Patrón Fábrica Abstracta

Julio Ariel Hurtado Alegría

Ingeniería de Software II

2015

Contenido

- Descripción General
- Problema
- Solución
- Participantes
- Ejemplo: Fábrica de Carros
- Ejemplo: RelojVisual

Descripción General

- El patrón *Abstract Factory* busca definir una interface abstracta para crear una familia de objetos relacionados que son dependientes de una variedad de soluciones concretas (Gamma, Helm, Johnson, & Vlissides, 1994).

- El patrón *Abstract Factory* es útil cuando un objeto cliente desea crear un conjunto de instancias de clases relacionadas y dependientes, sin tener que conocer cuales clases específicas y concretas son instanciadas, manteniendo las restricciones propias de la familia de objetos.

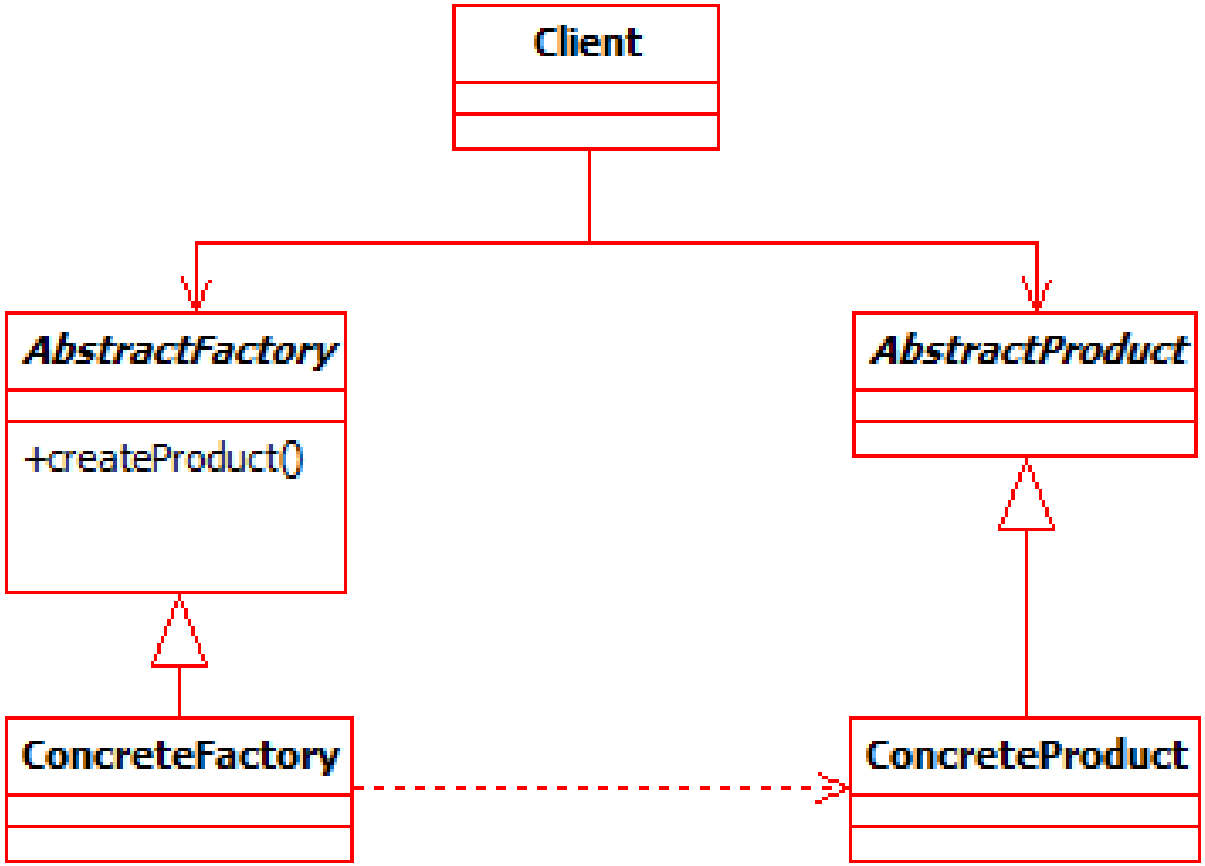
Problema

- Uno de los problemas críticos al momento de mantener el software es su capacidad para ser modificado. Para ello normalmente se utiliza los principios de diseño abierto-cerrado e inversión de dependencias (Martin, 2000).
- Una forma de lograrlo es definir una capa de servicios abstractos que define la interface hacia un módulo de nivel superior. Normalmente esta capa de servicios abstractos, debe ser instanciada, a partir de una implementación concreta, como un conjunto de objetos relacionados con restricciones propias de la implementación.
- El problema radica en cómo mantener la independencia de la implementación para el módulo de nivel superior a la capa de servicios abstracta, aún durante la instanciación de los objetos que la conforman dependa de elementos abstractos y no de elementos concretos.

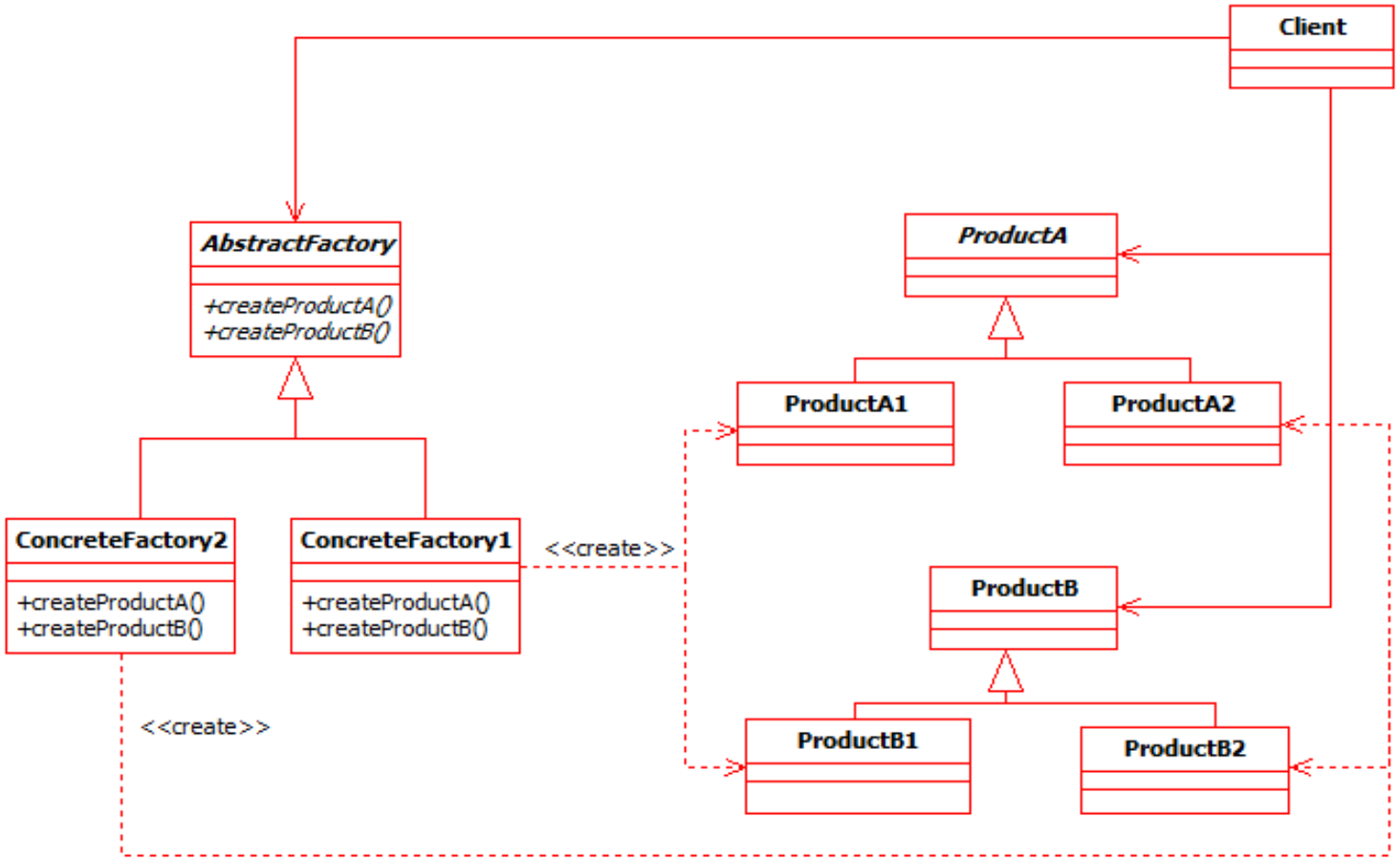
Solución

- En términos simples, un *abstract factory* es una interface o clase abstracta que suministra una interface para producir una familia de objetos.
- Una fábrica abstracta ayuda a evitar esta duplicación mediante el suministro de la interface necesaria para la creación de tales instancias. Diferentes fábricas concretas implementan esta interface.
    - Familias de clases relacionadas y dependientes entre sí.
    - Una clase *abstract factory* que define una interface que incluye métodos abstractos para crear cada tipo de objeto en la familia.
    - Un grupo de clases *concrete factory* que implementan en forma concreta la interface suministrada por la clase *abstract factory*.

Solución


Client
AbstractFactory
+createProduct()
AbstractProduct
ConcreteFactory
ConcreteProduct

Solución

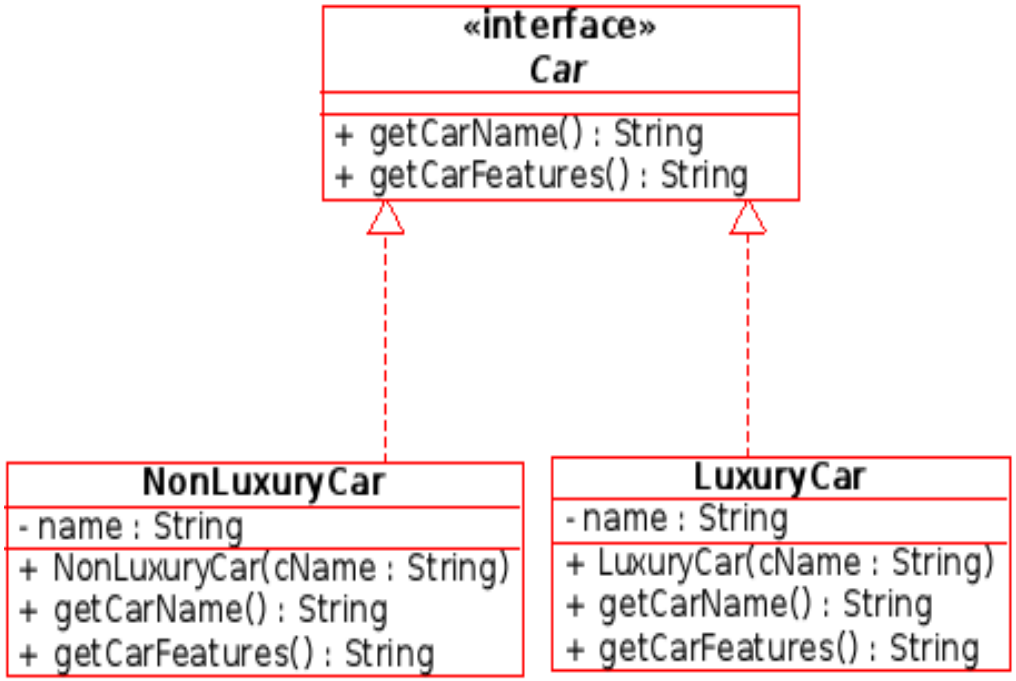

Client
AbstractFactory
+createProductA()
+createProductB()
ProductA
ProductA1
ProductA2
ConcreteFactory2
+createProductA()
+createProductB()
ConcreteFactory1
+createProductA()
+createProductB()
<<create>>
ProductB
<<create>>
ProductB1
ProductB2

Participantes

- *AbstractFactory* : Una interface con operaciones *create* para cada uno de los productos abstractos.
- *Factory1, Factory2* : Implementaciones de todas las operaciones de creación de *AbstractFactory.*
- *AbstractProduct* : Una interface para cada tipo de producto con sus propias operaciones.
- *ProductA1, ProductA2, ProductB1, ProductB2* : Clases que implementan la interface *AbstractProduct* y define objetos de productos que serán creados por las correspondientes fábricas.
- *Client* : Una clase que accede solamente a la *AbstractFactory* y a las interfaces *AbstractProduct.*

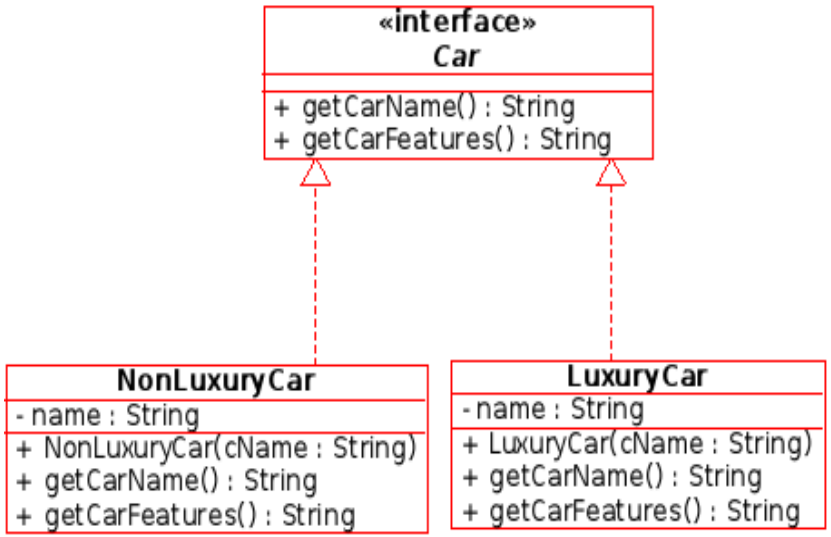
Ejemplo Fábrica de Registros de Carros

- Considere la siguiente Jerarquía de Registros de Carros

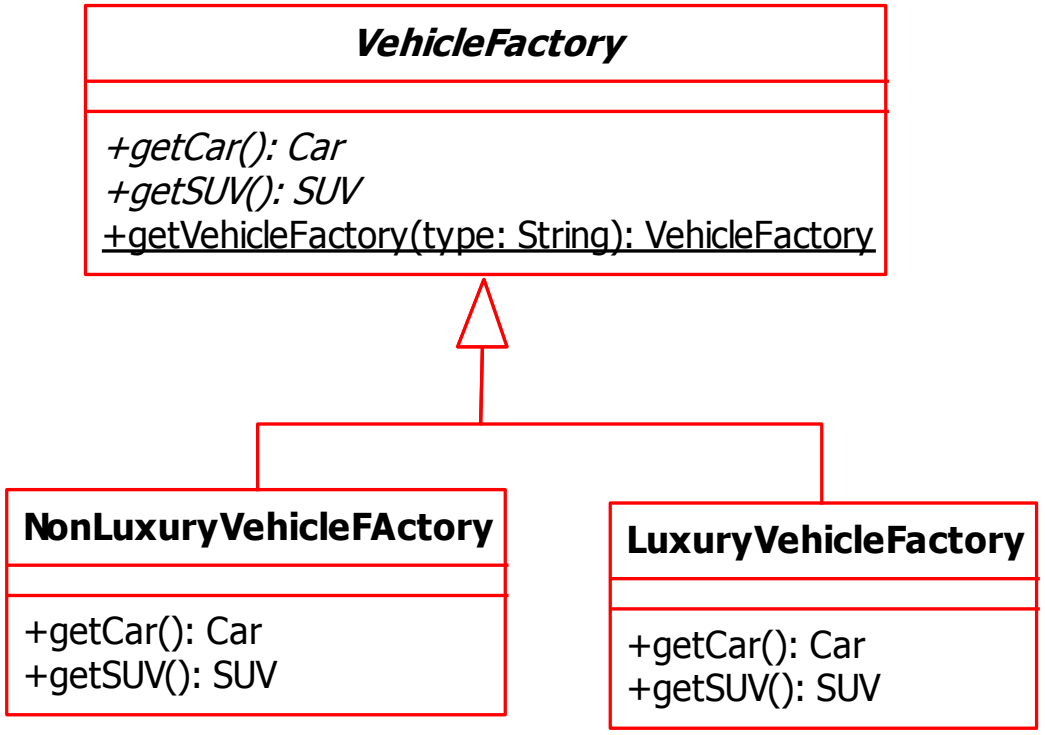


Ejemplo: Fábrica de Carros

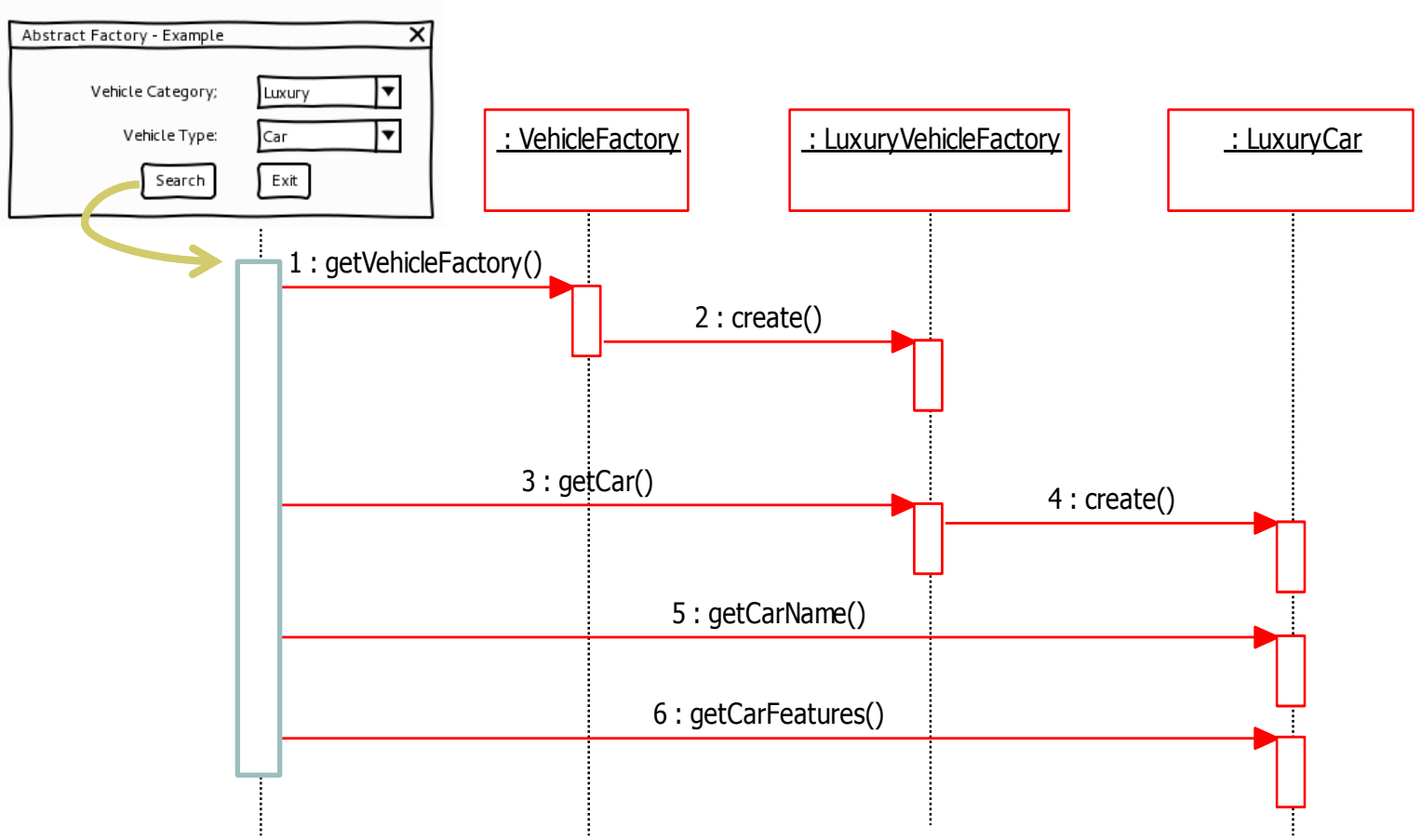
- Considere la necesidad de diseñar parte de una aplicación que consulte las características de diferentes tipos de vehículos que están categorizados como luxury (lujo) o nonluxury. Por simplicidad, consideraremos dos tipos de vehículos: Cars y SUVs (Sport Utility Vehicle - todoterreno).



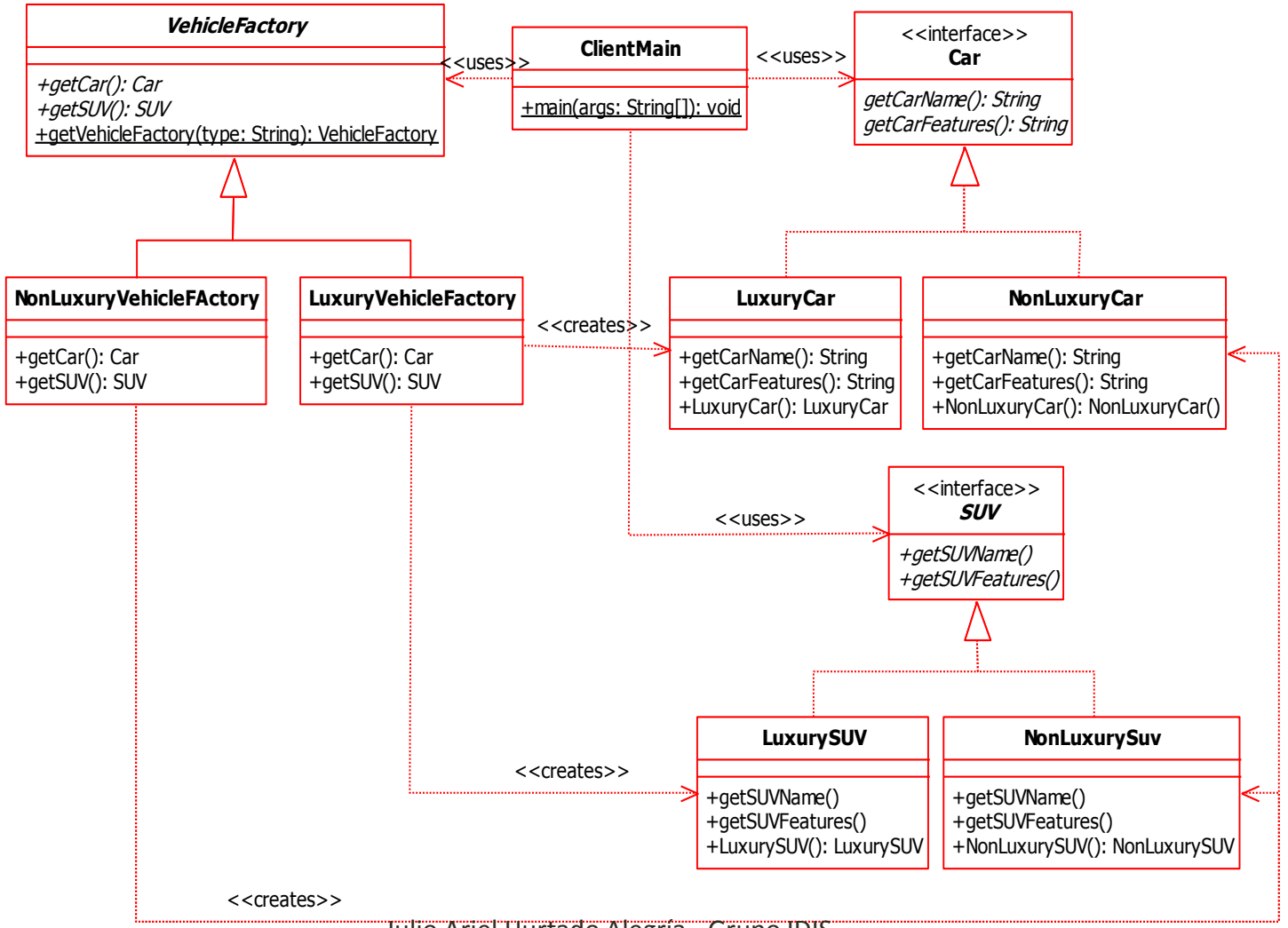
Fábrica de Carros



Fábrica de Registros de Carros



Fábrica de Registros de Carros



Producto Abstracto Carro

```
1. package patrones.abstractfactory.ejm1;
2. // Define la interface del producto abstracto
   Car
3. public interface Car {
4.         public String getCarName();
5.         public String getCarFeatures();
6. }
```

Producto Concreto

```
1.  package patrones.abstractfactory.ejm1;
2.  // Define el product concreto LuxuryCar
3.  public class LuxuryCar implements Car {
4.      private String name;
5.
6.      public LuxuryCar(String cName) {
7.          name = cName;
8.      }
9.      public String getCarName() {
10.         return name;
11.     }
12.     public String getCarFeatures() {
13.         return "Luxury Car Features ";
14.     }
15. }
```

Producto Concreto Carro

```
1.  package patrones.abstractfactory.ejm1;
2.  // Define el producto concreto NonLuxuryCar

3.  public class NonLuxuryCar implements Car {
4.      private String name;
5.
6.      public NonLuxuryCar(String cName) {
7.          name = cName;
8.      }
9.
10.     public String getCarName() {
11.         return name;
12.     }
13.
14.     public String getCarFeatures() {
15.         return "Non-Luxury Car Features ";
16.     }
17. }
```

Producto Abstracto SUV

```
1. package patrones.abstractfactory.ejm1;
2. // Define la interface del producto abstracto SUV
3. public interface SUV {
4.     public String getSUVName();
5.     public String getSUVFeatures();
6. }
```

Producto Concreto SUV

```
1.  package patrones.abstractfactory.ejm1;
2.  // Define el producto concreto LuxurySUV

3.  public class LuxurySUV implements SUV {
4.      private String name;
5.
6.      public LuxurySUV(String sName) {
7.          name = sName;
8.      }
9.
10.     public String getSUVName() {
11.         return name;
12.     }
13.
14.     public String getSUVFeatures() {
15.         return "Luxury SUV Features ";
16.     }
17. }
```

Producto Concreto SUV

```
1.  package patrones.abstractfactory.ejm1;
2.  // Define el producto concreto NonLuxurySUV
3.  public class NonLuxurySUV implements SUV {
4.      private String name;
5.
6.      public NonLuxurySUV(String sName) {
7.          name = sName;
8.      }
9.
10.     public String getSUVName() {
11.         return name;
12.     }
13.
14.     public String getSUVFeatures() {
15.         return "Non-Luxury SUV Features ";
16.     }
17. }
```

Fábrica de Registro de Carros

```
1.  package patrones.abstractfactory.ejm1;
2.  // Define la interface de fabricación abstracta de vehículos VehicleFactory

3.  public abstract class VehicleFactory {
4.      public static final String LUXURY_VEHICLE = "Luxury";
5.      public static final String NON_LUXURY_VEHICLE = "Non-Luxury";
6.          // Define un método de creación de objetos tipo Car en forma abstracta
7.      public abstract Car getCar();
8.          // Define un método de creación de objetos tipo Suv en forma abstracta
9.      public abstract SUV getSUV();
10. // Método de clase que determina la fábrica concreta del tipo suministrado en  la variable type
11.     public static VehicleFactory getVehicleFactory(String type) {
12.                 if (type.equals(VehicleFactory.LUXURY_VEHICLE))
13.                         return new LuxuryVehicleFactory();
14.                 if (type.equals(VehicleFactory.NON_LUXURY_VEHICLE))
15.                         return new NonLuxuryVehicleFactory();
16.                 return new LuxuryVehicleFactory();
17.     }
18. }
```

Fábricas

```
1.  package patrones.abstractfactory.ejm1;
2.  // Define la fábrica concreta de vehículos de lujo implementando los métodos de VehicleFactory

3.  public class LuxuryVehicleFactory extends VehicleFactory {
4.      public Car getCar() {
5.          return new LuxuryCar("L-C");
6.      }
7.
8.      public SUV getSUV() {
9.          return new LuxurySUV("L-S");
10.     }
11. }
```

•

```
1.  package patrones.abstractfactory.ejm1;
2.  // Define la fábrica concreta de vehículos de no lujo implementando los métodos de VehicleFactory

3.  public class NonLuxuryVehicleFactory extends VehicleFactory {
4.      public Car getCar() {
5.          return new NonLuxuryCar("NL-C");
6.      }
7.
8.      public SUV getSUV() {
9.          return new NonLuxurySUV("NL-S");
10.     }
11. }
```

Clase Cliente

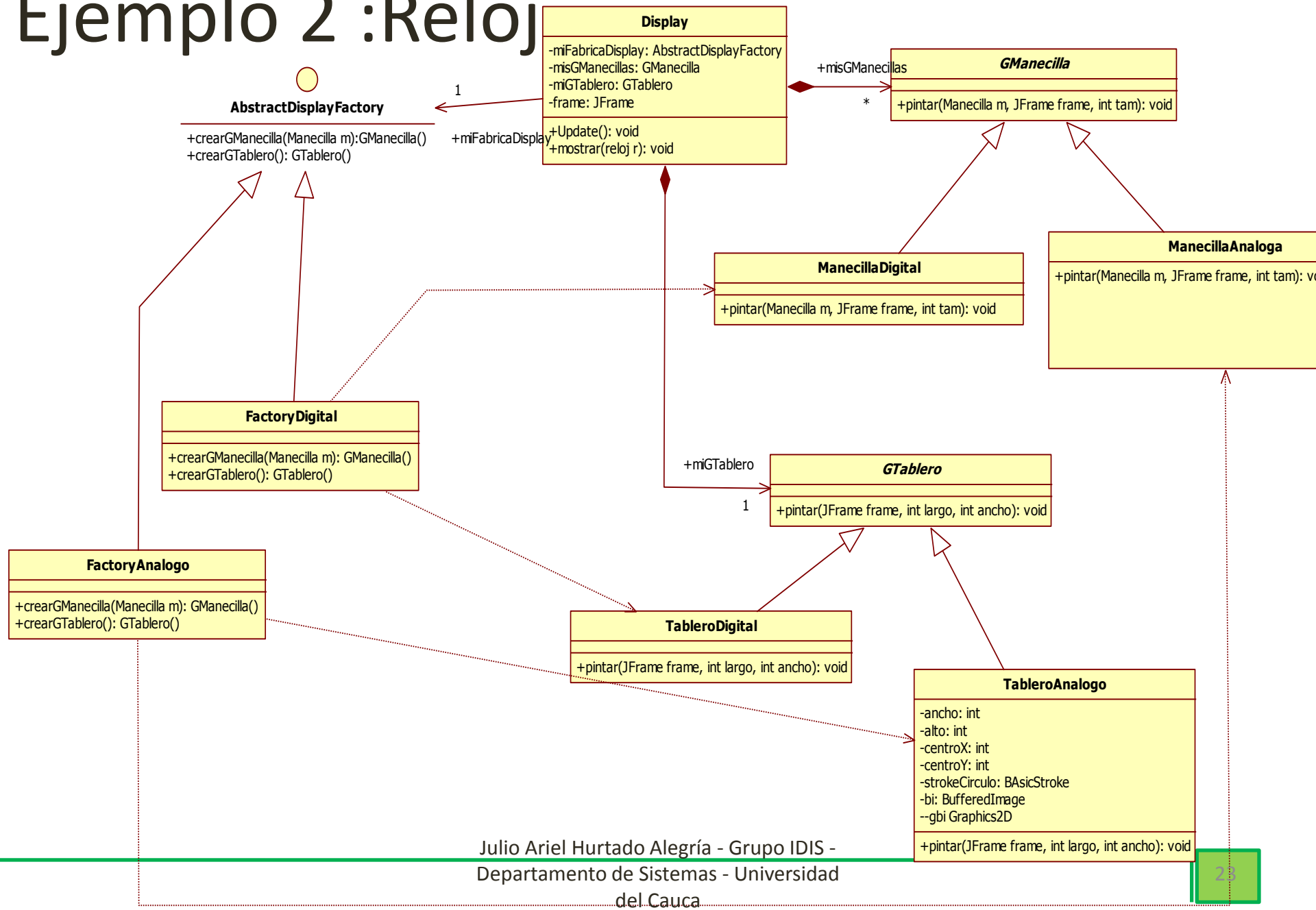
```
package patrones.abstractfactory.ejm1;
// Define la clase ClienteMain
public class ClienteMain {
        public static void main(String[] args) {
                String vhCategory = "Non-Luxury"; // o se puede colocar Luxury
                String vhType = "Suv"; // o se puede colocar Car
                String searchResult = "";
                // Obtiene la fábrica concreta en vf
                VehicleFactory vf = VehicleFactory.getVehicleFactory(vhCategory);

                // Obtiene un Car de lujo o no lujo dependiendo de la fábrica y muestra en pantalla la
            //información
            Car c = vf.getCar();
                searchResult = "Name: " + c.getCarName() + " Features: "
                                                                + c.getCarFeatures();
            System.out.println(searchResult);

             // Obtiene un SUV de lujo o no lujo dependiendo de la fábrica y muestra en pantalla la
            //información

            SUV s = vf.getSUV();
                                        searchResult = "Name: " + s.getSUVName() + " Features: "
                                                                + s.getSUVFeatures();
                System.out.println(searchResult);
        }
}
```

Ejemplo 2 :Reloj



Implementación

- Ver modelado e implementación en: http://artemisa.unicauca.edu.co/~ahurtado/ingsoftware/relojFabricaVisual.rar

- Tenga en cuenta que la clase ManecillaVisual y ManecillaDigital no cumplen con el principio de una sola responsabilidad, por lo que tienen la programación para desplegar el complejo completo de manecillas, es decir la hora.