1. LAB A – ABSTRACT FACTORY

Purpose

Consider a car factory manufacturing 2 types of cars. In this lab we will implement the abstract factory design pattern by creating an abstract Factory class and two concrete factories.

Clients will communicate with the abstract car factory, asking it to create car parts.

The car factory will delegate the request to a concrete factory so that the proper car part is constructed.

Lab Scenario (WHAT to do)	We want to develop an abstract factory class and 2 concrete factories (one for each car type) When the abstract factory is instantiated, it will choose a concrete factory class based on the class name loaded from a property file Creational requests will be delegated to the concrete factory loaded The concrete factory will assemble the car using the classes in the package patterns.creational.carparts
Implementation Steps (HOW to do it)	**Code the AbstractCarFactory class :** Implement 1 static (int) counter for car parts – all car parts should have a unique id Implement 2 abstract methods : **public abstract** Hood createHood(String color); **public abstract** Wheel createWheel(); **Code the FamilyCarFactory** Implement the createXXX methods so that they return FamilyXXX Parts and use the counter logic to assign a unique id to each created part **Code the MiniVanFactory** Implement the createXXX methods so that they return MiniVanXXX Parts and use the counter logic to assign a unique id to each created part **Code the FactoryManager** This class will load the concrete factory class value to use, instantiate It and return it Implement it as a singleton **Test it** FactoryManager fm = FactoryManager.*getInstance*(); AbstractCarFactory carFactory = fm.getFactory(); Wheel wheel = carFactory.createWheel(); System.*out*.println(wheel); * Changing the class name in the property file should change the type of wheel created

Review of Abstract Factory

Four classes will be created :

1. AbstractCarFactory will serve as the base for concrete factory classes

2. FamilyCarFactory a concrete factory creating family car parts

3. MiniVanCarFactory a concrete factory creating minivan car parts

4. FactoryManager wraps the factory logic and is the client access class

One property file will be created:

The property file will contain the concrete factory class name under the 'cars.factory.name' property key. Classes representing the individual car parts are already coded, and can be found in package `patterns.creational.carparts`

1.1.1. Create a new class – AbstractCarFactory

This class serves as the base for concrete car factories. It makes the application flexible, new concrete factories can be added later on.

1. Create the class
2. Implement a private static int `partsCounter` as member variable
3. Implement a public static int `getNextId()` which returns the current counter value and increases it by one. Concrete factories will later call this method when they create concrete parts. Each part should have a unique id
4. Code an abstract Hood createHood(String color) method
5. Code an abstract Wheel createWheel() method
6. Code an abstract Engine createEngine() method

The abstract methods will be implemented by the concrete factories.

1.1.2. Create a new class FamilyCarFactory

1. This factory will implement the AbstractCarFactory. The create methods will return Family Car Parts.
2. Create the class
3. Extend AbstractCarFactory
4. Implement the createHood(String color) method
5. Create a family car hood object
6. Call the abstract factory getNextId() method for a unique id
7. Assign the unique id to the hood created in item a
8. Return the hood
9. Implement the createWheel() method
10. Create a family car wheel object
11. Call the abstract factory getNextId() method for a unique id
12. Assign the unique id to the wheel created in item a
13. Return the wheel

1.1.3. Create a new class MinivanCarFactory

This factory will implement the AbstractCarFactory, and the create methods will return Minivan Car Parts

1. Create the class
2. Extend AbstractCarFactory
3. Implement the createHood(String color) method
4. Create a minivan car hood object
5. Call the abstract factory getNextId() method for a unique id
6. Assign the unique id to the hood created in item a
7. Return the hood
8. Implement the createWheel() method
9. Create a minivan car wheel object
10. Call the abstract factory getNextId() method for a unique id
11. Assign the unique id to the wheel created in item a
12. Return the wheel

1.1.4. Create a properties file

The file will contain the class name of 1 concrete factory. The factory manager will load the file and use it to decide which concrete factory to use (see item 1.2.5).

1. Create a text file abstractfactory.properties. The File should be placed in the same folder as the factory java files and should contain:

```
cars.factory.name=patterns.creational.abstractfactory.MiniVanFactory
```

Changing the class name in the file will affect the concrete factory loaded by the factory manager class, which is coded next.

1.1.5. Create a new class FactoryManager

This class will be the client access point. The class will load the properties file, decide which concrete factory class to use. It then loads the class and delegates create request to it.

1. Create the class
2. Implement the following member variables

```
private AbstractCarFactory factory;
```
this will store the concrete factory
```
public static FactoryManager instance;
```
will be used making the FactoryManager a singleton

3. Implement the constructor as private

a. Load the property file

```
 ResourceBundle props = ResourceBundle.getBundle(
"patterns.creational.abstractfactory.abstractfactory");

String factoryName = props.getString("cars.factory.name");
```

b. Initialize the concrete factory

```
Object obj = Class.forName(factoryName).getDefaultConstructor(new Class<?>[]
{}).newInstance(new Object[]{});
factory = (AbstractCarFactory) obj;
```

4. Implement a getInstance() method

```
public synchronized static FactoryManager getInstance() {

    if (instance == null) {

       instance = new FactoryManager();

    }

  return instance;

}
```

5. Implement the main testing code

```
public static void main(String[] args) {
       FactoryManager fm = FactoryManager.getInstance();
       AbstractCarFactory carFactory = fm.getFactory();
       Wheel wheel = carFactory.createWheel();
       System.out.println(wheel);
}
```

Check that changing the class name in the property file, changes the classes created. If a minivan car factory is defined in the properties file, minivan wheel should be created, whereas if a family car factory is defined, a different wheel object is created.

Note: The solution for this exercise is available in the ‘solutions’ directory