4. Abstract Factory Design Pattern

The abstract factory method is a design pattern that falls under the category of creational design patterns. It provides an interface for creating families of related or dependent objects without specifying their concrete classes. The main idea behind the abstract factory pattern is to abstract the creation of objects and allow a client to use a family of objects without knowing their specific implementations.

Here are the key components of the abstract factory pattern:

1. Abstract Factory Interface: This interface declares a set of creation methods for each type of product in a family of related products. The abstract factory itself does not provide the concrete implementations but defines the structure of the product creation.

2. Concrete Factories: These are the concrete implementations of the abstract factory interface. Each concrete factory produces a family of products that are related or compatible with each other.

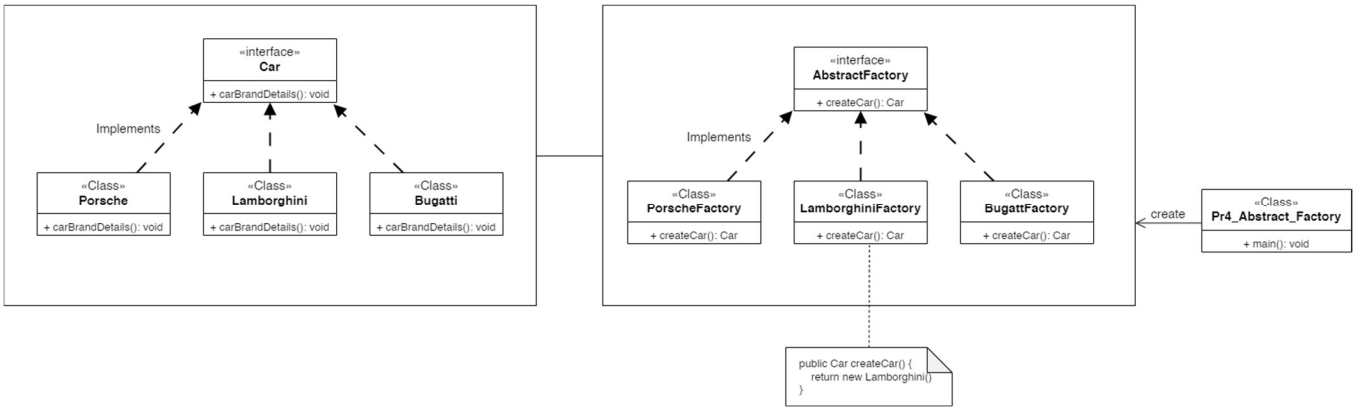
3. Abstract Product Interfaces: These interfaces declare the methods that must be implemented by the concrete products created by the abstract factories.

4. Concrete Products: These are the actual product implementations that conform to the abstract product interfaces.

5. Client Code: The client code uses the abstract factory to create and work with products without being aware of their specific implementations.

The abstract factory pattern is useful when you need to ensure that the created objects are compatible and belong to a specific family or when the system is configured to work with multiple families of objects interchangeably. It promotes flexibility and helps in avoiding dependencies on concrete classes, making it easier to introduce new families of products without modifying the client code.

UML Class Diagram:



Implementation:

```java
import java.util.*;

// Abstract product interface
interface Car {
    // Create an interface implemented by classes to print all brand details.
    public void carBrandDetails();
}

// Concrete product implementations
class Porsche implements Car {
    // Sub class that implements "Car" interface and "carBrandDetails" method
    public void carBrandDetails() {
        // Print Details of the Car Brand.
        System.out.println("This is Porsche.");
        System.out.println("Founder: Ferdinand Porsche");
        System.out.println("Founded: 25 April 1931, Stuttgart, Germany");
    }
}

class Bugatti implements Car {
    // Sub class that implements "Car" interface and "carBrandDetails" method
    public void carBrandDetails() {
        // Print Details of the Car Brand.
        System.out.println("This is Bugatti.");
        System.out.println("Founder: Ettore Bugatti");
        System.out.println("Founded: 1909, Molsheim, France");
    }
}
```

```java
class Lamborghini implements Car {
    // Sub class that implements "Car" interface and "carBrandDetails" method
    public void carBrandDetails() {
        // Print Details of the Car Brand.
        System.out.println("This is Lamborghini.");
        System.out.println("Founder: Ferruccio Lamborghini");
        System.out.println("Founded: May 1963, Sant'Agata Bolognese, Italy");
    }
}

interface AbstractFactory {
    Car createCar();
}

class PorscheFactory implements AbstractFactory {

    @Override
    public Car createCar() {
        return new Porsche();
    }

}

class BugattiFactory implements AbstractFactory {

    @Override
    public Car createCar() {
        return new Bugatti();
    }

}

class LamborghiniFactory implements AbstractFactory {

    @Override
    public Car createCar() {
        return new Lamborghini();
    }

}

// Main Driver Class
public class Pr4_Abstract_Factory {

    // Main Method
```

```
    public static void main(String[] args) {
        // Create Scanner Object with Input stream
        Scanner sc = new Scanner(System.in);

        // Display Prompt to request User Input Data
        System.out.println("Enter the Car Brand Name:");

        // Input the user data using scanner object and "nextLine()" method and
store in
        // the "input" string.
        String input = sc.nextLine();

        AbstractFactory carAbstractFactory; // Create Reference of the Abstract
Factory

        // Switch case to create new factory object based on input
        switch (input.toLowerCase()) {
            case "porsche":
                carAbstractFactory = new PorscheFactory();
                break;
            case "bugatti":
                carAbstractFactory = new BugattiFactory();
                break;
            case "lamborghini":
                carAbstractFactory = new LamborghiniFactory();
                break;
            default:
                System.out.println("No Object Created! Class with brand name '" +
input + "' doesn't exist!");
                sc.close();
                return;
        }

        // use the reference to call the create car method of the factory
        Car car = carAbstractFactory.createCar();
        car.carBrandDetails(); // Print the car brand details of the car.

        // Close the Scanner input stream to prevent data leaks.
        sc.close();
    }
}
```

Output:

```
Enter the Car Brand Name:
Porsche
This is Porsche.
Founder: Ferdinand Porsche
Founded: 25 April 1931, Stuttgart, Germany
```