Aim:
To design and program the Abstract Design Pattern.

Description:
The abstract factory design pattern is a software engineering design pattern that allows for the creation of related objects without specifying their concrete classes. It's a creational design pattern that defines an interface for creating distinct products, but leaves the actual product creation to concrete factory classes.
The abstract factory design pattern is used when only the interface of a collection of objects is to be exposed, and not the implementation. It can be used by any client to create objects without any background knowledge of the system's structure, composition, and architecture.
The abstract factory design pattern works around a super-factory that creates other factories. The client code has to work with both factories and products via their respective abstract interfaces. This allows the type of factory passed to the client code, as well as the product variant that the client code receives, to be changed without breaking the actual client code.

Code:

1. Product.java
```
package Product;
interface OS{
void displayOS();
}
```

2. AndroidOS.java
```
package Product;
public class AndroidOS implements OS{
public void displayOS(){
System.out.println("Powered by Android");
}
}
```

3. IphoneOS.java
```
package Product;

public class IphoneOS implements OS {
public void displayOS() {
System.out.println("Iphone OS");
}
}
```

4. WindowsOS.java
```
package Product;
public class WindowsOS implements OS{
public void displayOS(){
System.out.println("Powered by Windows");
}
}
```

5. Mobile.java
```
package Product;
interface Mobile{
```

```
void displayInfo();
}
```

6. AppleMobile.java

```
package Product;
public class AppleMobile implements Mobile{
public void displayInfo(){
System.out.println("This is an Apple mobile");
}
}
```

7. SamsungMobile.java

```
package Product;
public class SamsungMobile implements Mobile{
public void displayInfo(){
System.out.println("This is a Samsung mobile");
}
}
```

8. NokiaMobile.java

```
package Product;
public class NokiaMobile implements Mobile{
public void displayInfo(){
System.out.println("This is a nokia mobile");
}
}
```

9. MobileFactory.java

```
package Product;
interface MobileFactory{
Mobile createMobile();
OS createOS();
}
```

10. AppleFactory.java

```
package Product;
public class AppleFactory implements MobileFactory{
public Mobile createMobile(){
return new AppleMobile();
}
public OS createOS(){
return new IphoneOS();
}
}
```

11. SamsungFactory.java

```
package Product;

public class SamsungFactory implements MobileFactory{
public Mobile createMobile(){
return new SamsungMobile();
}
public OS createOS(){
return new AndroidOS();
}
}
```

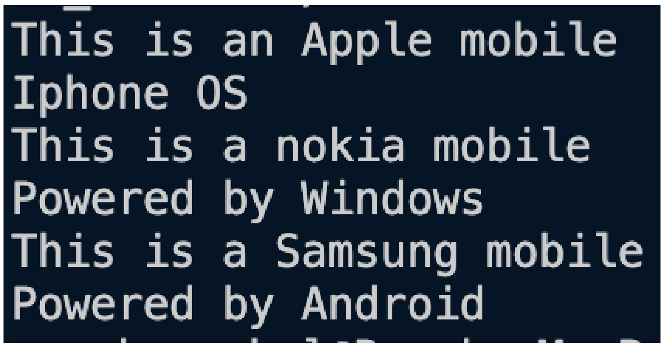
12. NokiaFactory.java

```
package Product;
public class NokiaFactory implements MobileFactory{
public Mobile createMobile(){
return new NokiaMobile();
}
public OS createOS(){
return new WindowsOS();
}
}
```

13. test.java

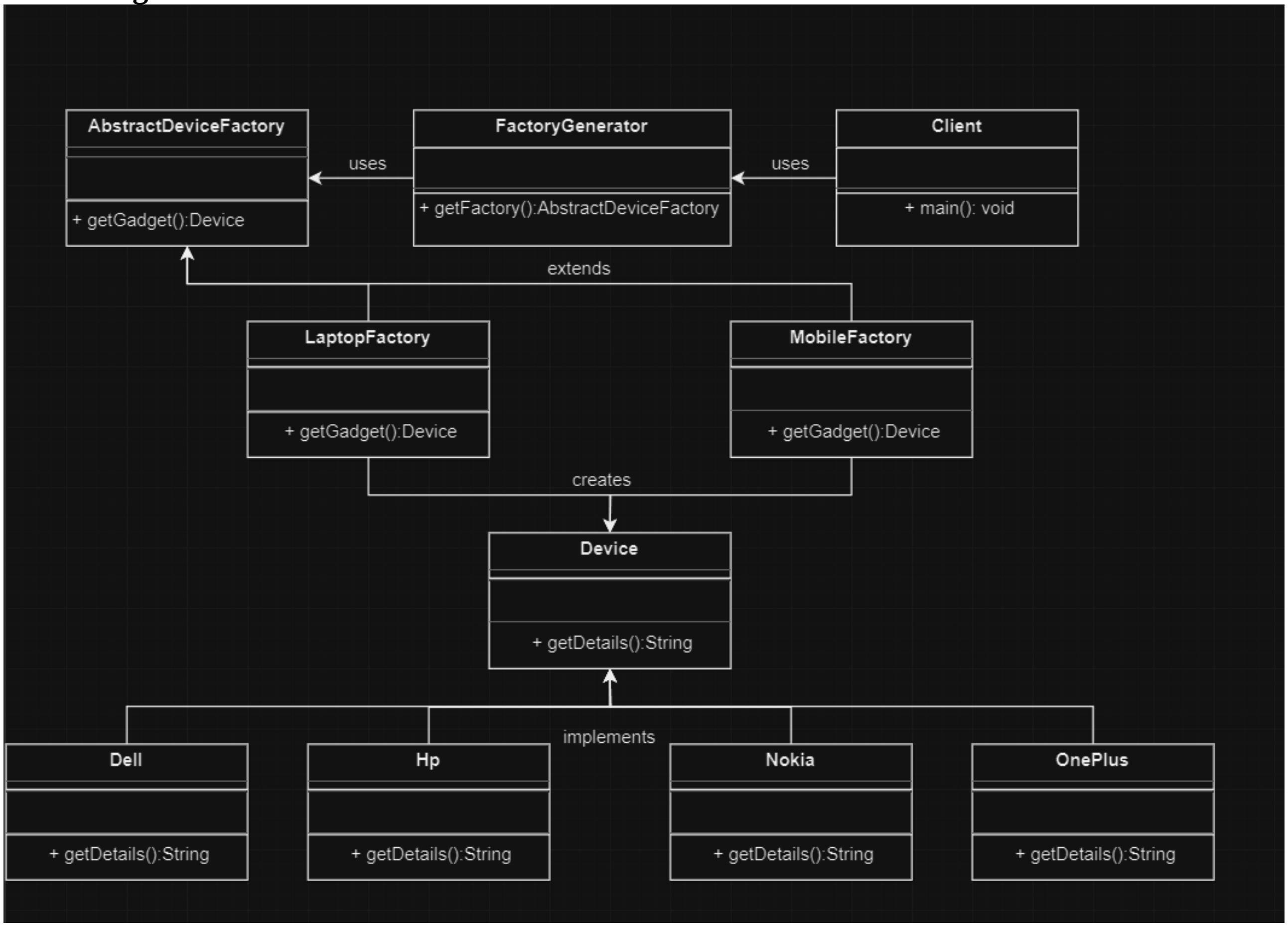
```
package Product;
class test{
public static void main(String[] args) {
MobileFactory factory;
Mobile mobile;
OS os;
factory = new AppleFactory();
mobile = factory.createMobile();
os = factory.createOS();
mobile.displayInfo();
os.displayOS();
factory = new NokiaFactory();
mobile = factory.createMobile();
os = factory.createOS();
mobile.displayInfo();
os.displayOS();
factory = new SamsungFactory();
mobile = factory.createMobile();
os = factory.createOS();
mobile.displayInfo();
os.displayOS();
}
}
```

Output:



```
This is an Apple mobile
Iphone OS
This is a nokia mobile
Powered by Windows
This is a Samsung mobile
Powered by Android
```

Class Diagram:



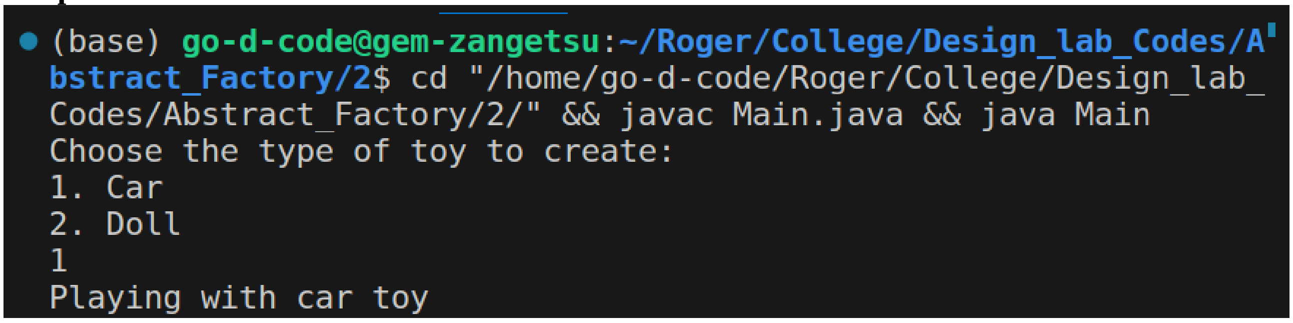
Aim:
Implementing Abstract Factory pattern for flexible creation of toy families, ensuring extensibility and scalability without tight coupling to specific implementations.

Description:
The provided code demonstrates the Abstract Factory design pattern, facilitating the creation of families of related objects (toys) without directly specifying their concrete classes. Through interfaces for both products (toys) and factories, it enables flexible, extensible, and scalable object creation, promoting modular and maintainable code. The client can create toys without tight coupling to specific implementations, enhancing code flexibility and facilitating easy addition of new toy types while upholding principles of encapsulation and abstraction.

Code:

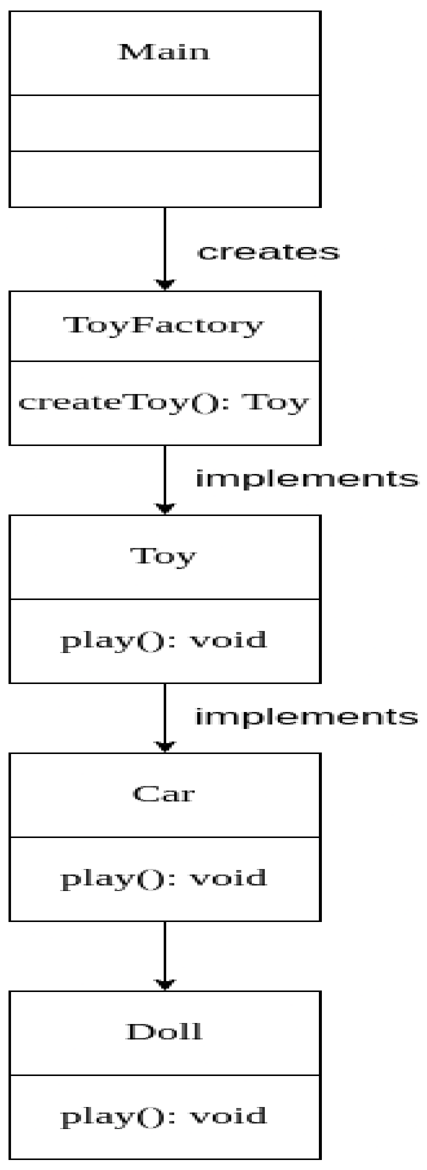
1. Main.java
```java
import java.util.Scanner;

// Abstract Product: Toy
interface Toy {
    void play();
}

// Concrete Products: Car and Doll
class Car implements Toy {
    @Override
    public void play() {
        System.out.println("Playing with car toy");
    }
}

class Doll implements Toy {
    @Override
    public void play() {
        System.out.println("Playing with doll toy");
    }
}

// Abstract Factory: ToyFactory
interface ToyFactory {
    Toy createToy();
}

// Concrete Factories: CarFactory and DollFactory
class CarFactory implements ToyFactory {
    @Override
    public Toy createToy() {
        return new Car();
    }
}

class DollFactory implements ToyFactory {
    @Override
    public Toy createToy() {
        return new Doll();
    }
}

// Client class to demonstrate usage
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Choose the type of toy to create:");
        System.out.println("1. Car");
        System.out.println("2. Doll");
        int choice = scanner.nextInt();

        ToyFactory toyFactory = null;

        switch (choice) {
            case 1:
                toyFactory = new CarFactory();
                break;
            case 2:
                toyFactory = new DollFactory();
                break;
            default:
                System.out.println("Invalid choice. Exiting...");
                System.exit(1);
        }

        // Create toy using the selected factory
        Toy toy = toyFactory.createToy();
        toy.play();

        scanner.close();
    }
}
```

Output:



```
● (base) go-d-code@gem-zangetsu:~/Roger/College/Design_lab_Codes/A
  bstract_Factory/2$ cd "/home/go-d-code/Roger/College/Design_lab_
  Codes/Abstract_Factory/2/" && javac Main.java && java Main
  Choose the type of toy to create:
  1. Car
  2. Doll
  1
  Playing with car toy
```

Class Diagram:


Main
creates
ToyFactory
createToy(): Toy
implements
Toy
play(): void
implements
Car
play(): void
Doll
play(): void