- ✓ Parse a complex representation, create one of several targets.

Context: *Car*

In the context of a car, the builder design pattern can be used to create different cars with different customizations using the same construction process. The builder class would have methods for setting different car properties, such as the **Colour, Type of fuel, Transmission Control, Infotainment system, No. of woofers**. The client code would then use these methods to create the desired car object.

Structure (Class Diagram)



Implementation (Code)

Car.java

```
public class Car {
    private String color;
    private String fuel;
    private String control;
    private String assistant;
    private int woofers;


    public Car(String color, String fuel, String control, String assistant, int woofers) {
        super();
        this.color = color;
        this.fuel = fuel;
        this.control = control;
        this.assistant = assistant;
        this.woofers = woofers;
    }


    public String toString() {
        return "Car [Color: " + color + ", Fuel: " + fuel + ", Control: " + control + ", Assistant: " +
assistant + ", Woofers: " + woofers + "]";
    }
}
```

Showroom.java

```
public class Showroom {
    public static void main(String[] args) {
        Car c = new BuildCar().setColor("Red").setFuel("Electric").setControl("Automatic")
                .setAssistent("Android Auto").setWoofers(8).getCar();
        System.out.println(c);
    }
}
```

BuildCar.java

```
public class BuildCar {
    private String color;
    private String fuel;
    private String control;
    private String assistant;
    private int woofers;


    public BuildCar setColor(String color){
        this.color = color;
        return this;
    }
    public BuildCar setFuel(String fuel){
        this.fuel = fuel;
        return this;
    }
    public BuildCar setControl(String control) {
        this.control = control;
        return this;
    }
    public BuildCar setAssistent(String assistant){
        this.assistant = assistant;
        return this;
    }
    public BuildCar setWoofers(int woofers) {
        this.woofers = woofers;
        return this;
    }


    public Car getCar(){
        return new Car(color, fuel, control, assistant, woofers);
    }
}
```

Output

```
Car [Color: Red, Fuel: Electric, Control: Automatic,
 Assistant: Android Auto, Woofers: 8]
```

Applicability

1. Use the Builder pattern to get rid of a ***telescoping constructor***.
2. Use the Builder pattern when you want your code to be able to create **different representations** of some product (for example sports and SUV car).
3. Use the Builder to construct **Complex objects**.

Assignment 3: Prototype Design Pattern

What is Prototype Design Pattern?

Prototype is a **creational design pattern** that lets you copy existing objects (**prototypes**) without making your code dependent on their classes.

Intend

- ✓ Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- ✓ Co-opt one instance of a class for use as a breeder of all future instances.
- ✓ The new operator considered harmful.

Context: *Car*

In the context of car manufacturing, the prototype design pattern can be used to create new car models by copying and modifying existing car prototypes. This allows for a more efficient and cost-effective way of creating new cars, as the process of creating a new car from scratch can be time-consuming and costly.

For example, a car manufacturer may have a prototype of a sports car that they have been using to test different design features. They can then use this prototype as a starting point to create a new car model by making changes to the design and components of the prototype. This allows the manufacturer to quickly create new car models without having to go through the entire process of designing and building a car from scratch.

Problem

Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to **create a new object** of the same class. Then you have to **go through all the fields** of the original object and **copy their values** over to the new object.

Nice! But there's a catch. Not all objects can be copied that way because some of the **object's fields may be private** and **not visible from outside** of the object itself.

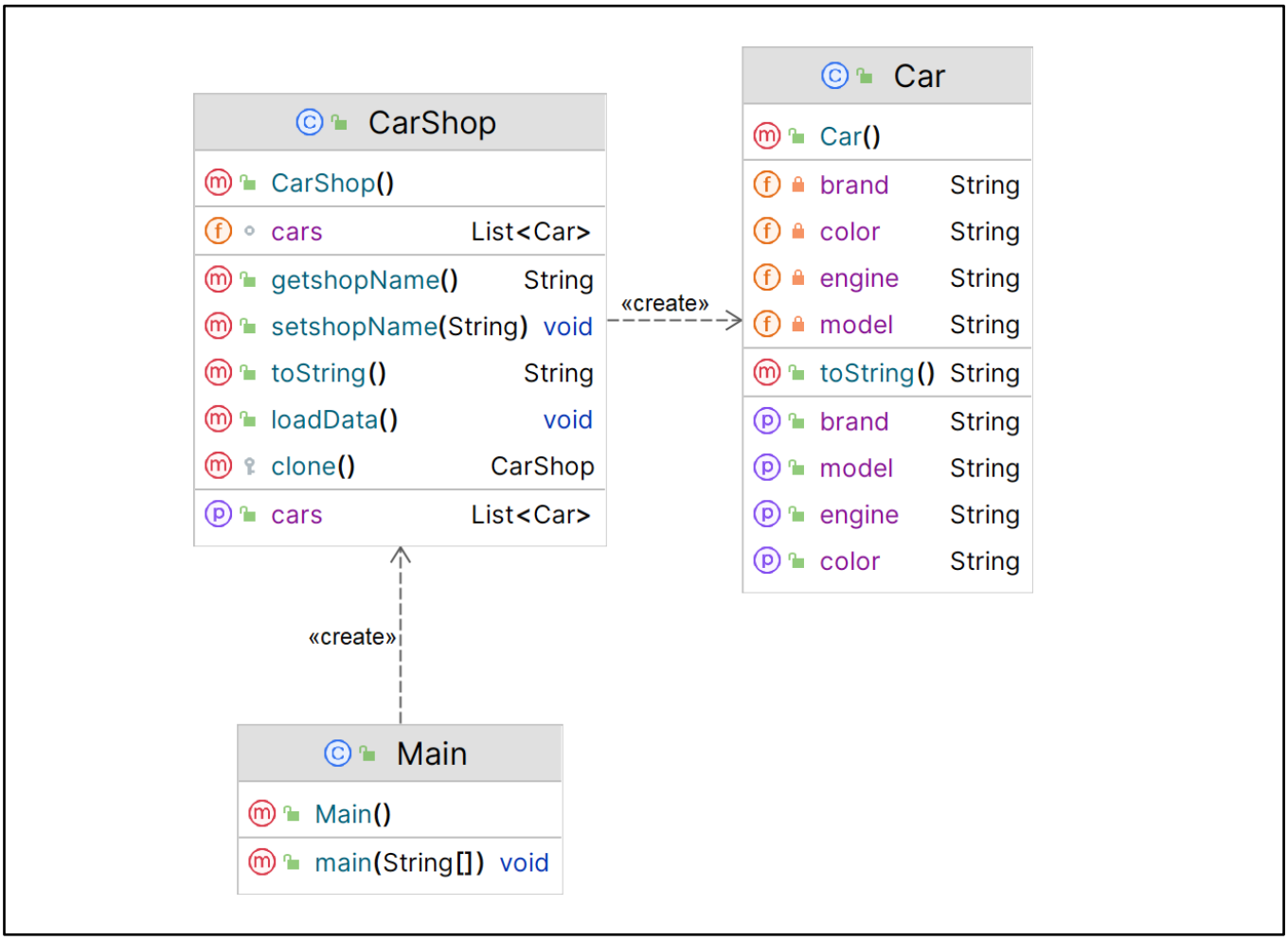
There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes **dependent** on that class. If the extra dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete class, when, for example, a parameter in a method accepts any objects that follow some interface.

Solution

The Prototype pattern delegates the **cloning** process to the actual objects that are being cloned. The pattern declares a **common interface** for all objects that support cloning. This interface lets you clone an object **without coupling** your code to the class of that object. The implementation of the **clone** method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one.

You can **even copy private fields** because most programming languages let objects access private fields of other objects that belong to the same class. An object that supports cloning is called a ***prototype***.

Structure (Class Diagram)



Implementation (Code)

Main.java

```
public class Main {
    public static void main(String[] args) throws CloneNotSupportedException {
        CarShop cs1 = new CarShop();
        cs1.setshopName("Shop 1");
        cs1.loadData();

        CarShop cs2 = cs1.clone();
        cs1.getCars().remove(2);
        cs2.setshopName("Shop 2");

        System.out.println(cs1);
        System.out.println(cs2);
    }
}
```

Car.java

```
public class Car {
   private String brand;
   private String engine;
   private String color;
   private String model;

   public String getBrand(){
      return brand;
   }
   public void setBrand(String brand){
      this.brand = brand;
   }
   public String getEngine(){
      return engine;
   }
   public void setEngine(String engine){
      this.engine = engine;
   }
   public String getColor(){
      return color;
   }
   public void setColor(String color){
      this.color = color;
   }
   public String getModel(){
      return model;
   }
   public void setModel(String model){
      this.model = model;
   }
   public String toString(){
      return "Car [Model = " + model + ", Engine = " + engine + ", Color = " + color + ", Brand =
" + brand + "]\n";
   }
}
```

BuildCar.java

```
import java.util.*;
public class CarShop implements Cloneable {
   private String shopName;
   List<Car> cars = new ArrayList<>();
   public String getshopName(){
      return shopName;
   }
   public void setshopName(String shopName){
      this.shopName = shopName;
   }

   public List<Car> getCars(){
      return cars;
   }
   public void setCars(List<Car> getCars){
      this.cars = cars;
   }
   public void loadData(){
      for (int i=65 ; i<=68 ; i++){
         Car c = new Car();
         c.setModel(String.valueOf(i));
         c.setBrand("Brand"+i);
         int flag = 0;
         c.setEngine("Petrol");
         c.setColor("#FF" + i + flag + "D");
         if (i%2==0){
            flag = 1;
            c.setEngine("Diesel");
            c.setColor("#FF" + i + flag + "D");
         }
         getCars().add(c);
      }
   }
   public String toString(){
      return "BookShop [ ShopName = " + shopName + ", Cars = " + cars+ " ]";
   }
   protected CarShop clone() throws CloneNotSupportedException {
      CarShop shop =  new CarShop();
      for ( Car c : this.getCars() ){
         shop.getCars().add(c);
      }
      return shop;
   }
}
```

Output

```
BookShop [ ShopName = Shop 1, Cars = [Car [Model = 65, Engine =
 Petrol, Color = #FF650D, Brand = Brand65]
, Car [Model = 66, Engine = Diesel, Color = #FF661D, Brand = Brand66]
, Car [Model = 68, Engine = Diesel, Color = #FF681D, Brand = Brand68]
] ]
BookShop [ ShopName = Shop 2, Cars = [Car [Model = 65, Engine =
 Petrol, Color = #FF650D, Brand = Brand65]
, Car [Model = 66, Engine = Diesel, Color = #FF661D, Brand = Brand66]
, Car [Model = 67, Engine = Petrol, Color = #FF670D, Brand = Brand67]
, Car [Model = 68, Engine = Diesel, Color = #FF681D, Brand = Brand68]
] ]
```

Applicability

1. Use the Prototype pattern when your **code shouldn't depend on the concrete classes** of objects that you need to copy.
2. Use the pattern when you want to **reduce the number of subclasses** that only differ in the way they initialize their respective objects.

Assignment 4: Abstract Factory Pattern

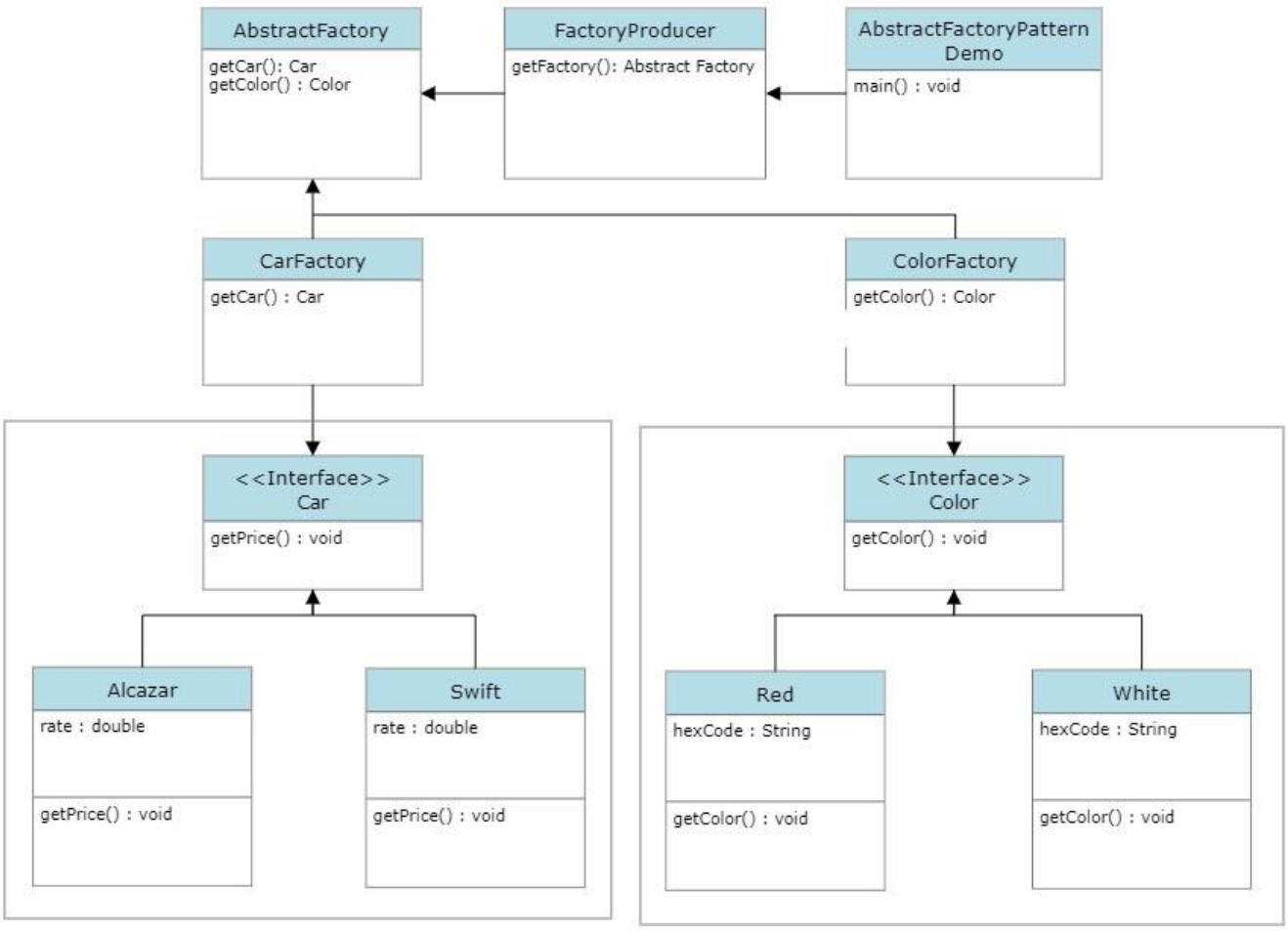
What is Abstract Factory Design Pattern?

Abstract Factory is a **creational design pattern** that lets you produce **families** of related objects without specifying their concrete classes.

Intend

- ✓ Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- ✓ A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".
- ✓ The new operator considered harmful.

Structure



Implementation (Code)

➢ **Car.java**

```
abstract class Car {
   protected double rate;
   abstract void getPrice();
}
```

- **Alcazar.java**

```
public class Alcazar extends Car {
   public void getPrice() {
      rate = 1500000;
      System.out.println("Rate for Alcazar: " + rate);
   }
}
```

- **Swift.java**

```
public class Swift extends Car {
   public void getPrice() {
      rate = 500000;
      System.out.println("Rate for Swift: " + rate);
   }
}
```

- **CarFactory.java**

```
public class CarFactory extends AbstractFactory{
   public Car getCar(String carName){
      if (carName == null){
         return null;
      }
      if (carName.equalsIgnoreCase("Alcazar")){
         return new Alcazar();
      }
      else if (carName.equalsIgnoreCase("Swift")){
         return new Swift();
      }
      return null;
   }
   Color getColor(String color) {
      return null;
   }
}
```

- **Color.java**

```
abstract class Color {
   protected String hexCode;
   abstract void getColor();
}
```

- **Red.java**

```
public class Red extends Color {
   public void getColor(){
      hexCode = "#FF0000";
      System.out.println("Hex Code: "+hexCode);
   }
}
```

- **White.java**

```
public class White extends Color {
   public void getColor(){
      hexCode = "#FFFFFF";
      System.out.println("Hex Code: "+hexCode);
   }
}
```

- **ColourFactory.java**

```
public class ColourFactory extends AbstractFactory{
   public Color getColor(String color){
      if (color == null){
         return null;
      }
      if (color.equalsIgnoreCase("Red")){
         return new Red();
      }
      else if (color.equalsIgnoreCase("White")){
         return new White();
      }
      return null;
   }
   Car getCar(String carName) {
      return null;
   }
}
```

- **AbstractFactory.java**

```
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Car getCar(String carName);
}
```

- **Producer.java**

```
public class Producer {
   public static AbstractFactory getFactory(String choice){
      if (choice.equals("Car")){
         return new CarFactory();
      }
      else if (choice.equals("Color")){
         return new ColourFactory();
      }
      return null;
   }
}
```

- **Demo.java**

```
import java.util.*;
public class Demo {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter the Car name[Alcazar, Swift]: ");
      String carName = sc.nextLine();
      System.out.print("Enter the Color[White, Red]: ");
      String color = sc.nextLine();


      AbstractFactory carFactory = Producer.getFactory("Car");
      assert carFactory != null;
      Car car1 = carFactory.getCar(carName);
      car1.getPrice();

      AbstractFactory colorFactory = Producer.getFactory("Color");
      assert colorFactory != null;
      Color color1 = colorFactory.getColor(color);
      color1.getColor();
   }
}
```

Applicability

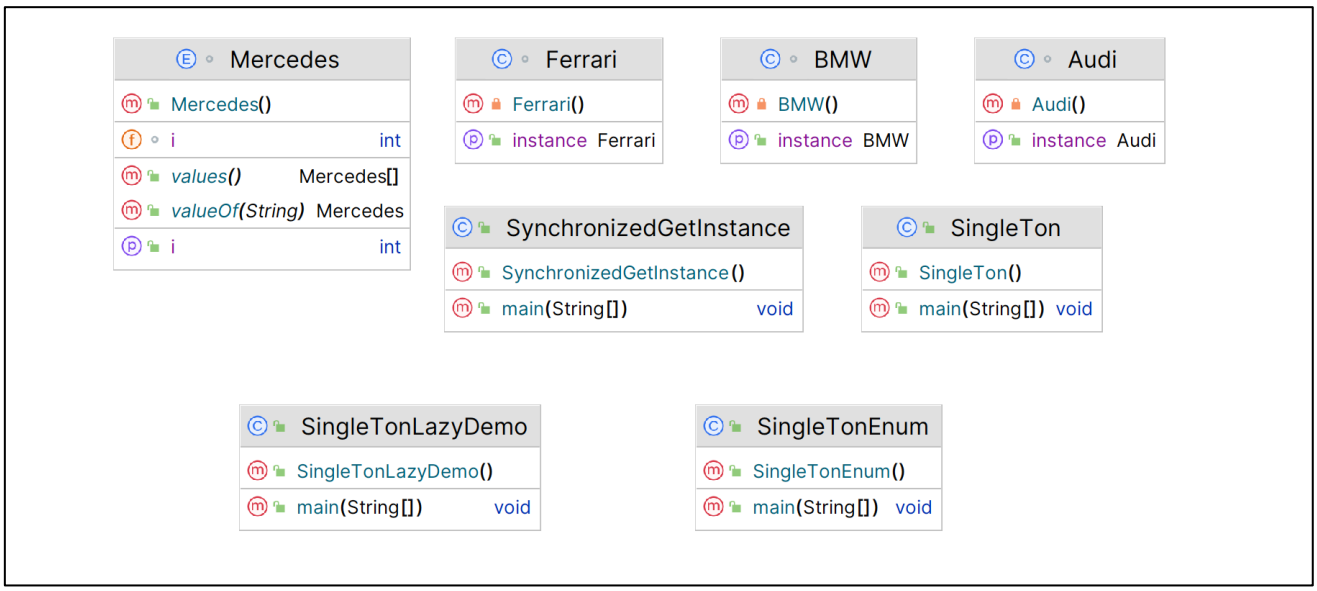
1. Use the **Abstract Factory** when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.
2. The **Abstract Factory** provides you with an interface for creating objects from each class of the **product family**. As long as your code creates objects via this interface, you don't have to worry about creating the wrong variant of a product which doesn't match the products already created by your app.

Assignment 5: Singleton Design Pattern

What is Singleton Design Pattern?

Singleton is a **creational design pattern**, which ensures that **only one object** of its kind exists and provides a **single point of access** to it for any other code.

Structure (Class Diagram)



Implementation (Code)

1) Eager Singleton

```
public class SingleTon {
   public static void main(String[] args) {
      Audi obj1 = Audi.getInstance();
      Audi obj2 = Audi.getInstance();
   }
}
class Audi {
   public static Audi obj = new Audi(); // Creating static object of class Audi
   private Audi(){ } // Creating Constructor
   // Creating Static object to achieve singleton pattern.
   public static Audi getInstance(){
      System.out.println("This is Audi Q3");
      return obj;
   }
}
```

Output:

```
This is Audi Q3
This is Audi Q3
```

2) Lazy Singleton

```
public class SingleTonLazyDemo {
    public static void main(String[] args) {
        BMW obj1 = BMW.getInstance();
        BMW obj2 = BMW.getInstance();
    }
}

class BMW {
    public static BMW obj = new BMW();
    private BMW(){
        System.out.println("This is BMW I4");
    }
    public static BMW getInstance(){
        if (obj == null){
            obj = new BMW(); // Creating the object here….lazy 😥
        }
        return obj;
    }
}
```

Output:

```
This is BMW I4
```

3) Double-checked Locking

```
public class SynchronizedGetInstance {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                Ferrari obj = Ferrari.getInstance();
            }
        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                Ferrari obj = Ferrari.getInstance();
            }
        });
        t1.start();
        t2.start();
    }
}
```

```
class Ferrari {
   public static Ferrari obj;
   private Ferrari(){
      System.out.println("Ferrari F8: 40200000");
   }
   public static Ferrari getInstance(){ // Double checked Locking – removing synchronized
      if (obj == null){
         synchronized (Ferrari.class) { // Putting Synchronized here
            if (obj == null) {
               obj = new Ferrari();
            }
         }
      }
      return obj;
   }
}
```

Output:

```
Ferrari F8: 40200000
```

4) Enum Singleton

```
public class SingleTonLazyDemo {
   public static void main(String[] args) {
      BMW obj1 = BMW.getInstance();
      BMW obj2 = BMW.getInstance();
   }
}
class BMW {
   public static BMW obj = new BMW();
   private BMW(){
      System.out.println("This is BMW I4");
   }
   public static BMW getInstance(){
      if (obj == null){
         obj = new BMW();
      }
      return obj;
   }
}
```

Output:

```
Price of Mercedes-Benz A-Class: 4200000
Price of Mercedes-Benz GLA Class: 5000000
```

Applicability

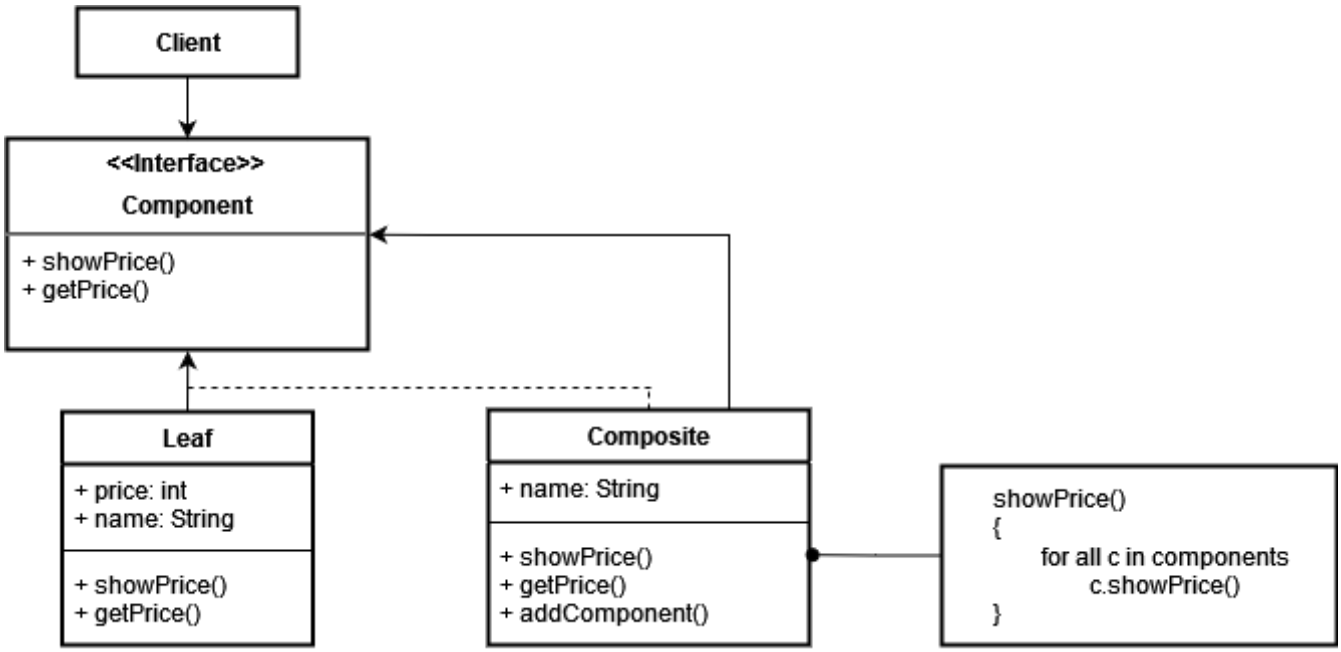
1. Use the **Singleton pattern** when a class in your program should have just a **single instance** available to all clients; for example, a single database object shared by different parts of the program.
2. The Singleton pattern disables all other means of creating objects of a class except for the special **creation method**. This method either creates a new object or returns an existing one if it has already been created.
3. Use the Singleton pattern when you need **stricter control over global variables**.

Assignment 6: Composite Design Pattern

What is Composite Design Pattern?

Composite is a **structural design pattern** that lets you compose objects into tree structures and then work with these structures as if they were **individual objects**.

Structure (Class Diagram)



Implementation (Code)

1) Component.java

```
interface Component {
   void showPrice();
   int getPrice();
}
class Leaf implements Component {
   int price;
   String name;
   Leaf(String name, int price){
      this.name = name;
      this.price = price;
   }
   public void showPrice(){
      System.out.println("Leaf -> " + name + " : " + price);
   }
   public int getPrice() {  return price;  }
}
```

2) Composite.java

```
public class CompositePattern {
   public static void main(String[] args) {
      Component flw = new Leaf("Front Left Wheel", 4000);
      Component frw = new Leaf("Front Right Wheel", 4500);
      Component rlw = new Leaf("Rear Left Wheel", 5000);
      Component rrw = new Leaf("Rear Right Wheel", 4000);
      Component piston = new Leaf("Piston", 9000);
      Component chamber = new Leaf("Combustion Chamber", 8000);
      Component camshaft = new Leaf("Cam Shaft", 10000);
      Component valves = new Leaf("Valves", 8500);

      Composite wheel = new Composite("Wheel");
      Composite engine = new Composite("Engine");
      Composite car = new Composite("Car");

      wheel.addComponent(flw);
      wheel.addComponent(frw);
      wheel.addComponent(rlw);
      wheel.addComponent(rrw);
      engine.addComponent(piston);
      engine.addComponent(chamber);
      engine.addComponent(camshaft);
      engine.addComponent(valves);

      car.addComponent(wheel);
      car.addComponent(engine);
      car.showPrice();
   }
}
```

3) CompositePattern.java

```
class Composite implements Component{
   String name;
   List<Component> components = new ArrayList<>();

   public Composite(String name) {
      super();
      this.name = name;
   }
   public void addComponent(Component com){
      components.add(com);
   }
   public int getPrice(){
      int p=0;
      for (Component c : components){
         p += c.getPrice();
      }
      return p;
   }
   public void showPrice(){
      System.out.println("\nComposite -> " + name + " : Price " + getPrice());
```

```
        System.out.println("Leaf of " + name + "\n[");
        for (Component c : components){
            c.showPrice();
        }
        System.out.println("]");
    }
}
```

Output:

```
Composite -> Car : Price 53000
Leaf of Car
[

Composite -> Wheel : Price 17500
Leaf of Wheel
[
Leaf -> Front Left Wheel : 4000
Leaf -> Front Right Wheel : 4500
Leaf -> Rear Left Wheel : 5000
Leaf -> Rear Right Wheel : 4000
]
```

```
Composite -> Engine : Price 35500
Leaf of Engine
[
Leaf -> Piston : 9000
Leaf -> Combustion Chamber : 8000
Leaf -> Cam Shaft : 10000
Leaf -> Valves : 8500
]
]
```

Applicability

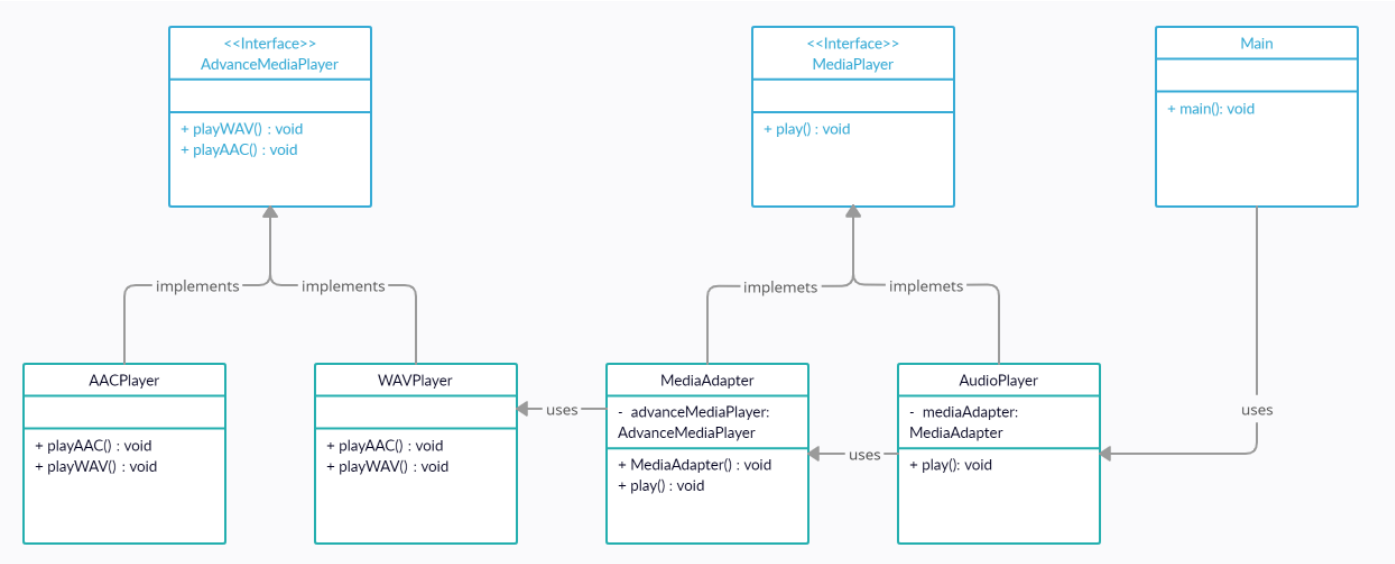
1. Use the **Composite** pattern when you have to implement a **tree-like object structure**.
2. Use the pattern when you want the client code to treat both **simple** and **complex elements uniformly**.

Assignment 7: Adapter Design Pattern

What is Adapter Design Pattern?

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Structure (Class Diagram)



A Car has an audio player. It plays a different types of audio files. To play those we need an **Adapter Class** to convert those files to mp3 file and play the audio. I have created a program which supports only **“.mp3”, “.aac” & “.wav”** formats. If any other file format is used then the code gives a message that the **“File format is not supported”**.

Implementation (Code)

```
interface MediaPlayer {
    public void play(String audioType ,String fileName);
}

interface AdvanceMediaPlayer {
    public void playAAC(String audioType, String fileName);
    public void playWAV(String audioType, String fileName);
}

class AACPlayer implements AdvanceMediaPlayer {
    public void playAAC(String audioType, String fileName) {
        System.out.println("Playing " + fileName + "." + audioType);
    }
    public void playWAV(String audioType, String fileName) {
        // Do Nothing
    }
}
```

```
class WAVPlayer implements AdvanceMediaPlayer {
    public void playAAC(String audioType, String fileName) {
        // Do Nothing
    }
    public void playWAV(String audioType, String fileName) {
        System.out.println("Playing " + fileName + "." + audioType);
    }
}

class MediaAdapter implements MediaPlayer {
    AdvanceMediaPlayer advMusicPlay;

    public MediaAdapter(String audioType){
        if (audioType.equalsIgnoreCase("aac")) {
            advMusicPlay = new AACPlayer();
        } else if (audioType.equalsIgnoreCase("wav")) {
            advMusicPlay = new WAVPlayer();
        }
    }

    public void play (String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("aac")) {
            advMusicPlay.playAAC(audioType, fileName);
        } else if (audioType.equalsIgnoreCase("wav")) {
            advMusicPlay.playWAV(audioType, fileName);
        }
    }
}

class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;

    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing " + fileName + "." + audioType);
        }
        else if (audioType.equalsIgnoreCase("aac") ||
audioType.equalsIgnoreCase("wav")) {
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        }
        else {
            System.out.println("Sorry! '" + audioType + "' format not
supported");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();
        Scanner sc = new Scanner(System.in);

        System.out.print("How many files do you have :: ");
        int n = sc.nextInt();

        for (int i=1 ; i<=n ; i++) {
            System.out.print("\nEnter Name of File " + i + " (without extension)
:: ");
            String fileName = sc.next();
            System.out.print("Enter Audio Type of file" + i + " :: ");
            String audioType = sc.next();

            audioPlayer.play(audioType, fileName);
        }
    }
}
```

Output

```
How many files do you have :: 4

Enter Name of File 1 (without extension) :: NoLie
Enter Audio Type of file1 :: mp3
Playing NoLie.mp3

Enter Name of File 2 (without extension) :: Excuses
Enter Audio Type of file2 :: aac
Playing Excuses.aac

Enter Name of File 3 (without extension) :: SummerHigh
Enter Audio Type of file3 :: wav
Playing SummerHigh.wav

Enter Name of File 4 (without extension) :: StereoHearts
Enter Audio Type of file4 :: avi
Sorry! 'avi' format not supported
```

Applicability

1. **Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.**
   The Adapter pattern lets you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd-party class or any other class with a weird interface.

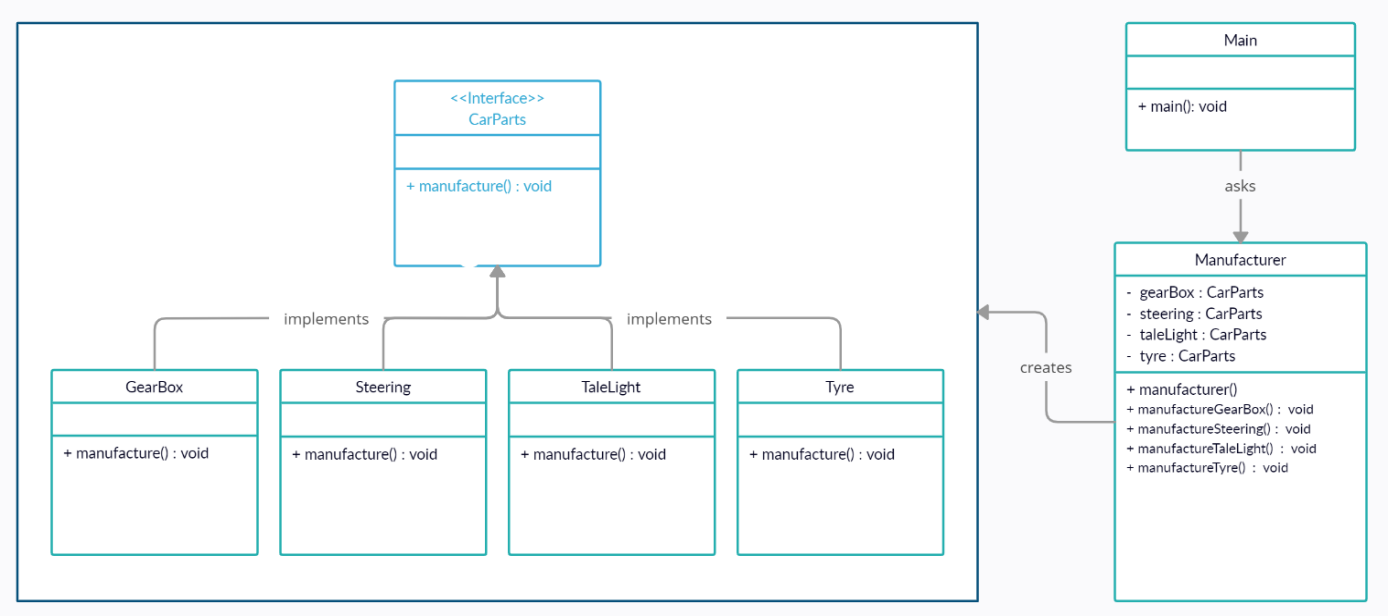
2. **Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.**
   You could extend each subclass and put the missing functionality into new child classes. However, you'll need to duplicate the code across all of these new classes.

Assignment 8: Facade Design Pattern

What is Facade Design Pattern?

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Structure (Class Diagram)



Implementation (Code)

```
public interface CarParts {
    void manufacture();
}


public class GearBox implements CarParts {
    public void manufacture() {
        System.out.println("Gear Box Manufacturing in Progress\nManufacturing
Process Completed ✅");
    }
}


public class Steering implements CarParts {
    public void manufacture() {
        System.out.println("Steering Manufacturing in Progress\nManufacturing
Process Completed ✅");
    }
}
```

```
public class TaleLight implements CarParts {
    public void manufacture() {
        System.out.println("Tale Light Manufacturing in
Progress\nManufacturing Process Completed ✅");
    }
}


public class Tyre implements CarParts {
    public void manufacture() {
        System.out.println("Tyre Manufacturing in Progress\nManufacturing
Process Completed ✅");
    }
}


public class Manufacturer {
    private CarParts gearBox;
    private CarParts steering;
    private CarParts taleLight;
    private CarParts tyre;

    public Manufacturer() {
        gearBox = new GearBox();
        steering = new Steering();
        taleLight = new TaleLight();
        tyre = new Tyre();
    }

    public void manufactureGearBox() {
        gearBox.manufacture();
    }
    public void manufactureStreering() {
        steering.manufacture();
    }
    public void manufactureTaleLight() {
        taleLight.manufacture();
    }
    public void manufactureTyre() {
        tyre.manufacture();
    }
}
```

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Manufacturer manufacturer = new Manufacturer();

        System.out.println("What do you want to manufacture? ");
        System.out.println("1. GearBox \n2. Steering \n3. TaleLight \n4.
Tyre");
        System.out.print("Enter your Choice : ");
        int choice = sc.nextInt();
        System.out.println();

        switch (choice) {
            case (1) -> manufacturer.manufactureGearBox();
            case (2) -> manufacturer.manufactureStreering();
            case (3) -> manufacturer.manufactureTaleLight();
            case (4) -> manufacturer.manufactureTyre();
            default -> System.out.println("Enter a Valid Choice!");
        }

    }
}
```

Output

```
What do you want to manufacture?
1. GearBox
2. Steering
3. TaleLight
4. Tyre
Enter your Choice : 1

Gear Box Manufacturing in Progress
Manufacturing Process Completed ✅
```

Applicability

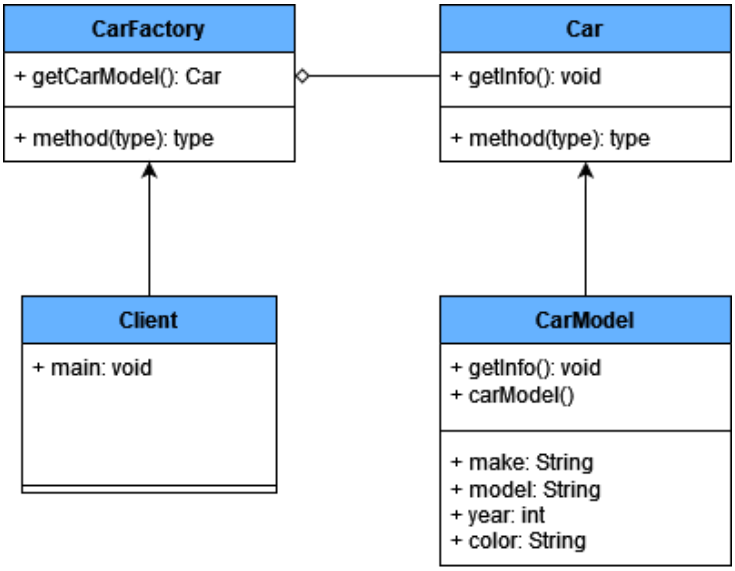
1. Use the **Facade** pattern when you need to have a **limited but straightforward interface** to a complex subsystem.
2. Use the Facade when you want to structure a **subsystem into layers.**

Assignment 9: Flyweight Design Pattern

What is Flyweight Design Pattern?

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Structure (Class Diagram)



Implementation (Code)

```
import java.util.*;

// Flyweight interface
interface Car {
    void getInfo(String owner, String location, int mileage);
}

// Concrete flyweight class
class CarModel implements Car {
    private String make;
    private String model;
    private int year;
    private String color;

    public CarModel(String make, String model, int year, String color) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.color = color;
    }

    public void getInfo(String owner, String location, int mileage) {
        System.out.println("\nCar Model: " + make + " " + model + " " + year + " " + color);
        System.out.println("Current owner: " + owner);
        System.out.println("Current location: " + location);
        System.out.println("Current mileage: " + mileage);
    }
}
```

```
// Flyweight factory
class CarFactory {
    private static Map<String, Car> carModels = new HashMap<>();
    public static Car getCarModel(String make, String model, int year, String color) {
        String key = make + model + year + color;
        Car carModel = carModels.get(key);
        if (carModel == null) {
            carModel = new CarModel(make, model, year, color);
            carModels.put(key, carModel);
        }
        return carModel;
    }
}

// Client code
public class Client {
    public static void main(String[] args) {
        Car car1 = CarFactory.getCarModel("Toyota", "Fortuner", 2021, "Red");
        Car car2 = CarFactory.getCarModel("Toyota", "Camry", 2021, "Red");

        // Both car1 and car2 will reference the same CarModel object
        if (car1 == car2) {
            System.out.println("Car1 and Car2 reference the same CarModel object");
        }

        car1.getInfo("John", "London", 6000);
        car2.getInfo("Jane", "Canada", 3000);
    }
}
```

Output:

```
Car Model: Toyota Fortuner 2021 Red
Current owner: John
Current location: London
Current mileage: 6000

Car Model: Toyota Camry 2021 Red
Current owner: Jane
Current location: Canada
Current mileage: 3000
```

Applicability

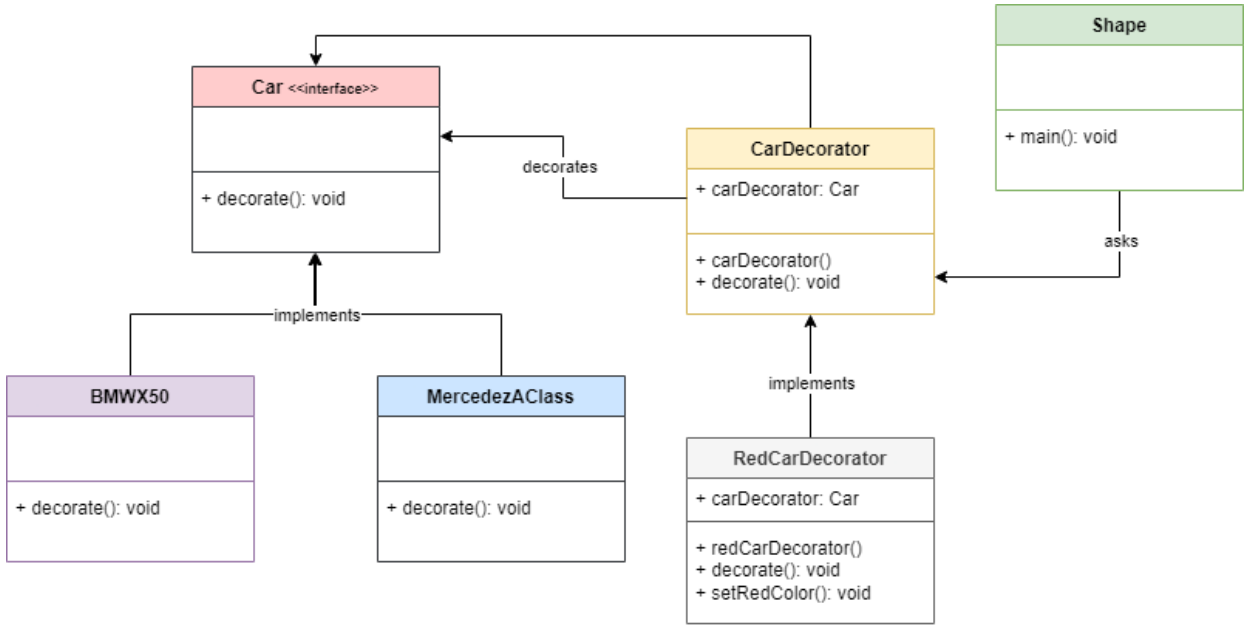
1. Use the Flyweight pattern only when your program must support a huge number of objects which barely fit into available RAM.
2. The benefit of applying the pattern depends heavily on how and where it's used. It's most useful when:
   - An application needs to spawn a huge number of similar objects
   - This drains all available RAM on a target device
   - The objects contain duplicate states which can be extracted and shared between multiple objects

Assignment 10: Decorator Design Pattern

What is Decorator Design Pattern?

Decorator is a structural design pattern that lets you attach new behaviours to objects by placing these objects inside special wrapper objects that contain the behaviours.

Structure (Class Diagram)



Implementation (Code)

```
interface Car {
   void decorate();
}

class BMWX50 implements Car {
   public void decorate() {
      System.out.println("Car: BMWX50");
   }
}

class MercedezAClass implements Car {
   public void decorate() {
      System.out.println("Car: MercedezAClass");
   }
}

abstract class carDecorator implements Car {
   protected Car decoratedCar;
   public carDecorator(Car decoratedCar){
      this.decoratedCar = decoratedCar;
   }
   public void decorate(){
      decoratedCar.decorate();
   }
}

class RedCarDecorator extends carDecorator {
   public RedCarDecorator(Car decoratedCar) {
```

```
        super(decoratedCar);
    }
    public void decorate() {
        decoratedCar.decorate();
        setRedColor(decoratedCar);
    }
    private void setRedColor(Car decoratedCar){
        System.out.println("Car Color: Red");
    }
}

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

        Car mercedezAClass = new MercedezAClass();
        Car bmwX50 = new BMWX50();

        Car redMercedezAClass = new RedCarDecorator(new MercedezAClass());
        Car redBMWX50 = new RedCarDecorator(new BMWX50());

        System.out.println("\nBMW X-50 with white Color");
        bmwX50.decorate();

        System.out.println("\nMercedez A-Class with white Color");
        mercedezAClass.decorate();

        System.out.println("\nBMW X-50 of red color");
        redMercedezAClass.decorate();

        System.out.println("\nMercedez A-Class of red color");
        redBMWX50.decorate();
    }
}
```

Output

```
BMW X-50 with white Color
Car: BMWX50

Mercedez A-Class with white Color
Car: MercedezAClass

BMW X-50 of red color
Car: MercedezAClass
Car Color: Red

Mercedez A-Class of red color
Car: BMWX50
Car Color: Red
```

Applicability

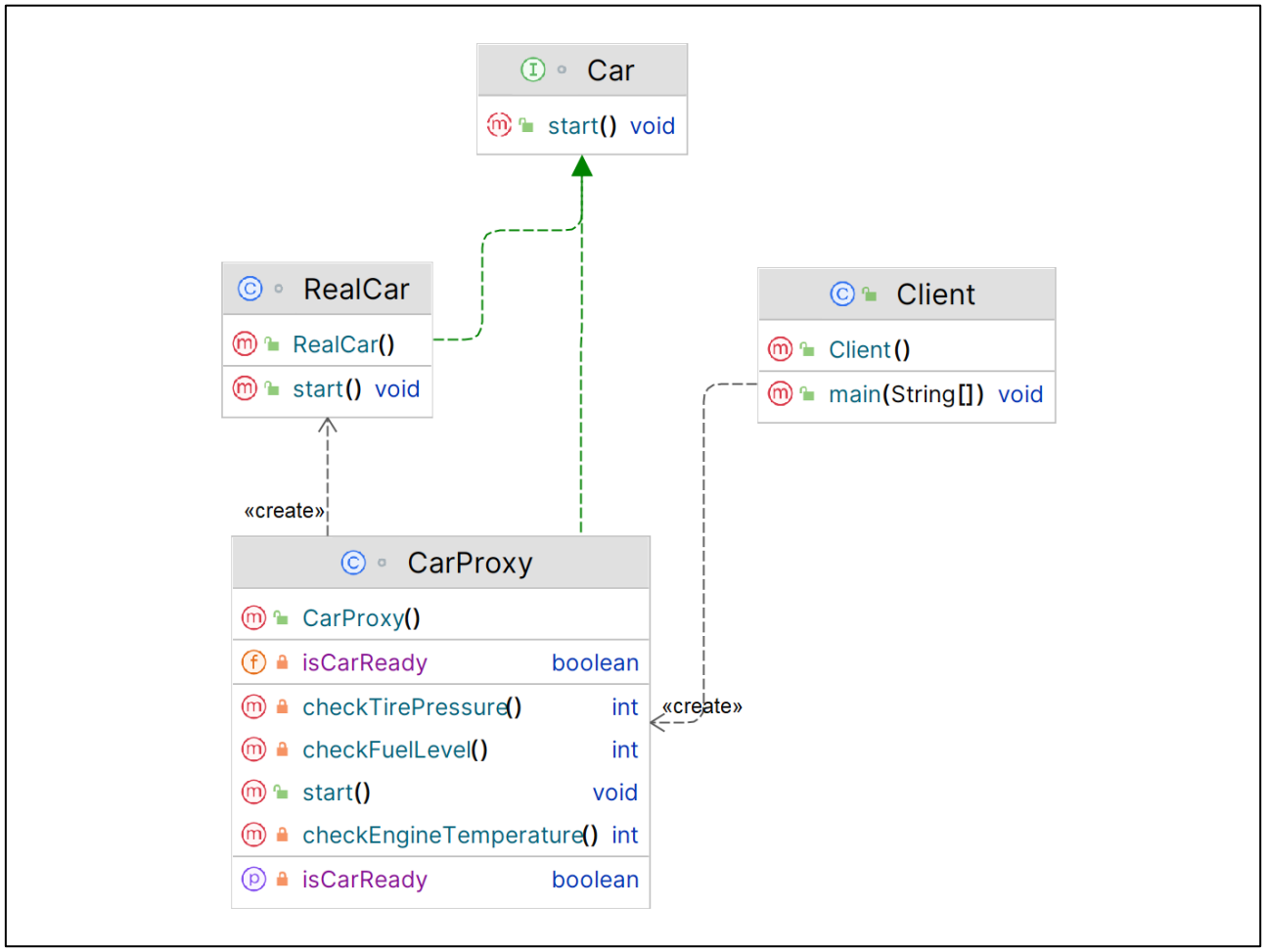
1. Use the **Decorator** pattern when you need to be able to **assign extra behaviours** to objects at runtime **without breaking the code** that uses these objects.
2. Use the pattern when **it's awkward** or **not possible** to extend an object's behaviour using inheritance.

Assignment 11: Proxy Design Pattern

What is Proxy Design Pattern?

Proxy is a **structural design pattern** that lets you provide a **substitute** or **placeholder** for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Structure (Class Diagram)



Implementation (Code)

```
// Subject interface
interface Car {
    void start();
}

// Real subject
class RealCar implements Car {
    public void start() {
        System.out.println("Starting the car");
    }
}
```

```
// Proxy
class CarProxy implements Car {
    private RealCar realCar;
    private boolean isCarReady;
    private int fuelLevel;
    private int tirePressure;
    private int engineTemperature;

    public void start() {
        if (realCar == null) {
            realCar = new RealCar();
        }
        if (isCarReady()) {
            realCar.start();
        }
    }

    private boolean isCarReady() {
        if (!isCarReady) {
            System.out.println("Checking the car's status...");
            fuelLevel = checkFuelLevel();
            tirePressure = checkTirePressure();
            engineTemperature = checkEngineTemperature();
            isCarReady = fuelLevel > 0 && tirePressure > 0 && engineTemperature < 100;
        }
        return isCarReady;
    }

    private int checkFuelLevel() {
        // Perform checks to determine the fuel level
        int fuelLevel = 50; // Set to 50 for demonstration purposes only
        System.out.println("Fuel level: " + fuelLevel);
        return fuelLevel;
    }

    private int checkTirePressure() {
        // Perform checks to determine the tire pressure
        int tirePressure = 30; // Set to 30 for demonstration purposes only
        System.out.println("Tire pressure: " + tirePressure);
        return tirePressure;
    }

    private int checkEngineTemperature() {
        // Perform checks to determine the engine temperature
        int engineTemperature = 90; // Set to 90 for demonstration purposes only
        System.out.println("Engine temperature: " + engineTemperature);
        return engineTemperature;
    }
}
```

```
// Client code
public class Client {
    public static void main(String[] args) {
        Car car = new CarProxy();
        car.start();
    }
}
```

Output:

```
Checking the car's status...
Fuel level: 50
Tire pressure: 30
Engine temperature: 90
Starting the car
```

Applicability

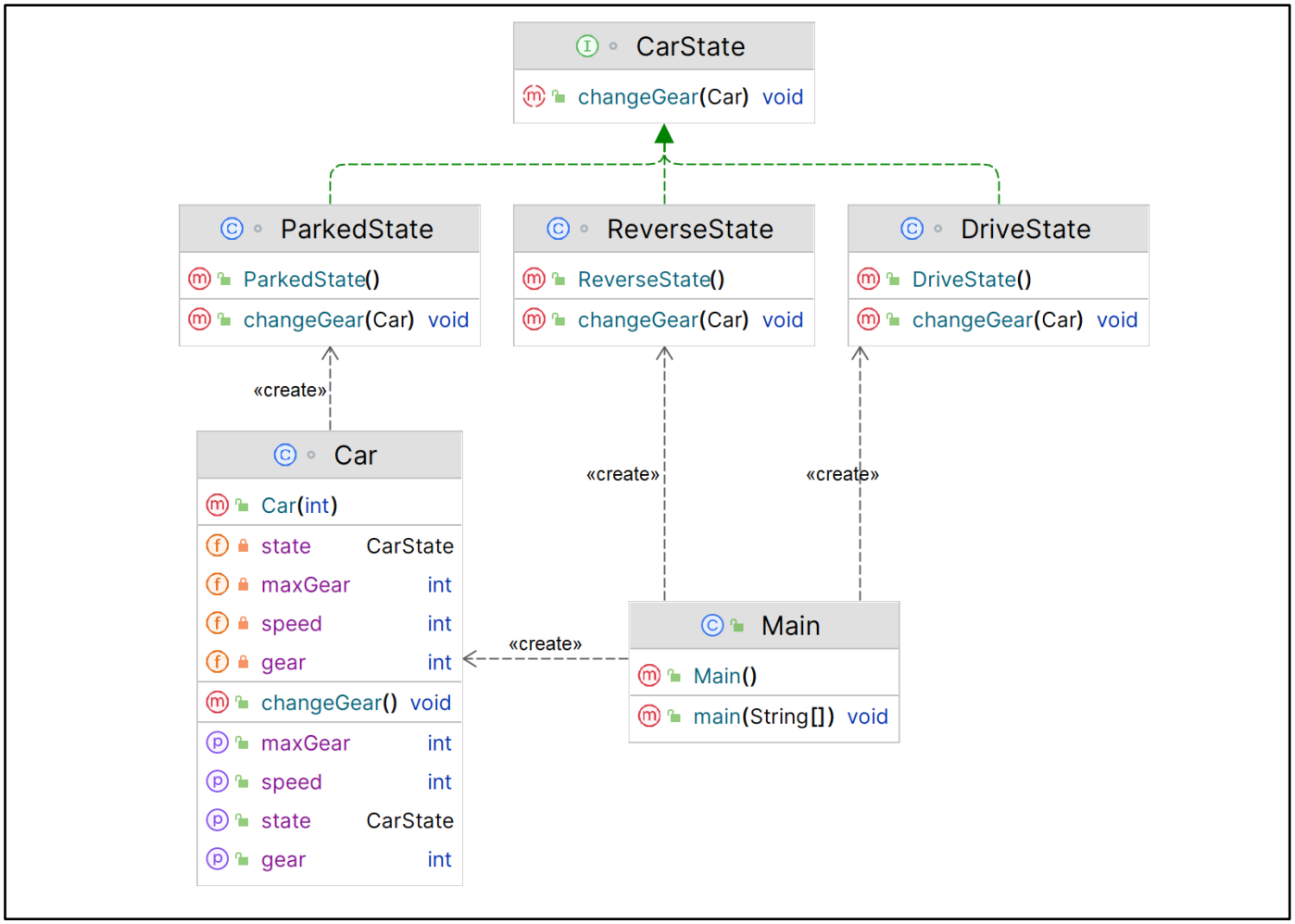
1. **Lazy initialization (virtual proxy):** This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.
2. **Access control (protection proxy):** This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).
3. **Local execution of a remote service (remote proxy):** This is when the service object is located on a remote server.
4. **Logging requests (logging proxy):** This is when you want to keep a history of requests to the service object.
5. **Caching request results (caching proxy):** This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.
6. **Smart reference:** This is when you need to be able to dismiss a heavyweight object once there are no clients that use it.

Assignment 12: State Design Pattern

What is State Design Pattern?

State is a **behavioural design pattern** that lets an object alter its behaviour when its internal state changes. It appears as if the object changed its class.

Structure (Class Diagram)



Implementation (Code)

```
// Interface for CarState
interface CarState {
    void changeGear(Car car);
}

// Concrete class for ParkedState
class ParkedState implements CarState {

    public void changeGear(Car car) {
        // Car can only change gear when it's not parked
        System.out.println("Cannot change gear while car is parked.");
```

```
    }
}

// Concrete class for DriveState
class DriveState implements CarState {
      public void changeGear(Car car) {
       // Car can change gear to higher gear when driving at certain speed
       if (car.getSpeed() < 20) {
          System.out.println("Cannot change to higher gear when car is moving slowly.");
       } else if (car.getGear() >= car.getMaxGear()) {
          System.out.println("Cannot shift to higher gear, already in top gear.");
       } else {
          car.setGear(car.getGear() + 1);
          System.out.println("Changed gear to " + car.getGear());
       }
    }
}

// Concrete class for ReverseState
class ReverseState implements CarState {

    public void changeGear(Car car) {
       // Car can only change to reverse gear when speed is 0
       if (car.getSpeed() > 0) {
          System.out.println("Cannot shift to reverse gear when car is moving forward.");
       } else {
          car.setGear(-1);
          System.out.println("Changed gear to reverse");
       }
    }
}

// Context class for Car
class Car {
    private int speed;
    private int gear;
    private int maxGear;
    private CarState state;

    public Car(int maxGear) {
       this.speed = 0;
       this.gear = 0;
       this.maxGear = maxGear;
       this.state = new ParkedState();
    }
```

```
    public void changeGear() {
        this.state.changeGear(this);
    }

    // Getters and setters for speed, gear, and maxGear

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public int getSpeed() {
        return this.speed;
    }

    public void setGear(int gear) {
        this.gear = gear;
    }

    public int getGear() {
        return this.gear;
    }

    public int getMaxGear() {
        return this.maxGear;
    }

    // Method to set the state of the car
    public void setState(CarState state) {
        this.state = state;
    }
}

// Example usage
public class Main {
    public static void main(String[] args) {
        Car car = new Car(4);

        // Car starts in parked state
        car.changeGear(); // Output: "Cannot change gear while car is parked."

        // Car can shift to reverse gear when speed is 0
        car.setState(new ReverseState());
        car.changeGear(); // Output: "Changed gear to reverse"
```

```
        // Car cannot shift to higher gear when moving slowly
        car.setState(new DriveState());
        car.setSpeed(10);
        car.changeGear(); // Output: "Cannot change to higher gear when car is moving slowly."

        // Car can shift to higher gear when moving at certain speed
        car.setSpeed(25);
        car.changeGear(); // Output: "Changed gear to 0"
        car.changeGear(); // Output: "Changed gear to 1"
        car.changeGear(); // Output: "Changed gear to 2"
        car.changeGear(); // Output: "Changed gear to 3"
        car.changeGear(); // Output: "Changed gear to 4"

        // Car cannot shift to higher gear when already in top gear
        car.changeGear(); // Output: "Cannot shift to higher gear, already in top gear"
    }
}
```

Output

```
Cannot change gear while car is parked.
Changed gear to reverse
Cannot change to higher gear when car is moving slowly.
Changed gear to 0
Changed gear to 1
Changed gear to 2
Changed gear to 3
Changed gear to 4
Cannot shift to higher gear, already in top gear.
```

Applicability

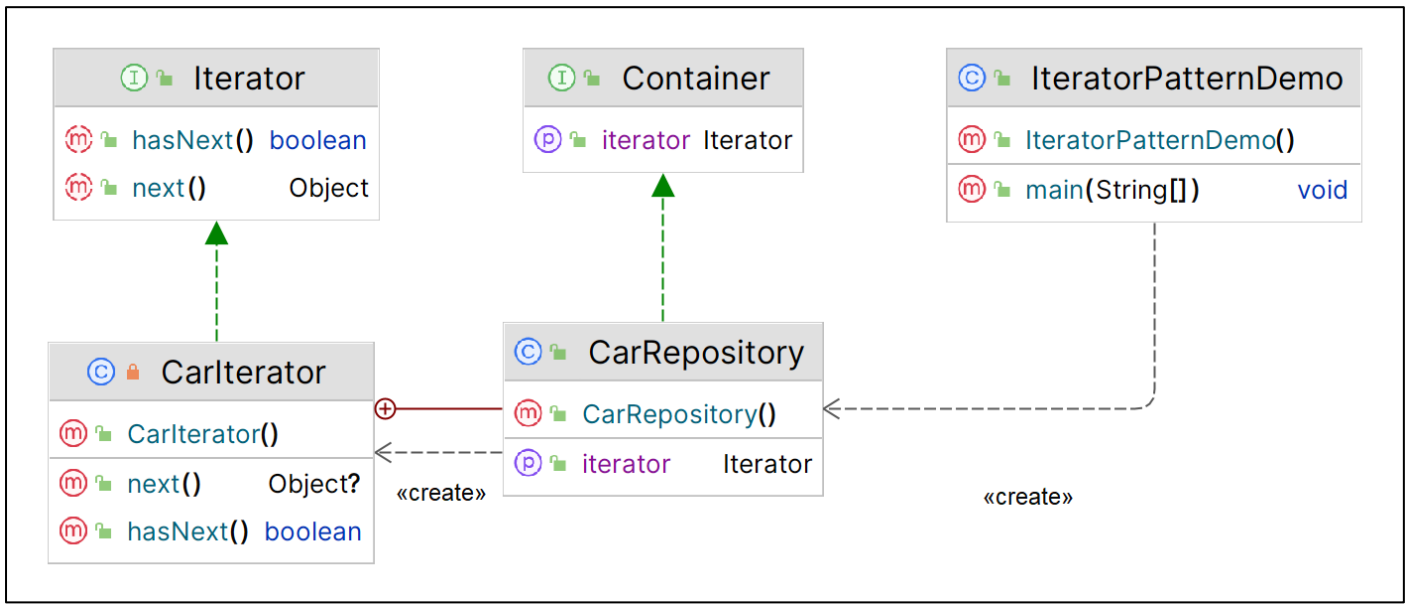
1. Use the **State pattern** when you have an object that **behaves differently** depending on its current state, the number of states is enormous, and the **state-specific code** changes frequently.
2. Use the pattern when you have a class polluted with massive conditionals that alter how the class **behaves** according to the current values of the class's fields.
3. Use State when you have a lot of **duplicate code** across similar states and transitions of a condition-based **state machine**.

Assignment 13: Iterator Design Pattern

What is Iterator Design Pattern?

The **Iterator** design pattern allows us to **traverse** a collection of objects **without exposing** the underlying **implementation** of the collection.

Structure (Class Diagram)



Implementation (Code)

```
// Interface for Iterator
public interface Iterator {
   public boolean hasNext();
   public Object next();
}

// Returns new instances of Iterator
public interface Container {
   public Iterator getIterator();
}

// Concrete Iterator to implement traversal of repository
public class CarRepository implements Container {
   public String cars[] = {"Mercedes", "BMW", "Audi", "Ferrari", "Jaguar"};

   public Iterator getIterator() {
      return new CarIterator();
   }

   private class CarIterator implements Iterator {
      int index;
```

```
        public boolean hasNext() {
            if (index < cars.length) {
                return true;
            }
            return false;
        }

        public Object next() {
            if (this.hasNext()) {
                return cars[index++];
            }
            return null;
        }
    }
}
```

// Demo - Main

```
public class IteratorPatternDemo {
    public static void main(String[] args) {
        CarRepository carRepo = new CarRepository();

        for (Iterator iter = carRepo.getIterator(); iter.hasNext();) {
            String name = (String)iter.next();
            System.out.println("Car : " + name);
        }
    }
}
```

Output

```
Car : Mercedes
Car : BMW
Car : Audi
Car : Ferrari
Car : Jaguar
```

Applicability

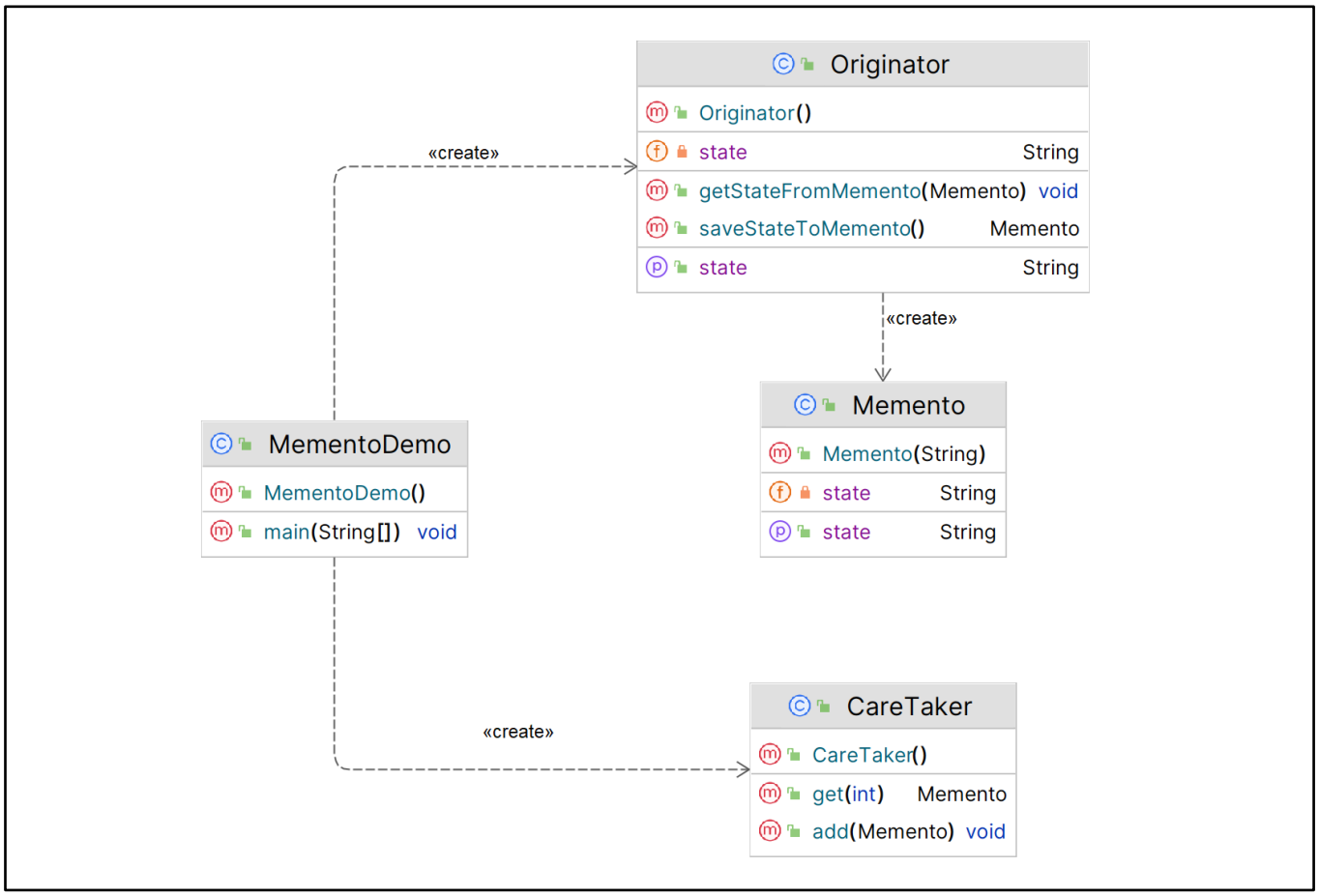
1. Use the **Iterator** pattern when your collection has a **complex data structure** under the hood, but you want to **hide its complexity from clients** (either for convenience or security reasons).
2. Use the pattern to **reduce duplication** of the traversal code across your app.
3. Use the Iterator when you want your code to be able to traverse different data structures or **when types of these structures are unknown** beforehand.

Assignment 14: Memento Design Pattern

What is Memento Design Pattern?

The **Memento** design pattern is a **behavioural** pattern that allows you to ***capture*** and ***save*** the state of an object **without violating encapsulation**.

Structure (Class Diagram)



Implementation (Code)

```
import java.util.*;

// Memento class
public class Memento {
    private String state;

    public Memento(String state) {
        this.state = state;
    }
    public String getState() {
        return state;
    }
}
```

```
// Originator Class
public class Originator {
   private String state;

   public String getState() {
      return state;
   }
   public void setState(String state) {
      this.state = state;
   }
   public Memento saveStateToMemento() {
      return new Memento(state);
   }
   public void getStateFromMemento(Memento memento) {
      state = memento.getState();
   }
}

// Caretaker class
public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();
   public void add(Memento state) {
      mementoList.add(state);
   }
   public Memento get (int index) {
      return mementoList.get(index);
   }
}

// Main - Demo
public class MementoDemo {
   public static void main(String[] args) {
      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();

      originator.setState("State 1");
      careTaker.add(originator.saveStateToMemento()); // Saved state 1 at index 0
      originator.setState("State 2");
      careTaker.add(originator.saveStateToMemento()); // Saved state 2 at index 1

      originator.setState("State 3");
      careTaker.add(originator.saveStateToMemento()); // Saved state 3 at index 2

      originator.setState("State 4");
```

```
        System.out.println("Current State: " + originator.getState());

        originator.getStateFromMemento(careTaker.get(0));
        System.out.println("First saved state: " + originator.getState());

        originator.getStateFromMemento(careTaker.get(1));
        System.out.println("Second saved state: " + originator.getState());

        originator.getStateFromMemento(careTaker.get(2));
        System.out.println("Second saved state: " + originator.getState());
    }
}
```

Applicability

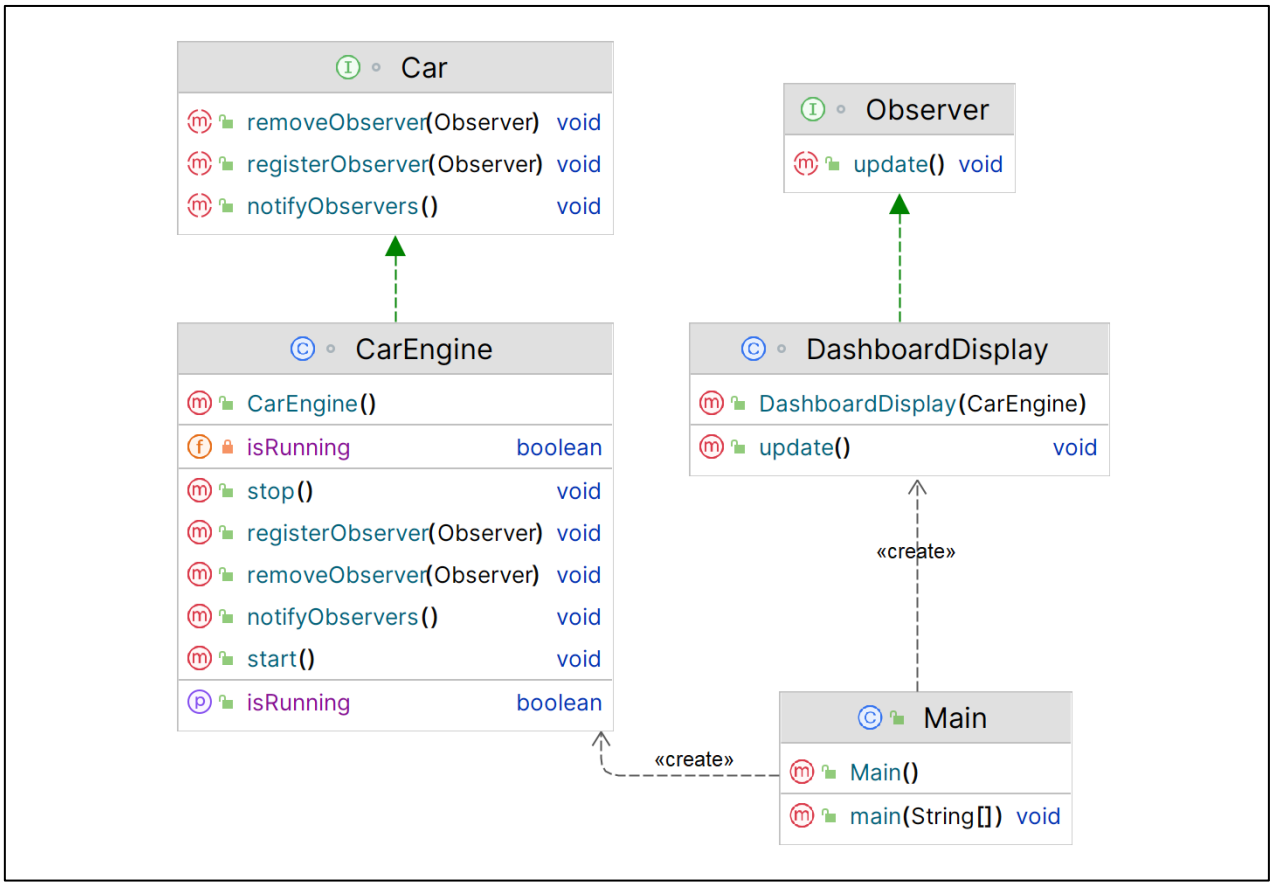
1. Use the **Memento** pattern when you want to produce **snapshots** of the object's state to be able to restore a previous state of the object.
2. Use the pattern when direct access to the object's fields/getters/setters violates its **encapsulation**.

Assignment 15: Observer Design Pattern

What is Observer Design Pattern?

Observer is a **behavioural** design pattern that lets you define a **subscription mechanism** to **notify** multiple objects about any events that happen to the object they're **observing**.

Structure (Class Diagram)



Implementation (Code)

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

// Define the Subject interface
interface Car {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// Define the Observer interface
interface Observer {
    void update();
}
```

```
// Define the concrete Subject class
class CarEngine implements Car {
    private List<Observer> observers = new ArrayList<>();
    private boolean isRunning = false;
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }

    // This method is called to start the car engine
    public void start() {
        System.out.println("Starting the car engine.");
        isRunning = true;
        notifyObservers();
    }
    // This method is called to stop the car engine
    public void stop() {
        System.out.println("Stopping the car engine.");
        isRunning = false;
        notifyObservers();
    }
    // This method is called to check if the car engine is running
    public boolean isRunning() {
        return isRunning;
    }
}

// Define the concrete Observer class
class DashboardDisplay implements Observer {
    private CarEngine carEngine;

    public DashboardDisplay(CarEngine carEngine) {
        this.carEngine = carEngine;
    }
    public void update() {
        if (carEngine.isRunning()) {
            System.out.println("Displaying current speed.");
        } else {
            System.out.println("Displaying engine warning light.");
        }
    }
}
```

```
// Usage example
public class Main {
    public static void main(String[] args) {
        // Create the subject (car engine)
        CarEngine carEngine = new CarEngine();

        // Create some observers (dashboard displays)
        DashboardDisplay display1 = new DashboardDisplay(carEngine);
        DashboardDisplay display2 = new DashboardDisplay(carEngine);

        // Register the observers with the subject
        carEngine.registerObserver(display1);
        carEngine.registerObserver(display2);

        // Start the car engine
        carEngine.start();

        // Stop the car engine
        carEngine.stop();
    }
}
```

Output

```
Starting the car engine.
Displaying current speed.
Displaying current speed.
Stopping the car engine.
Displaying engine warning light.
Displaying engine warning light.
```

Applicability

1. Use the **Observer** pattern when changes to the state of one object may **require changing other objects**, and the actual set of objects is unknown beforehand or **changes dynamically**.
2. Use the pattern when some objects in your app must **observe others**, but only for a **limited time** or in specific cases.

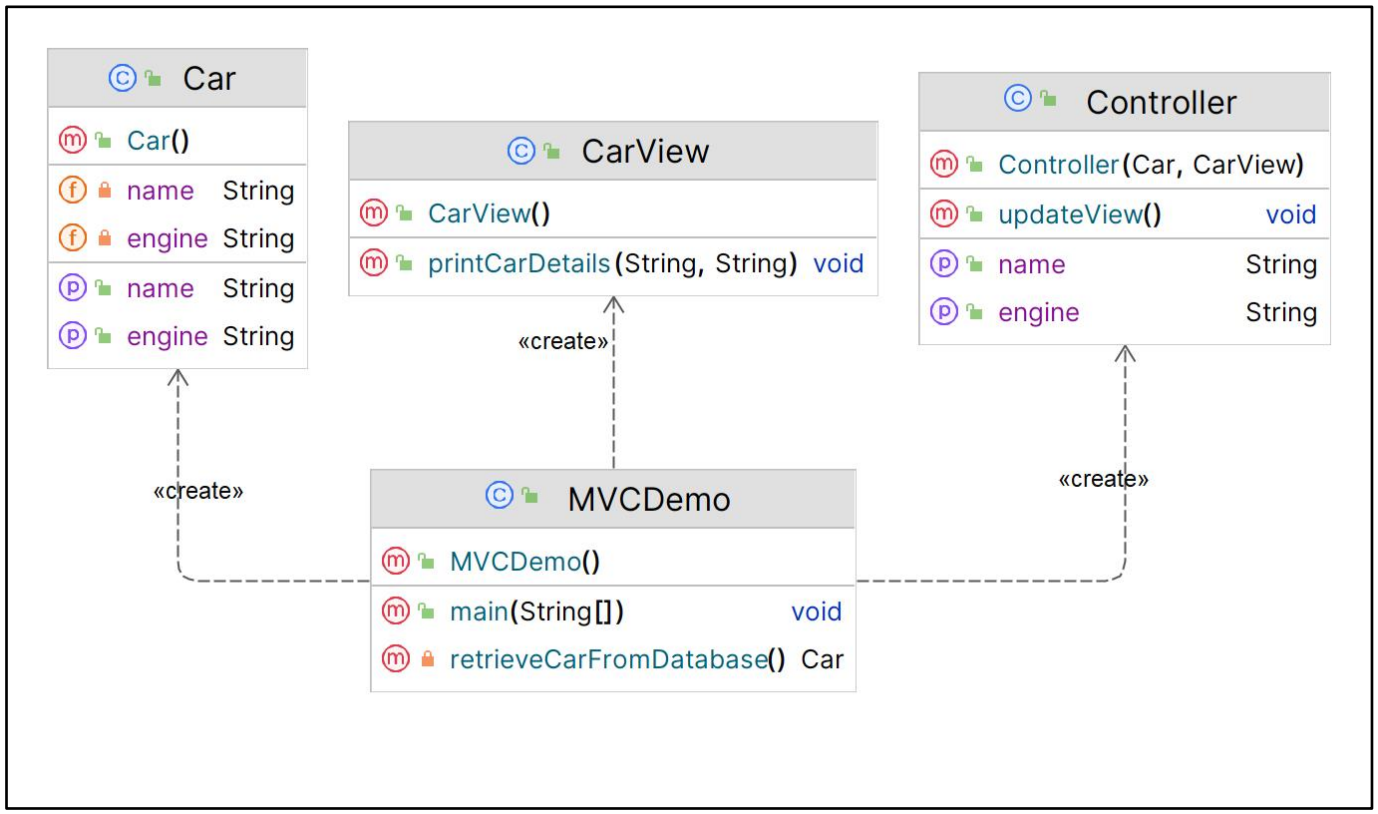
Assignment 16: MVC Design Pattern

What is MVC Design Pattern?

The **Model View Controller** 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 is mostly **UI / UX** of an application. It still needs business logic layer, maybe some service layer and data access layer.

Structure (Class Diagram)



Implementation (Code)

```
import java.sql.*;

// Model
public class Car {
    private String name;
    private String engine;

    public String getEngine() {
        return engine;
    }
    public void setEngine(String engine) {
        this.engine = engine;
    }
```

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

// View
public class CarView {
    public void printCarDetails(String carName, String carEngine) {
        System.out.println("\nCar: ");
        System.out.println("Name: " + carName);
        System.out.println("Engine: " + carEngine);
    }
}

// Controller
public class Controller {
    private Car model;
    private CarView view;

    public Controller(Car model, CarView view) {
        this.model = model;
        this.view = view;
    }
    public void setName(String name) {
        model.setName(name);
    }
    public String getName() {
        return model.getName();
    }

    public void setEngine(String engine) {
        model.setEngine(engine);
    }
    public String getEngine() {
        return model.getEngine();
    }

    public void updateView(){
        view.printCarDetails(model.getName(), model.getEngine());
    }
}
```

```
// Main Class
public class MVCDemo {
    public static void main(String[] args) {
        // Fetch student record based on his roll no from the database
        Car model = retrieveCarFromDatabase();

        // Create a view to write student details on console
        CarView view = new CarView();
        Controller controller = new Controller(model, view);
        controller.updateView();

        // Update model data
        controller.setName("Tata Nexon EV");
        controller.setEngine("1.5L");
        controller.updateView();
    }

    // Database Connection (SQL)
    private static Car retrieveCarFromDatabase() {
        Car car = new Car();

        String url = "jdbc:mysql://localhost:3306/DP_LAB?useSSL=false";
        String username = "user";
        String password = "***";

        try (Connection conn = DriverManager.getConnection(url, username, password)) {
            System.out.println("Connected to database!");

            Statement statement = conn.createStatement();
            ResultSet resultSet = statement.executeQuery("SELECT * FROM CARS");

            int count = 1;
            while (resultSet.next()) {
                String name = resultSet.getString("CAR_NAME");
                String engine = resultSet.getString("CAR_ENGINE");
                car.setName(name);
                car.setEngine(engine);
                count++;
            }
        } catch (SQLException ex) {
            System.err.println("Error connecting to database: " + ex.getMessage());
        }
        return car;
    }
}
```

Output

```
Connected to database!

Car:
Name: Chevrolet Corvette
Engine: Supercharged V8

Car:
Name: Tata Nexon EV
Engine: 1.5L
```

Database Design (SQL)

```
CREATE TABLE CARS (
   CAR_NAME VARCHAR(40),
   CAR_ENGINE VARCHAR(30)
);

INSERT INTO CARS (CAR_NAME, CAR_ENGINE)
VALUES ('Honda Civic', '1.5L'),
      ('Ford Mustang', 'V8'),
      ('Chevrolet Corvette', 'Supercharged V8');
```

	CAR_NAME	CAR_ENGINE
1	Honda Civic	1.5L
2	Ford Mustang	V8
3	Chevrolet Corvette	Supercharged V8

Applicability

1. **MVC** design pattern separates the application's concerns into **three distinct components**, making it easier to maintain and modify each component **independently**.
2. **MVC** promotes **code reusability** and **scalability**, as each component can be developed and tested separately before being **integrated** into the overall application.
3. **MVC** helps in designing **user-friendly interfaces** by providing a **clear separation** between the presentation layer and the underlying data and logic.