UNIVERSITÉ CÔTE D'AZUR

UTILISATION DU PATTERN FACTORY

@.fR Frédéric RALLO

CLASSIFICATION DES PATRONS DE CONCEPTION

Création

- Comment un objet peut être créé
- Indépendance entre la manière de créer et la manière d'utiliser

Structure

- Comment les objets peuvent être combinés
- Indépendance entre les objets et les connexions

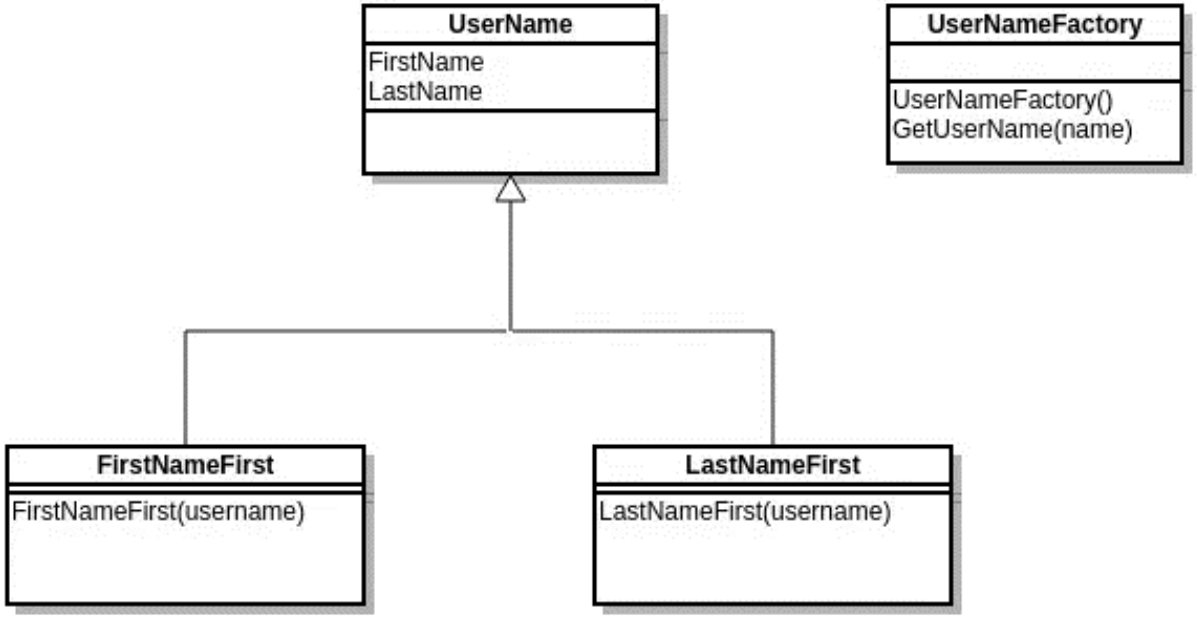
Comportement

- Comment les objets communiquent
- Encapsulation de processus (ex : observer/observable)

UN PATRON, UNE USINE !

- **Motif d'architecture logicielle**
- **Assure que les objets de forte valeur métier ne dépendent pas des objets de faible valeur métier**
- **Permet de séparer la création d'objets dérivant d'une classe mère de leur utilisation**
- **Il existe 3 variantes principales**
  - Simple Factory
  - Factory Method
  - Abstract Factory

SIMPLE FACTORY

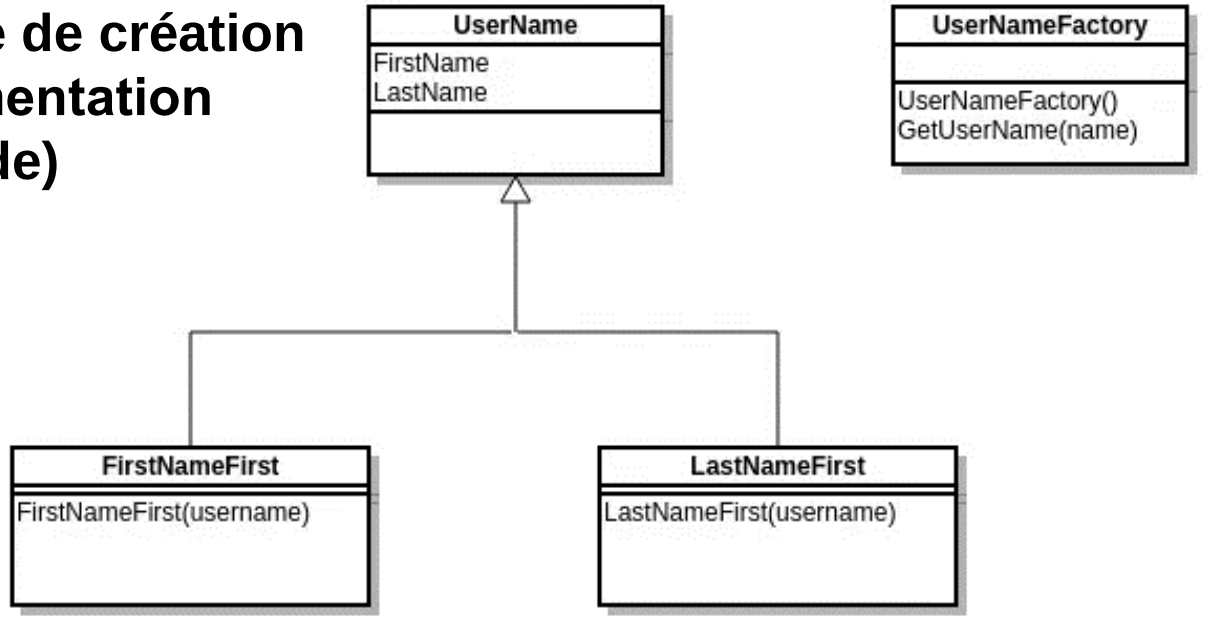


SIMPLE FACTORY

A quoi ça sert ?

- **S'assurer que l'on dispose de tous les sous-éléments (objets) dont on a besoin**

- **Dissocier la logique de création de celle de l'implémentation (réutilisation du code)**



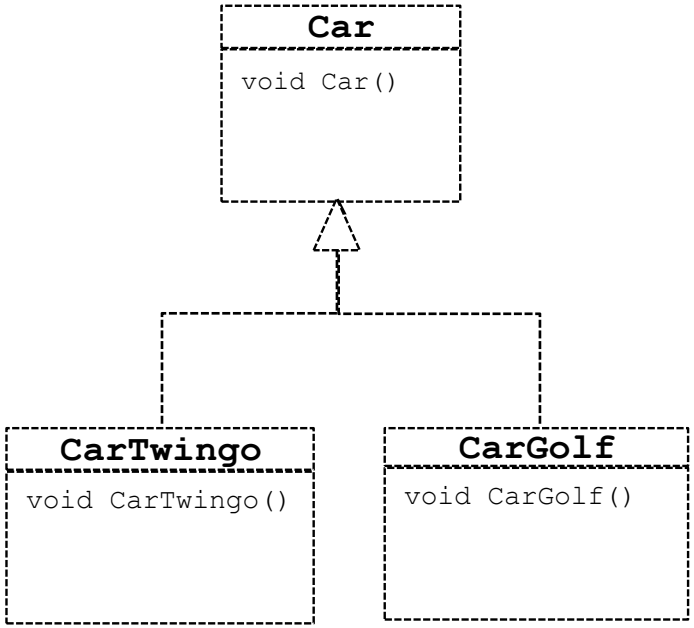
SIMPLE FACTORY

Exemple

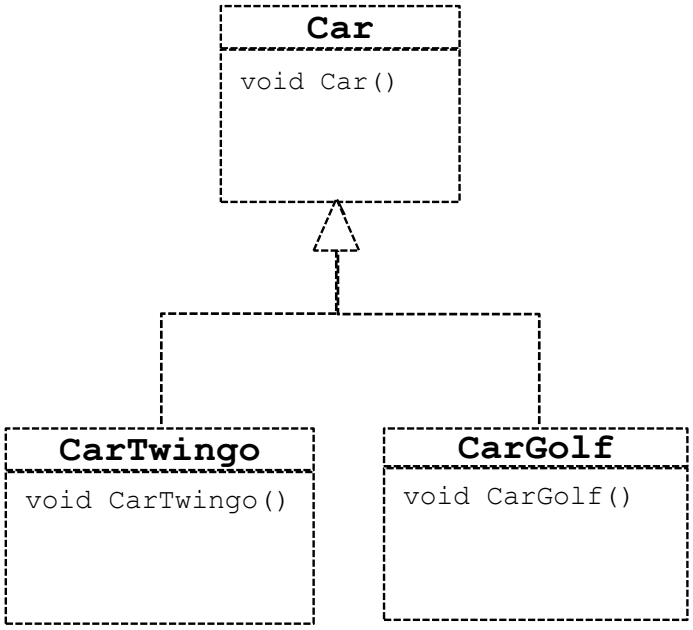
- **Je veux créer un objet Voiture**
- **Je veux m'assurer de disposer de tous les éléments de la voiture**
- **Je propose d'écrire quelque chose du type**

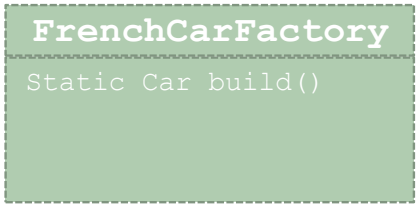
```
Car car = new Car();
```

- **MAIS… au fil du temps, je vais vouloir changer ce code et créer un objet CarTwingo et CarGolf à la place de Car**
- **Une solution naïve consiste à écrire quelque chose du type** `Car car = new CarTwingo();`
- **Je vais apporter des modifications partout où j'avais créé un objet Car et je sais que je serai amené à créer d'autres sous-type de Car**



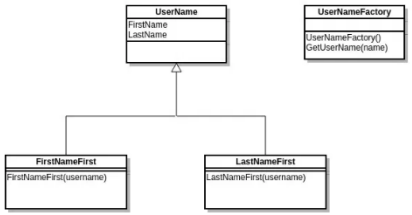
Exemple de Simple Factory



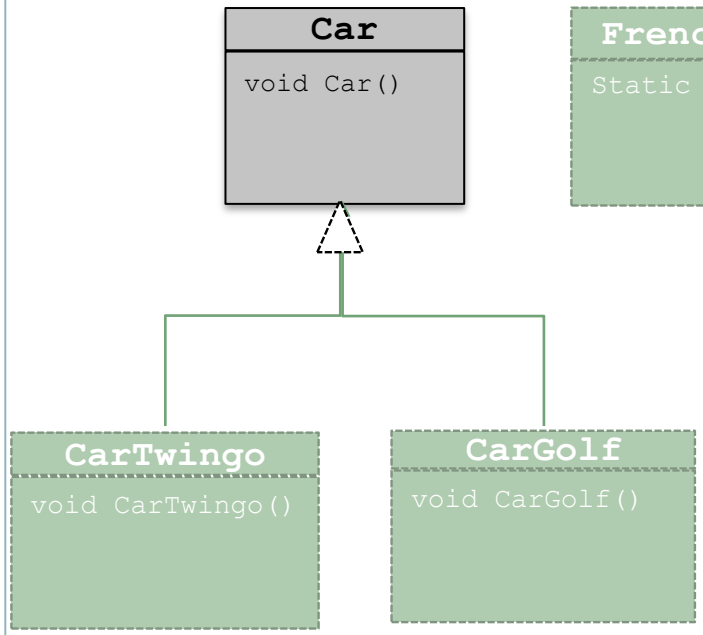


SOLUTION :

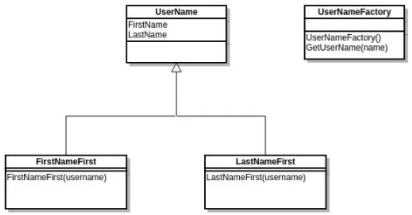
- **Dissocier la logique de création de celle de l'implémentation (réutilisation du code)**
- **Lorsque le moment sera venu de modifier ou de créer un nouveau sous-type de Car, il me suffira de modifier FrenchCarFactory**



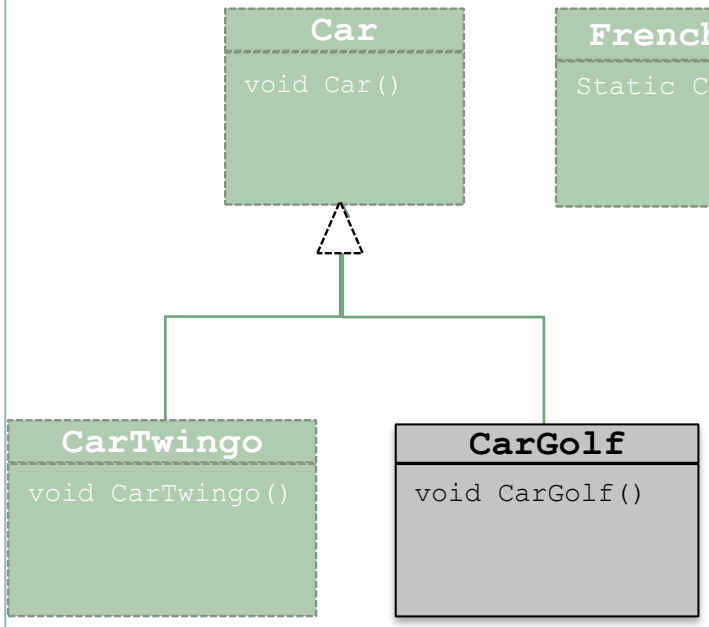
Exemple de Simple Factory



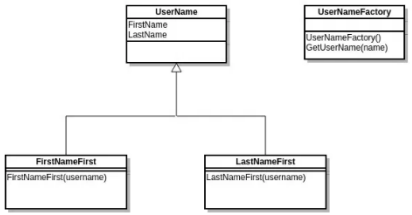

```
public abstract class Car {
   protected int pollutionIndex;
   public Car() {}
}
```



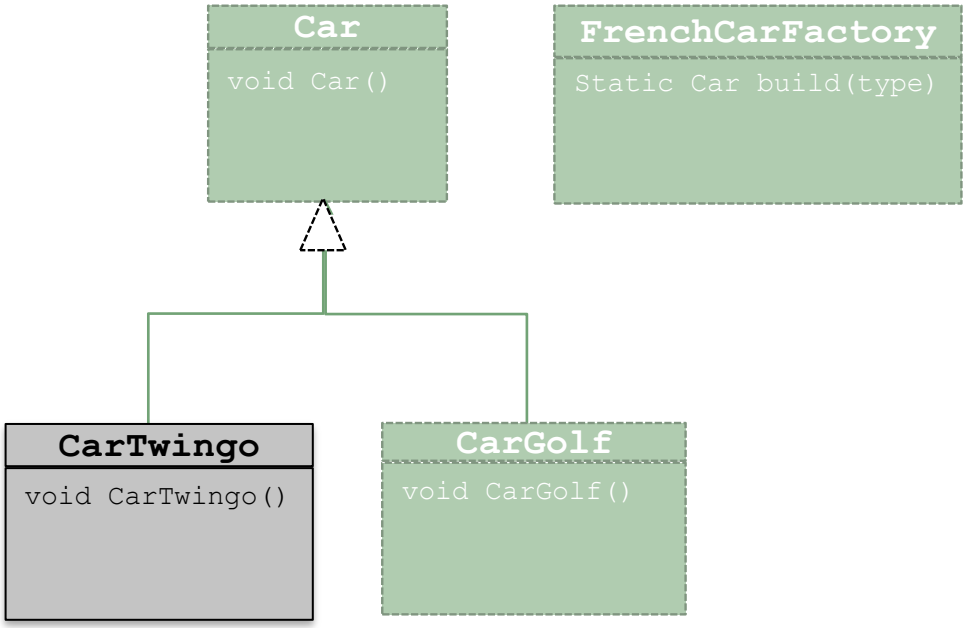
Exemple de Simple Factory



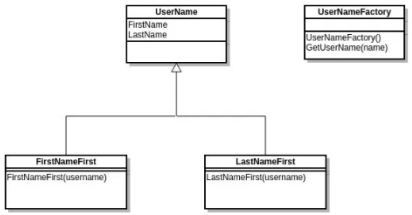

```
public class CarGolf extends Car {
   public CarGolf() {
      super();
      pollutionIndex = 2;
      System.out.println("Golf is created");
   }
}
```



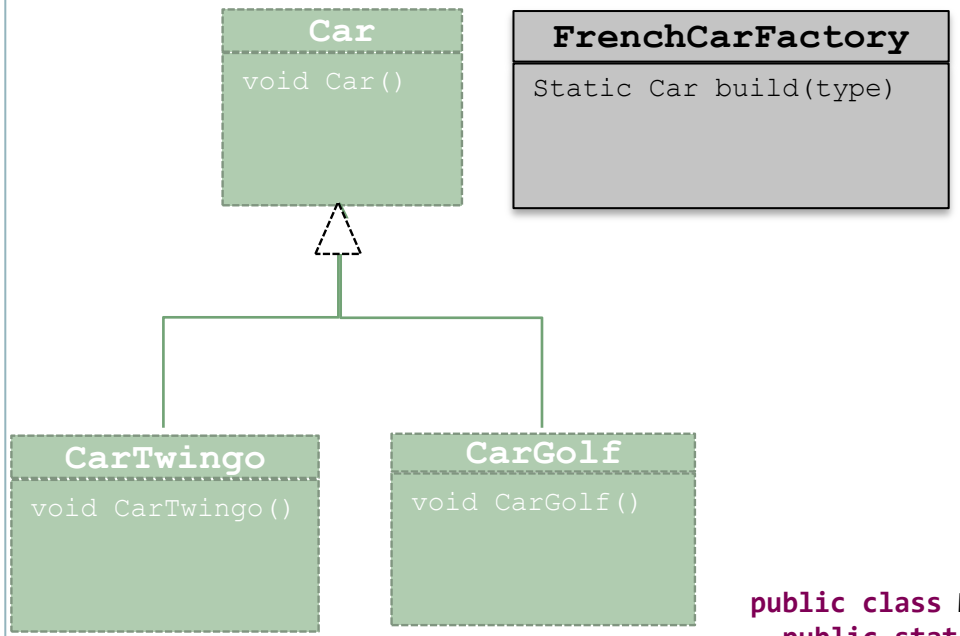
Exemple de Simple Factory




```
public class CarTwingo extends Car {
   public CarTwingo() {
      super();
      pollutionIndex = 4;
      System.out.println("Twingo is created");
   }
}
```

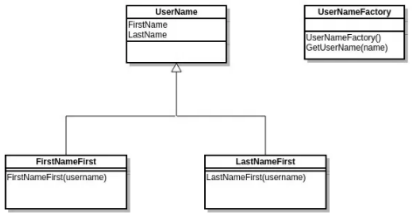


Exemple de Simple Factory

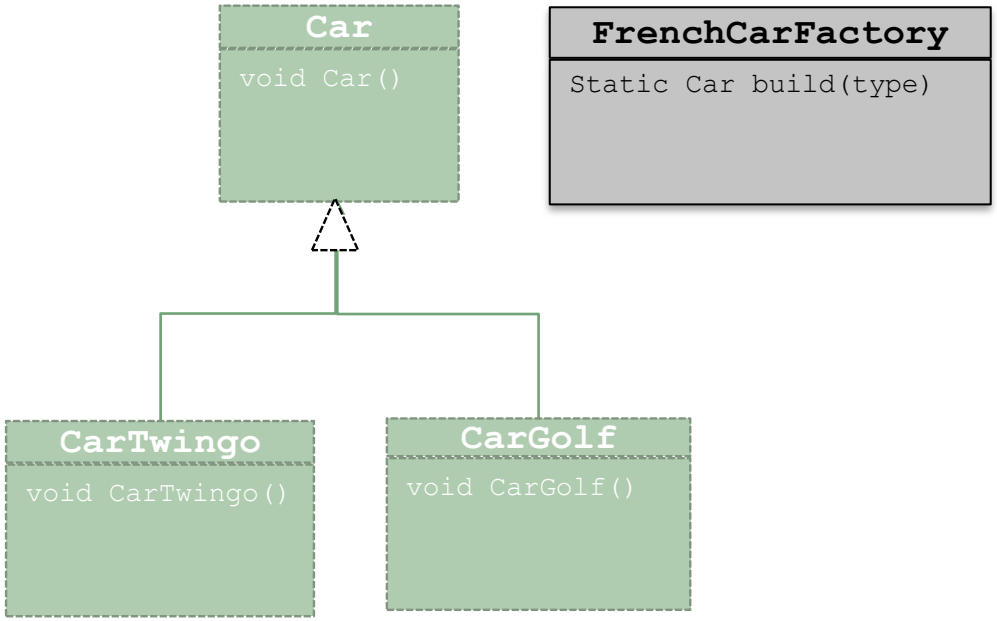



```
public class FrenchCarFactory {
    public static final int TWINGO = 1;
    public static final int GOLF = 2;
    static Car build(int type) throws Throwable {
      switch(type) {
            case TWINGO : return new CarTwingo();
            case GOLF : return new CarGolf();
            default: throw new Throwable("not made");
      }
   }
}
```

```
public class Main {
  public static void main(String[] args) throws Throwable {
    Car fredCar = FrenchCarFactory.build(FrenchCarFactory.TWINGO);
    Car lisaCar = FrenchCarFactory.build(FrenchCarFactory.GOLF);
  }
}
```



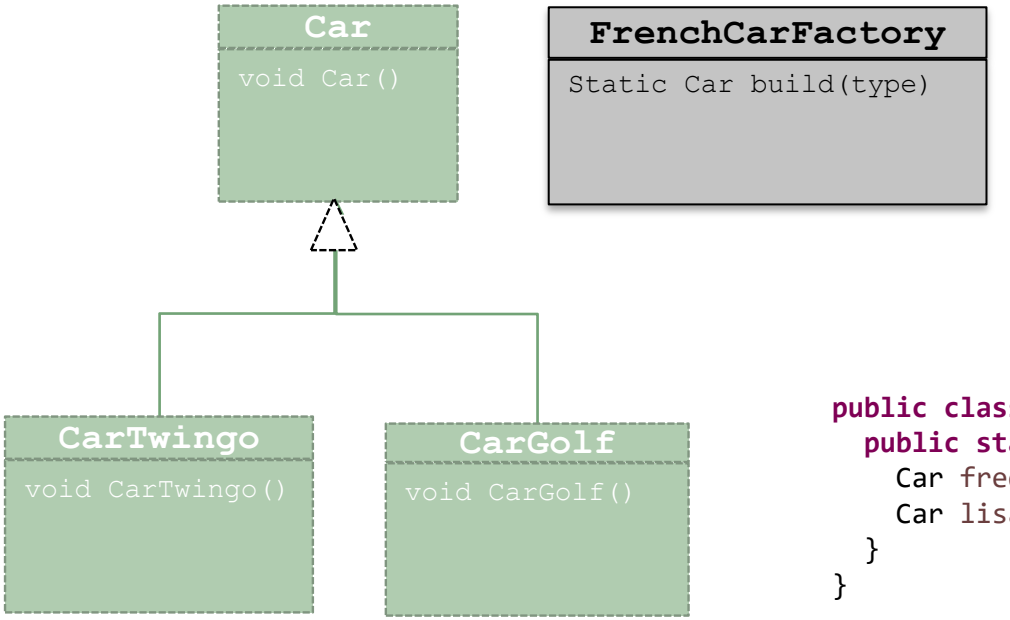
Exemple de Simple Factory




```
public class Main {
  public static void main(String[] args) throws Throwable {
    Car fredCar = FrenchCarFactory.build(FrenchCarFactory.TWINGO);
    Car lisaCar = FrenchCarFactory.build(FrenchCarFactory.GOLF);
  }
}
```

```
Twingo is created
Golf is created
```

Exemple de Simple Factory

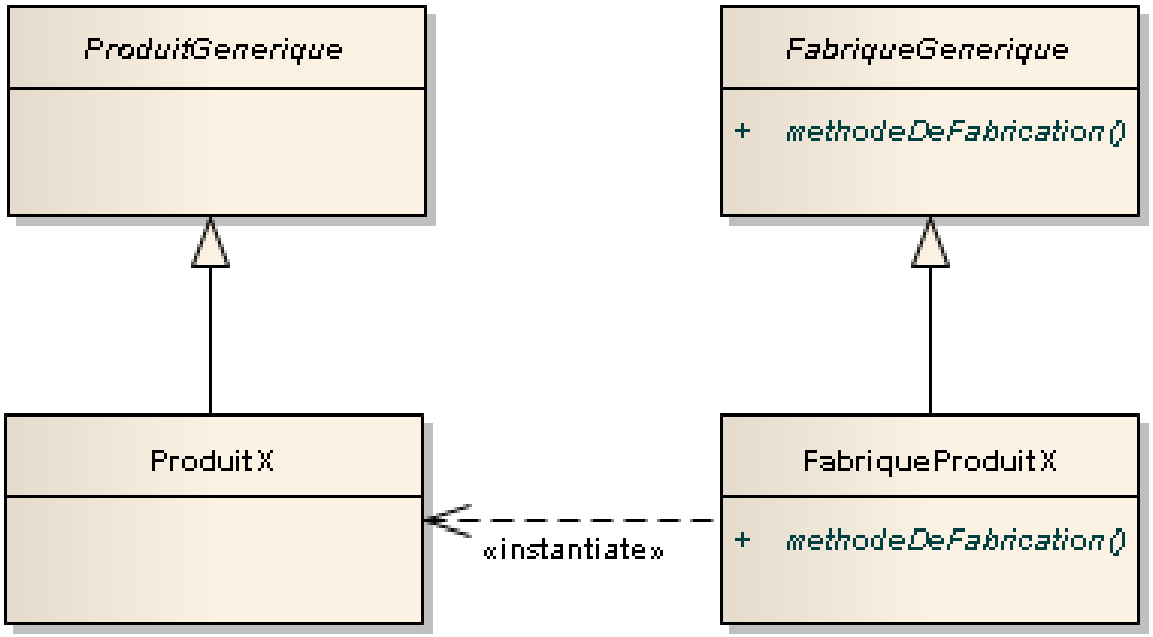


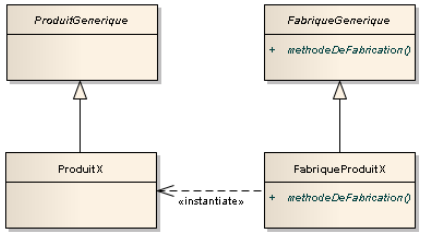

```
public class Main {
  public static void main(String[] args) throws Throwable {
    Car fredCar = CarFactory.build(CarFactory.TWINGO);
    Car lisaCar = CarFactory.build(CarFactory.GOLF);
  }
}
```

- On veut créer un objet Voiture
- Il faut s'assurer que l'on dispose de tous les éléments de la voiture
- Dissocier la logique de création de celle de l'implémentation (réutilisation du code)

- CONCLUSION : mission accomplie ! …mais on peut faire mieux !

FACTORY METHOD





FACTORY METHOD

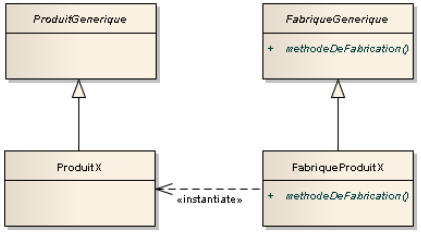
Comment ça fonctionne ?

- L'intentions est de
  - choisir la bonne sous-classe en fonction de certains paramètres (comme Simple Factory)
  - Déléguer l'instanciation (c'est-à-dire cache la classe concrète utilisée) pour permettre l'évolution

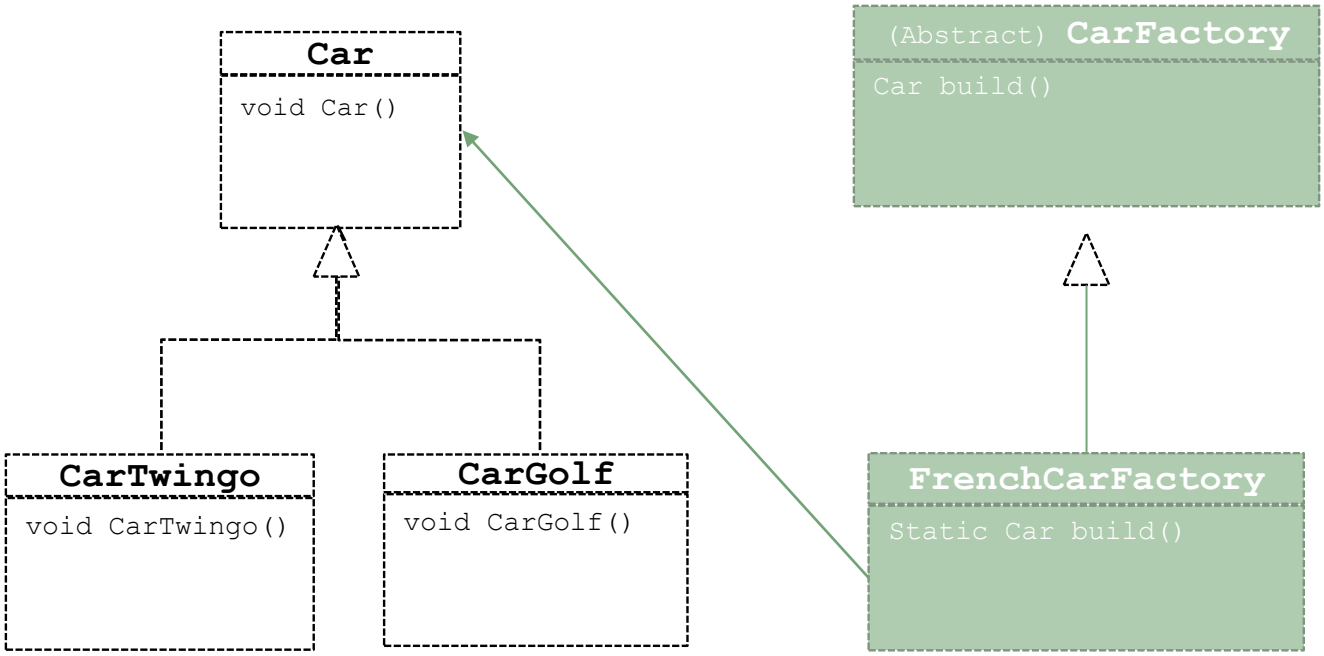
- La principale différence entre Simple Factory et Factory Method est que l'appelant dans la Factory method ne fait pas référence à FactoryMethod elle-même mais à une classe héritée

- La Factory utilisée est alors du type : `FactoryMethod f1 = new ConcreteFactory();`

- La fabrication est ainsi définie dans la classe **ConcreteFactory.** Ainsi ConcreteFactory pourra créer un produit concret et le renvoyer à l'appelant (l'appelant utilise une interface ou une classe parent pour accéder également au produit). Le diagramme suivant montre la relation:



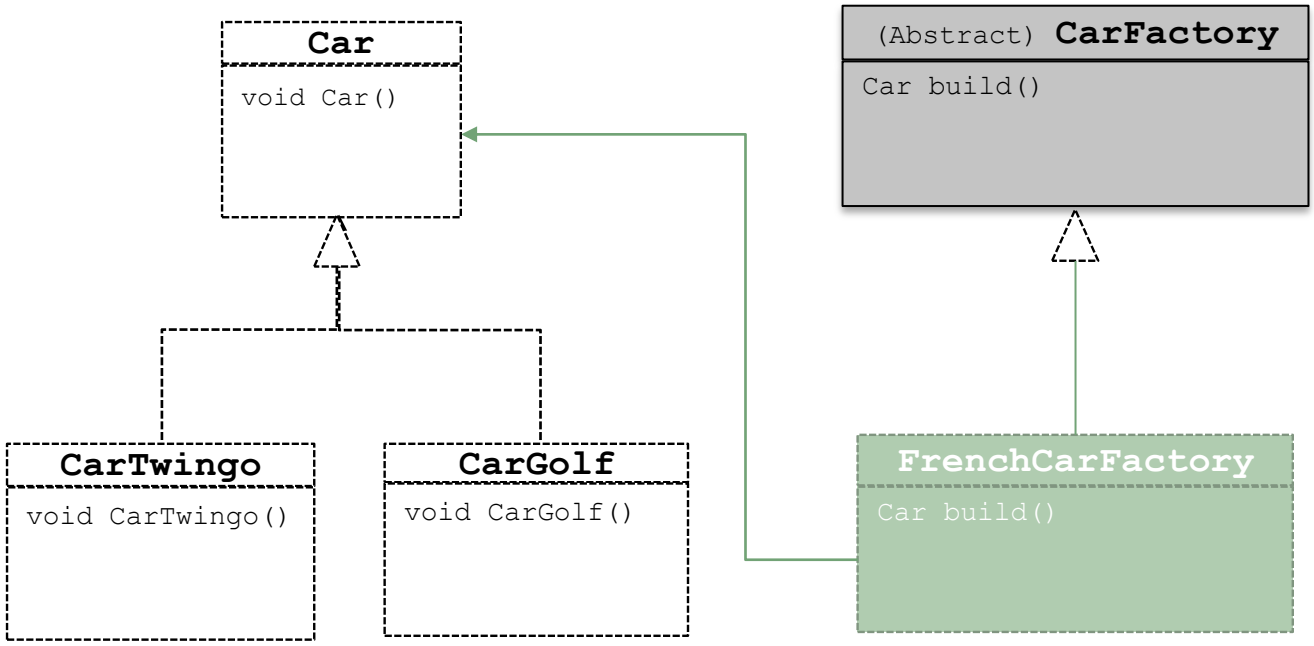
Exemple de Factory Method



Intention :

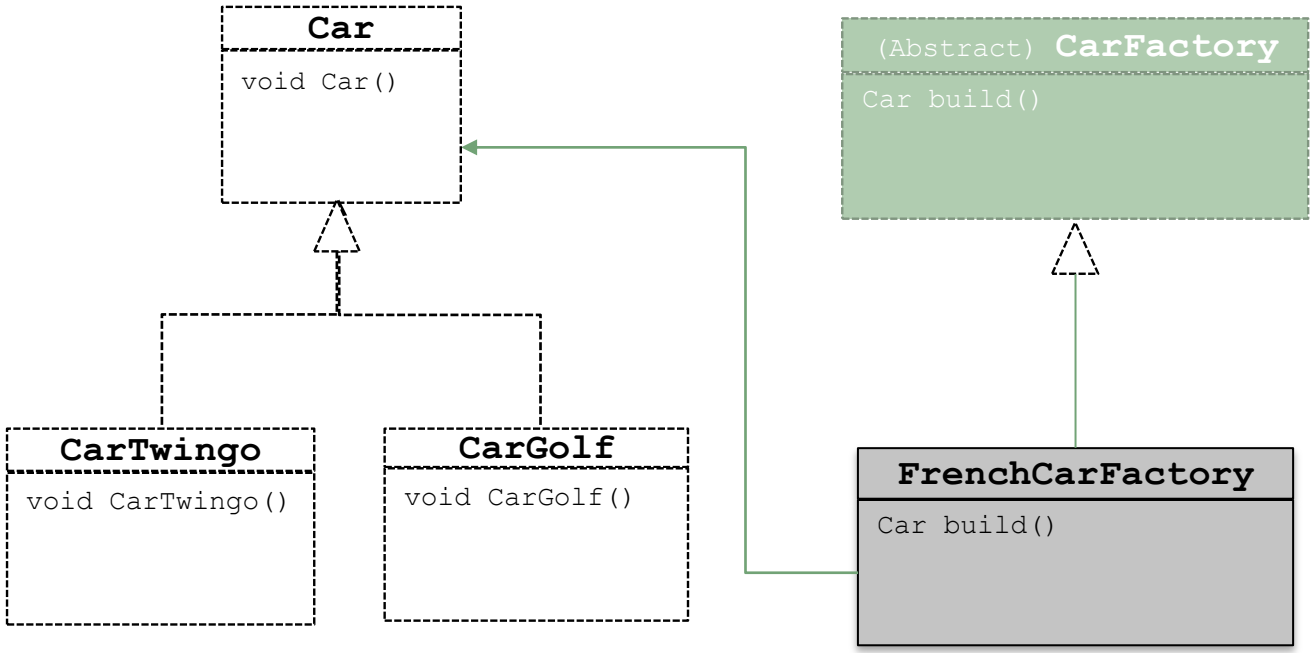
- **La fabrication d'une voiture peut dépendre du pays (attribut pollutionIndex)**
- **On souhaite pouvoir créer autant de CarFactory que de pays**

Exemple de Factory Method



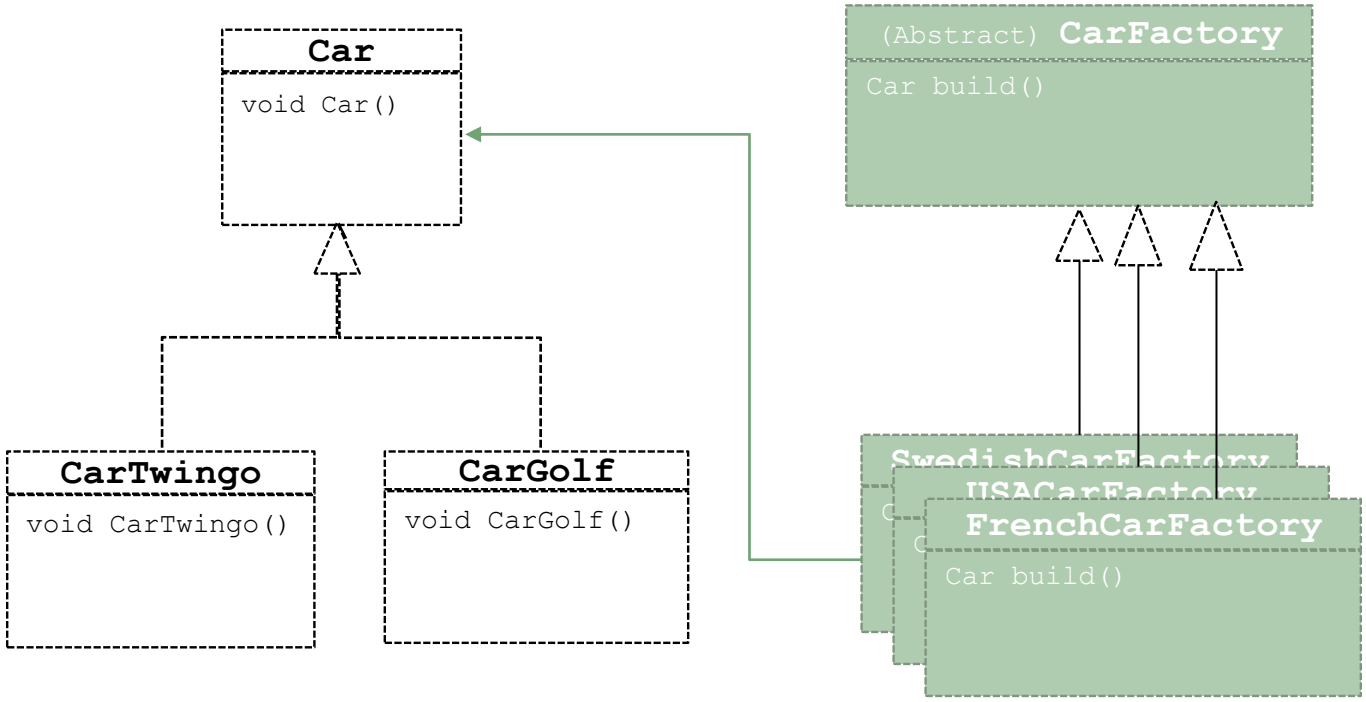

```
public abstract class CarFactory {
    public static final int TWINGO = 1;
    public static final int GOLF = 2;
    abstract protected Car build(int type) throws Throwable ;
}
```

Exemple de Factory Method




```
public class FrenchCarFactory extends CarFactory{
   public Car build(int type) throws Throwable {
      switch(type) {
         case TWINGO: return new CarTwingo();
         case GOLF: return new CarGolf();
         default: throw new Throwable("not made");
      }
   }
}
```

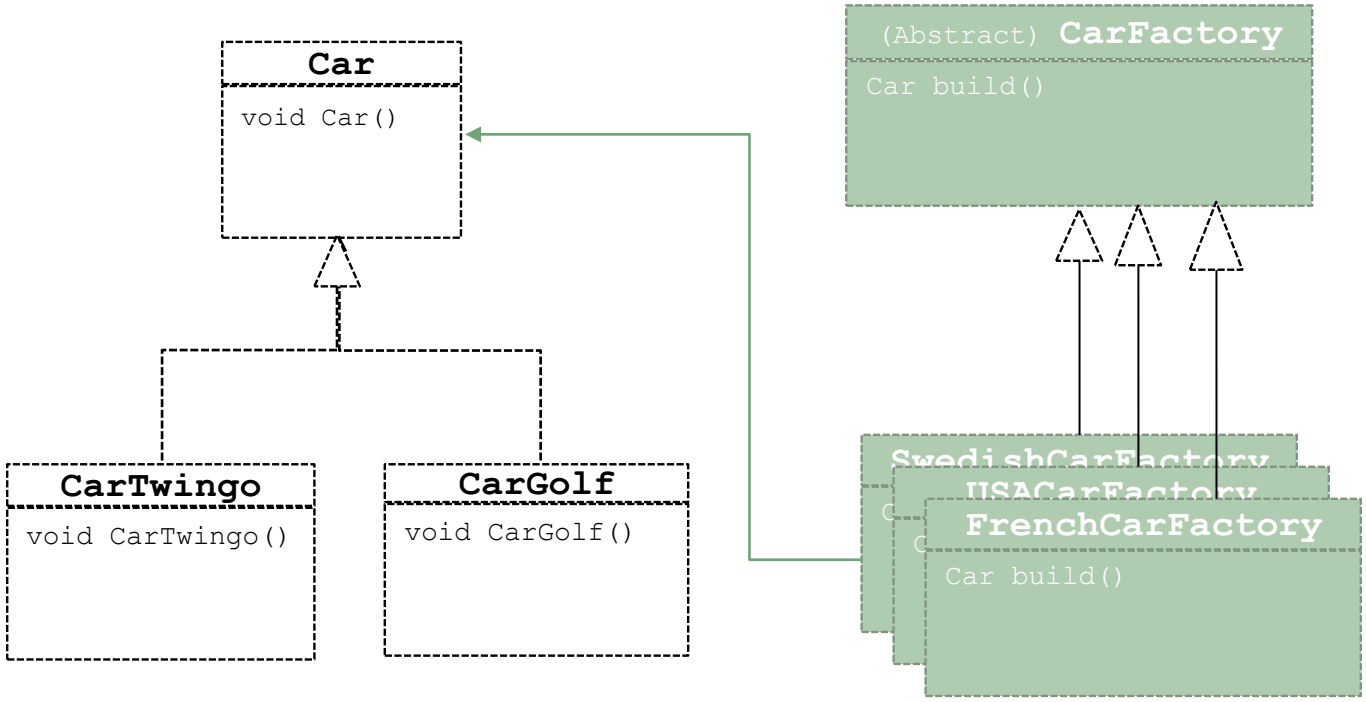
Exemple de Factory Method



?? Autre concrètes CarFactory ??

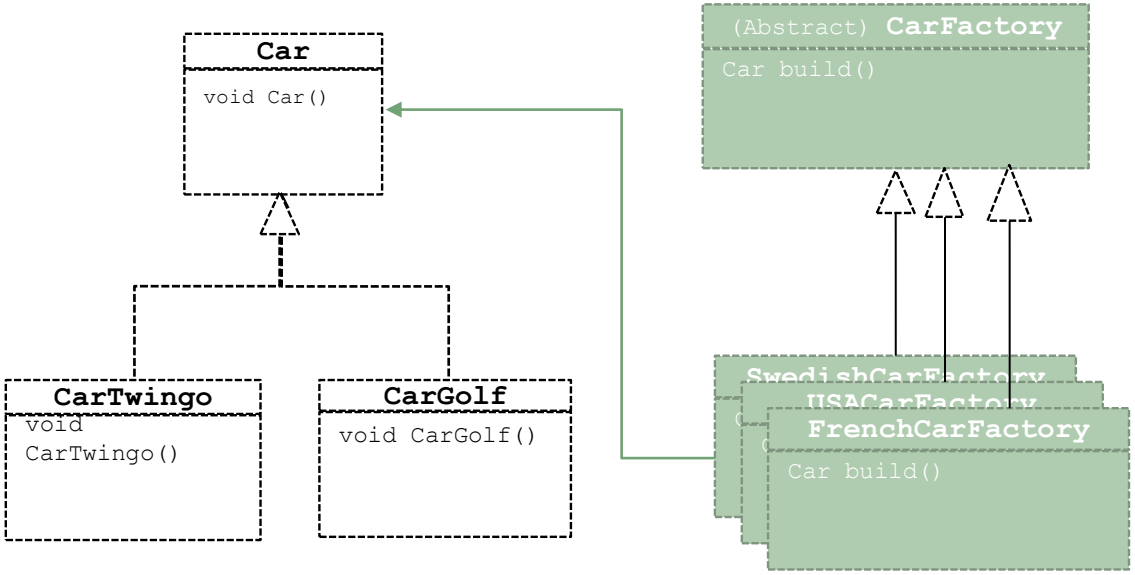
→ Dans le main, on peut utiliser et manipuler un CarFactory , sans se soucier du pays de provenance

Exemple de Factory Method




```
public static void main(String[] args) throws Throwable {
   CarFactory frenchCarFactory = new FrenchCarFactory();
   frenchCarFactory.build(CarFactory.TWINGO);
   frenchCarFactory.build(CarFactory.GOLF);
}
```

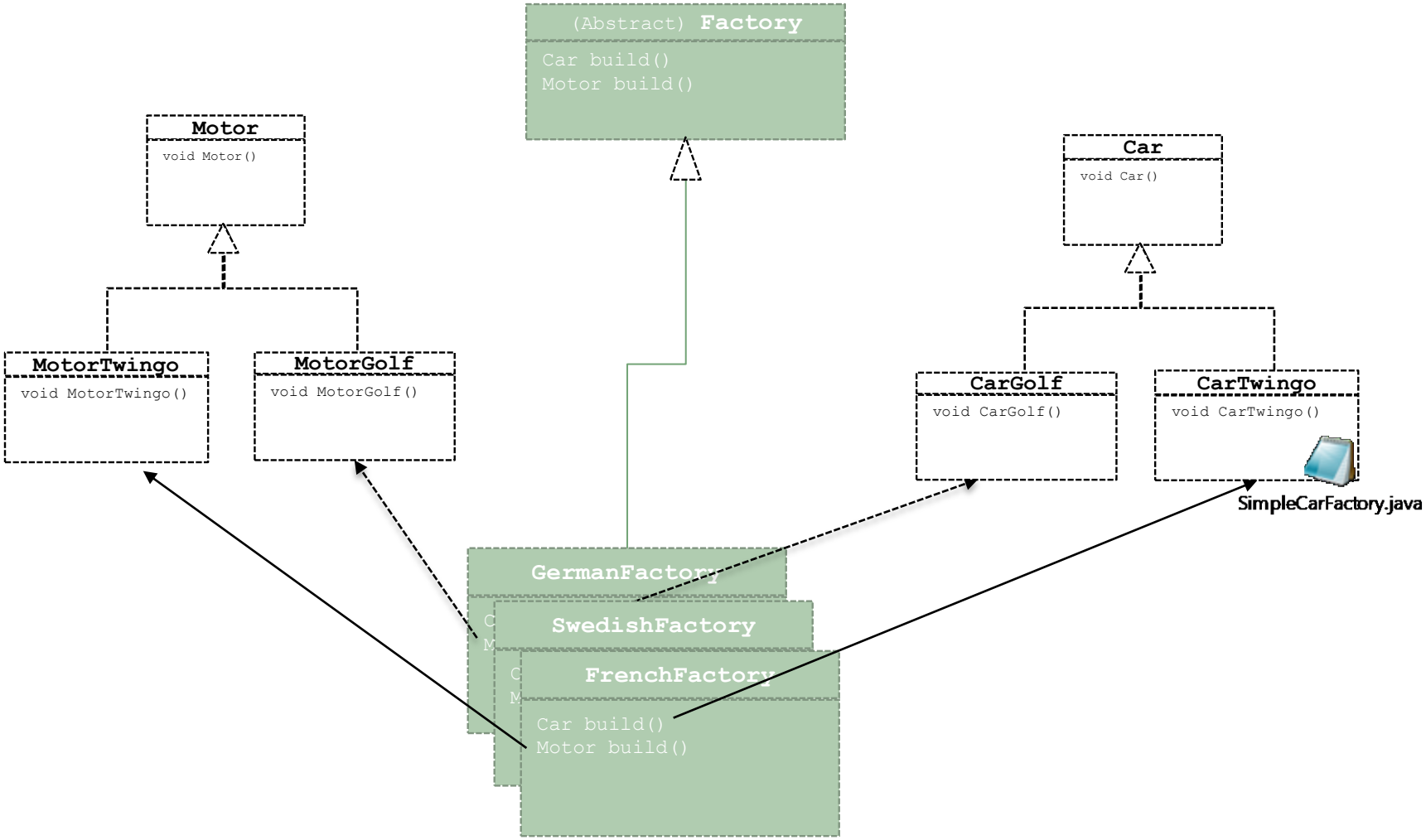
Exemple de Factory Method



- CarFactory est une classe abstraite dont la mission est de créer des objets abstraits Car
- CarFactory permet à ses sous-classes de décider comment implémenter la création des objets abstraits Car
- FrenchCarFactory pourra créer un produit concret CarTwingo par exemple et le renvoyer à l'appelant (qui lui, utilise la classe CarFactory pour accéder à Car).

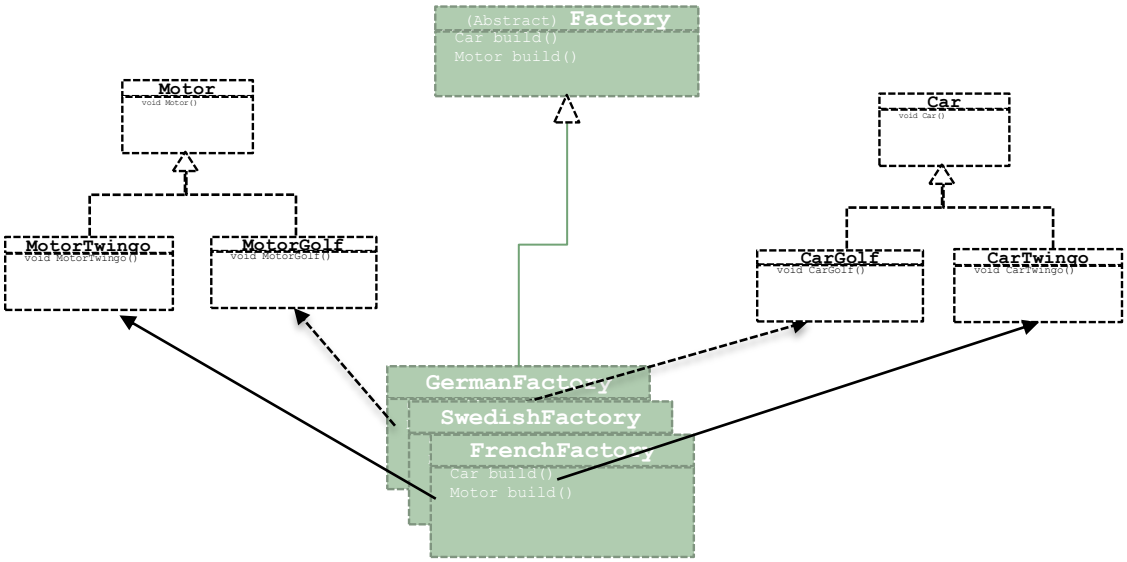
CONCLUSION : mission accomplie ! …mais si CarFactory fabriquait aussi les moteurs ?

ABSTRACT FACTORY



ABSTRACT FACTORY

Comment ça fonctionne ?



- L'intentions est de
  - Fournir une interface pour créer des familles d'objets dépendants ou associés sans connaître leur classe réelle
  - Fabriquer des fabriques.

- La principale différence entre Factory Method et Abstract Factory c'est que l'Abstract Factory a pour but de créer une famille (plusieurs) objets dépendants entre eux

- L'idée ici est de chercher à éviter les incohérences commises par erreur. Par exemple on cherche à créer un couple d'objet comme :
  - Un fragment pour Android ET un spinner
  - Un Layout pour IOS ET un picker
  - → On veut interdire de créer un fragment pour Android et un picker

Exemple de Abstract Factory

```
public abstract class AbstractFactory {
  protected ICar car;
  protected IMotor motor;

  public AbstractFactory() {
   if (car==null) car = buildCar();
   if (motor==null) motor = buildMotor();
  }
  abstract IMotor buildMotor();
  abstract ICar buildCar();
}
```

(abstract) Factory

(abstract) products

```
public interface ICar {
  void process();
}
```

```
interface IMotor {
  void process();
}
```

(concreate) Factory

```
public class FrenchFactory extends AbstractFactory {
  @Override
  IMotor buildMotor() { return new MotorTwingo();}

  @Override
  ICar buildCar() { return new CarTwingo();}
}
```

(concreate) products

```
public class Main {
   public static void main(String[] args) {
      AbstractFactory factory1 = new FrenchFactory();
      AbstractFactory factory2 = new GermanFactory();
   }
}
```

Console

<terminated> Main (5) [Java Application] C:\Program Files\Java\

```
Twingo car is processing...
Twingo motor is processing...
Golf car is processing...
Golf motor is processing...
```

Exemple de Abstract Factory

```
interface IMotor {
   void process();
}
```

(abstract) product

```
public interface ICar {
   void process();
}
```

(abstract) product

```
public class MotorGolf implements IMotor {
   public MotorGolf() { process();}

   @Override
   public void process() {
        System.out.println("Golf motor is processing...");
   }
}
```

(concreate) product

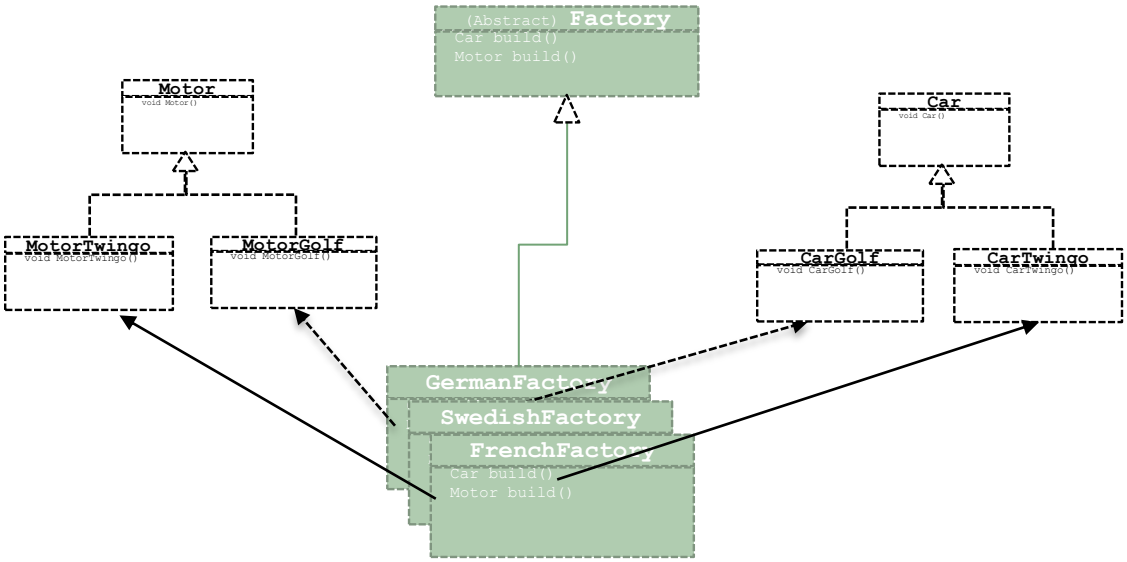
```
public class CarGolf implements ICar {
   public CarGolf() { process();}

   @Override
   public void process() {
        System.out.println("Golf car is processing...");
   }
}
```

(concreate) product

```
public class Main {
   public static void main(String[] args) {
      AbstractFactory factory1 = new FrenchFactory();
      AbstractFactory factory2 = new GermanFactory();
   }
}
```

Exemple de Abstract Factory



- L'intentions est de
  - Fournir une interface pour créer des familles d'objets dépendants ou associés sans connaître leur classe réelle
  - Fabriquer des fabriques.

- La principale différence entre Factory Method et Abstract Factory c'est que l'Abstract Factory a pour but de créer une famille (plusieurs) objets dépendants entre eux

- L'idée ici est de chercher à éviter les incohérences commises par erreur. Par exemple on cherche à créer un couple d'objet comme :
  - Une voiture Twingo ET un moteur pour Twingo
  - Une voiture Golf ET un moteur Golf
  - → CONCLUSION : On peut interdire de créer une voiture Twingo ET un moteur Golf