# Architecture Logicielle

# Les patrons de construction

Florent Nicart

Université de Rouen

*2016–2017*

# Rappels

- Jusqu'ici nous avons intensivement utilisé le *polymorphisme de type* :

```
1  TypeA variable = new TypeB();
```

- Induit par une relation d'héritage ou d'implémentation :

```
1  public class TypeB extends TypeA { ...
2        OU
3  public class TypeB implements TypeA { ...
```

- Le contrôle des types par le compilateur assure que tout type dérivé est conforme au supertype.

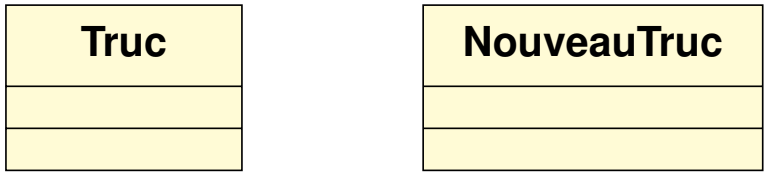
# Rappels

- Le principe de substitution de **Liskov** nous invite à faire en sorte que le comportement d'un type dérivé reste conforme à celui définit par ses supertypes :

```
1  public void methode(TypeA aa) {
2        // aa n'est probablement pas de type 'TypeA' mais
3        // je vais le traiter en tant que tel quand même :
4        aa.unMethodeDefinieDansTypeA();
5  }
6
7  public void test() {
8        // Ah vous voyez !
9        methode(new TypeB());
10 }
```

# Rappels

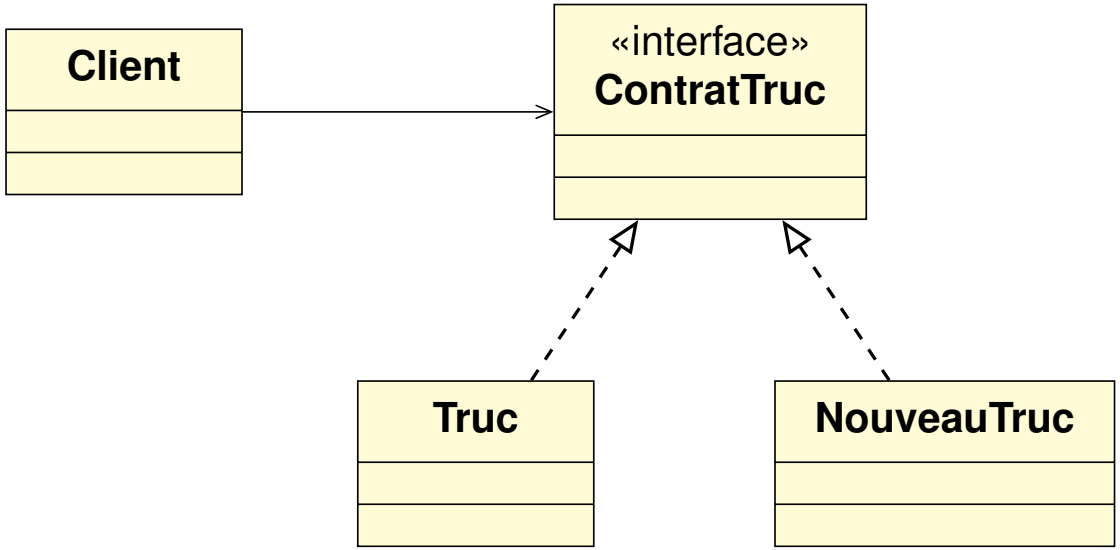
- L'OCP quant à lui nous invite à ajouter sans modifier :




- Le polymorphisme permet de faire fonctionner les nouveaux modules avec le code existant sans le modifier.

# Rappels

- L'OCP quant à lui nous invite à ajouter sans modifier :




- Le polymorphisme permet de faire fonctionner les nouveaux modules avec le code existant sans le modifier.

# Problème

- Si un objet peut être utilisé de manière polymorphe, le mécanisme d'instanciation ne l'est pas :

```
1  public void methode() {
2      ContratTruc polytruc=new Truc();
3      // Code statiquement lié pour fonctionner avec le type Truc !
4  }
```

- Nous avons besoin de pouvoir faire varier le type des objets instanciés :

```
1  public void methode() {
2      ContratTruc polytruc=donneMoiUnTruc(); // S'il te plaît...
3      // Code fonctionnant avec tout sous-type de Truc !
4  }
5
6  public void donneMoiUnTruc() { // Ceci n'est pas une fabrique !
7      if (condition) return new Truc();
8      else return new NouveauTruc();
9  }
```

- Mais ce n'est pas très orienté objet.

# Les patrons constructions

Les patrons constructions ont pour objectif :

- de faire disparaître les instanciations avec `new` qui rigidifie le code,
- faire varier l'instanciation de manière polymorphe plutôt que de manière codée en dure (Fabriques),
- produire un assemblage complexe d'objets (Builder).

# Le patron Factory Method (Fabrique)

Définir une interface pour créer un objet et laisser l'implémenteur choisir le type.

# Le patron **Factory Method**

## Aussi connu comme

Virtual Constructor, Fabrique

## Intention

Définir une interface pour créer un objet et laisse les sous-types décider de la classe à instancier.

## Motivation

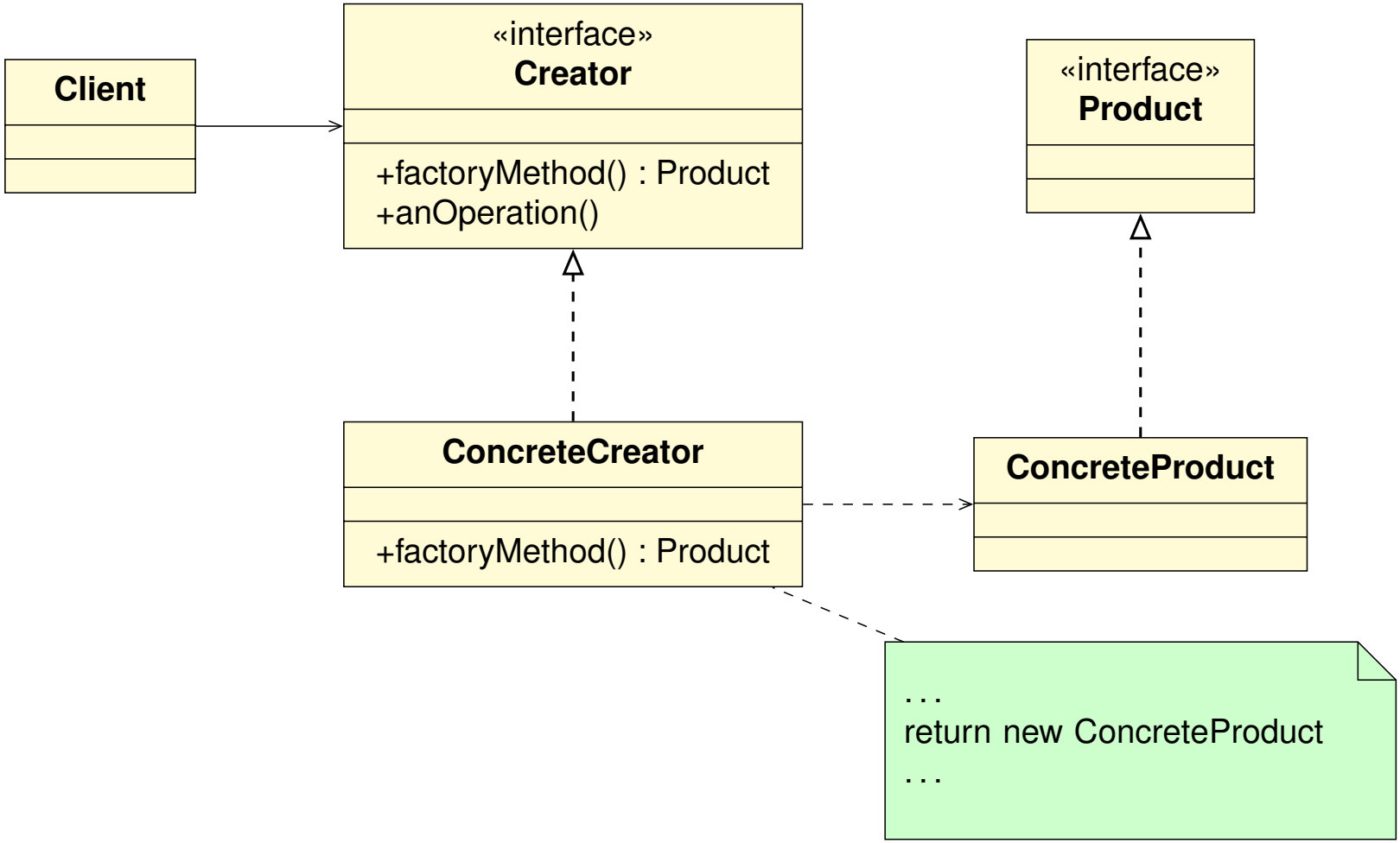
- Les framework définissent des abstractions inter-dépendantes.
- Il est nécessaire de garantir la cohérence entre les types effectifs des implémentations. Ex : pour une base de données, on a le concept de `Connection` et de `ResultSet`, un `MySQLResultSet` doit correspondre à un `MySQLConnection`

# Participants du patron **Factory Method**

- **Product** : L'interface qui spécifie l'objet à produire.
- **ConcreteProduct** : Une classe correspondant à l'objet à produire.
- **Creator** : L'interface spécifiant la `factory method` retournant l'objet.
- **ConcreteCreator** : Un acteur implémentant la `factory method` retournant l'objet.

# Factory Method

## Schéma de principe



# Polymorphism Only

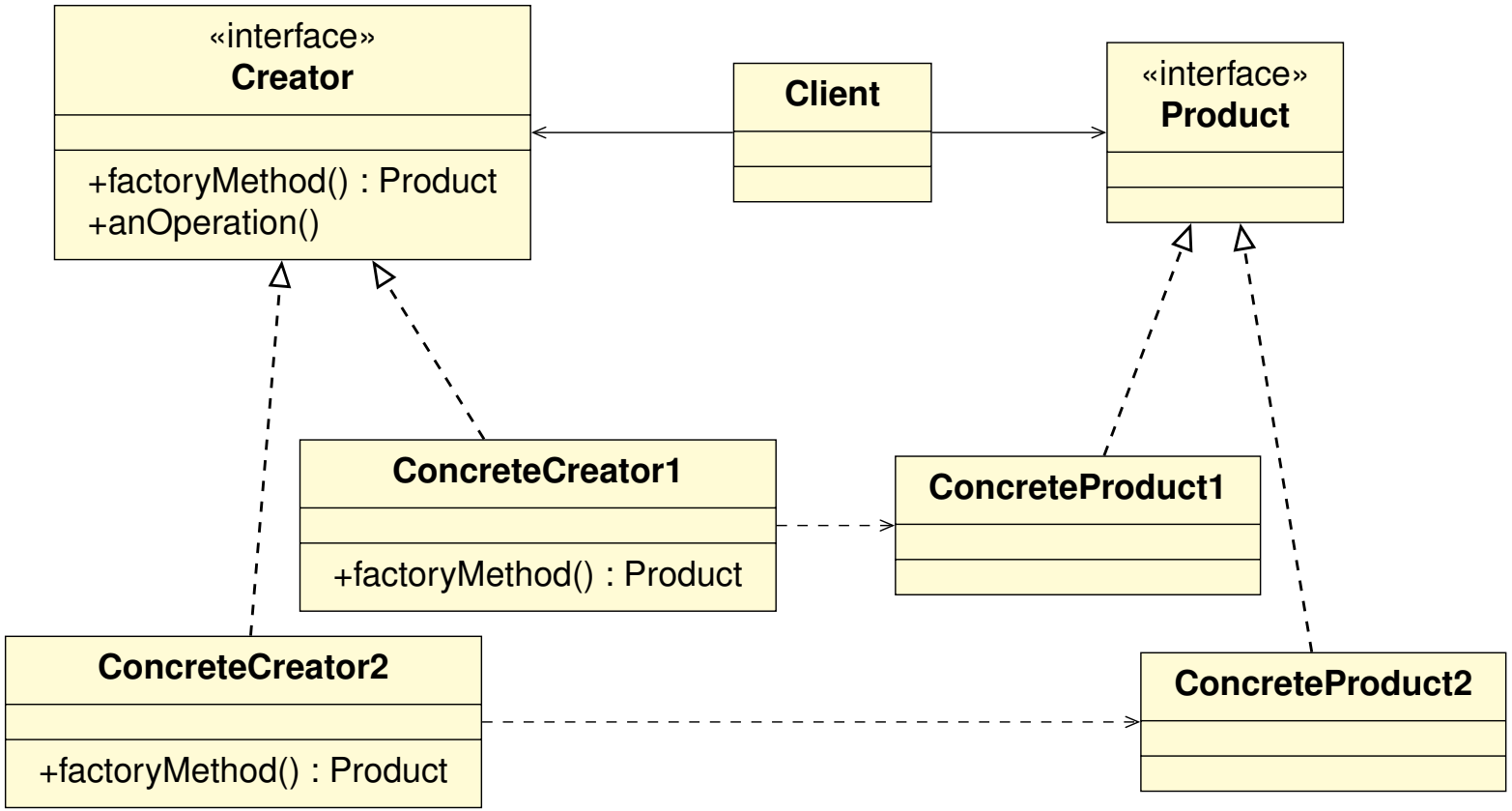
## Fonctionnement

« le patron *Factory Method* permet aux classes dérivés de déterminer la classe à instancier ? »

- Signifie que la classe de l'objet à créer est déterminé (polymorphiquement) en fonction de la classe de son créateur.
- Et non pas par son créateur en fonction d'une autre information.
- Une méthode qui produit des objets sans être polymorphe n'est pas une *factory method*.
- En général, une *Factory Method* contient une seul instanciation statique.
- Un créateur concret est une classe métier ayant déjà une autre fonction.

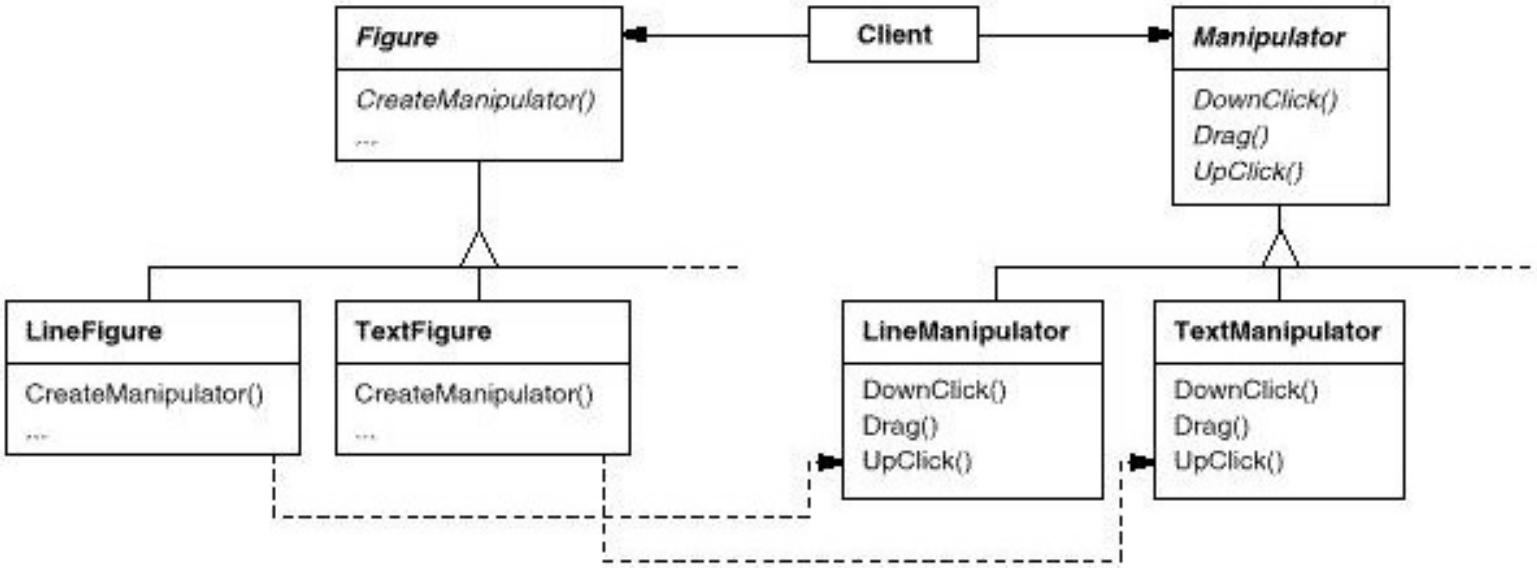
# Schéma de principe

## Plus précisément



# Exemple 1

## Manipulateur graphique (GOF)




- Manipulateur : points de contrôle (clé) sur des éléments de dessin.
- *Factory Method* : `CreateManipulator()`

# Exemple 2

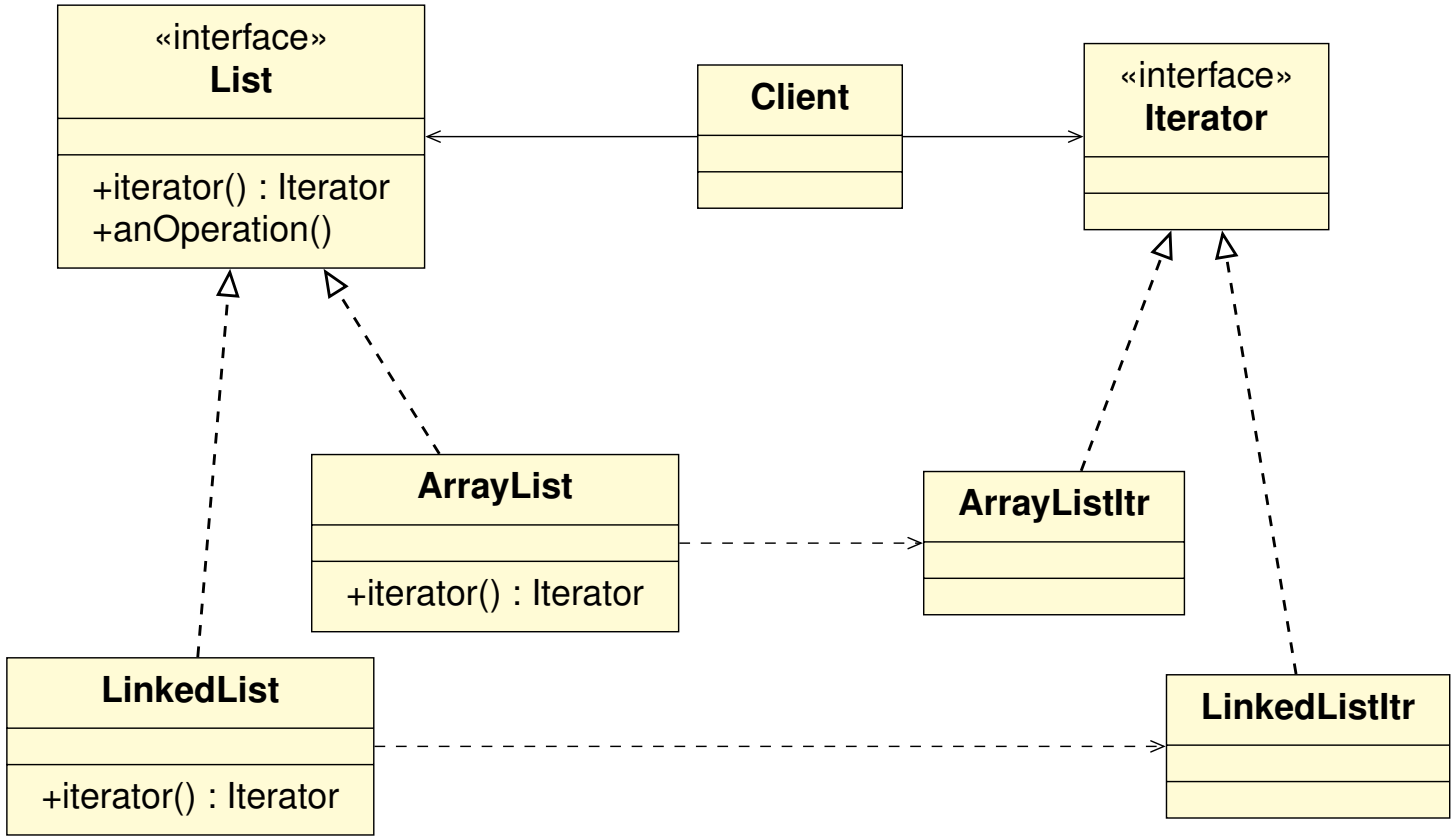
## API Java

- Vous l'utilisez tous les jours ! ... ?

# Exemple 2

## API Java

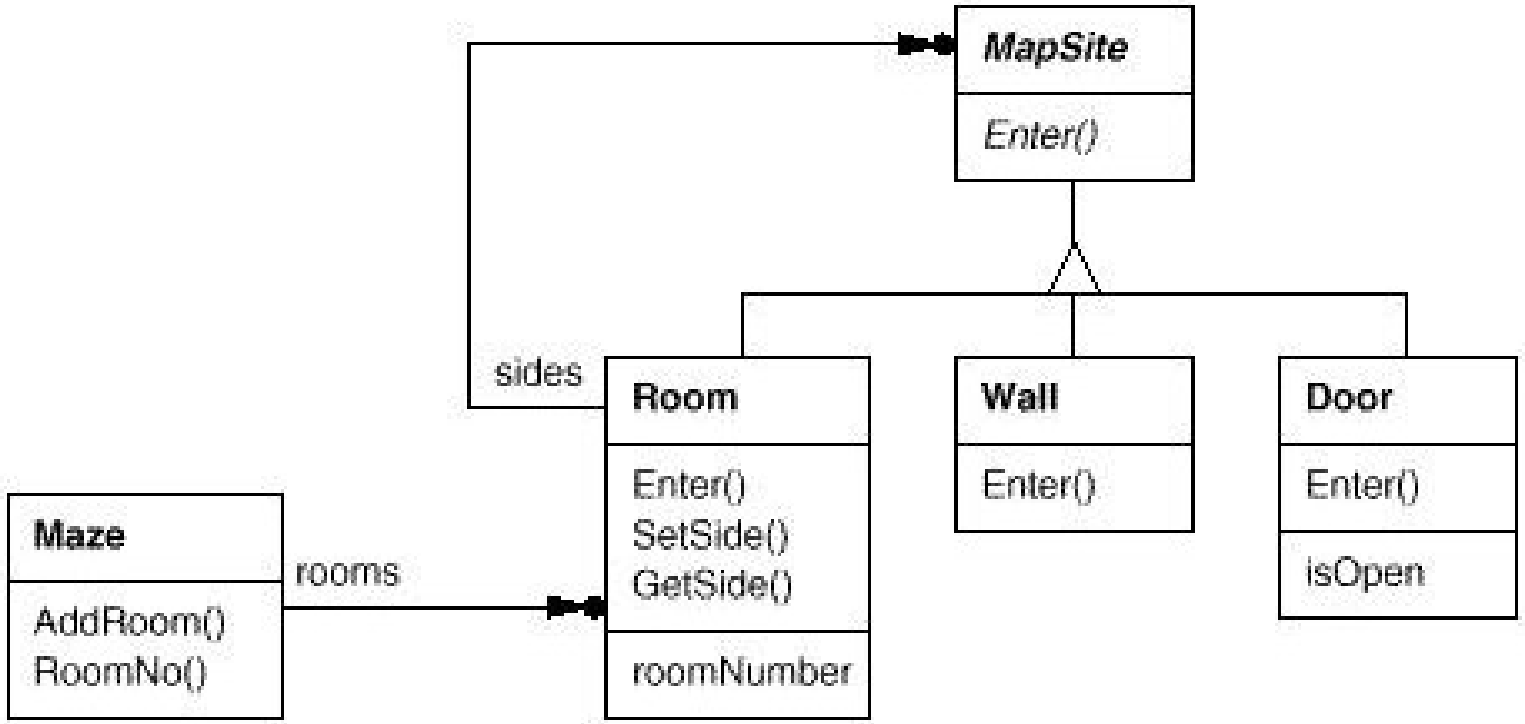
- Vous l'utilisez tous les jours ! ... ?




- *Factory Method* : `iterator()`.

# Exemple 3

## The labyrinth



# Exemple 3

## The labyrinth : creation

```
1  public class MazeGame {
2    // Create the maze.
3    public Maze createMaze() {
4      Maze maze = new Maze();
5      Room r1 = new Room(1);
6      Room r2 = new Room(2);
7      Door door = new Door(r1, r2);
8      maze.addRoom(r1);
9      maze.addRoom(r2);
10     r1.setSide(MazeGame.North, new Wall());
11     r1.setSide(MazeGame.East, door);
12     r1.setSide(MazeGame.South, new Wall());
13     r1.setSide(MazeGame.West, new Wall());
14     r2.setSide(MazeGame.North, new Wall());
15     r2.setSide(MazeGame.East, new Wall());
16     r2.setSide(MazeGame.South, new Wall());
17     r2.setSide(MazeGame.West, door);
18     return maze;
19   }
20 }
```

# Exemple 3

## The labyrinth : creation

- Cette méthode manque de flexibilité car elle est liée statiquement aux classes à instancier.
- On souhaite introduire un nouveau type de labyrinthe, un labyrinthe enchanté avec :
  - des salles enchantées et
  - des portes magiques
- La méthode `createMaze()` doit être réécrite si l'on souhaite créer un labyrinthe avec la même configuration mais basé sur les nouveaux concepts.

# Exemple 3

## The labyrinth : creation

- Ajoutons des *factory method* à la classe `MazeGame` :

```
1  /** MazeGame with a factory methods. */
2  public class MazeGame {
3        public Maze makeMaze() {return new Maze();}
4        public Room makeRoom(int n) {return new Room(n);}
5        public Wall makeWall() {return new Wall();}
6        public Door makeDoor(Room r1, Room r2) {
7        return new Door(r1, r2);
8  }
```

# Exemple 3

## The labyrinth : creation

- Modifions `createMaze()` pour qu'elle les utilise :

```
1  public Maze createMaze() {
2      Maze maze = makeMaze();
3      Room r1 = makeRoom(1);
4      Room r2 = makeRoom(2);
5      Door door = makeDoor(r1, r2);
6      maze.addRoom(r1);
7      maze.addRoom(r2);
8      r1.setSide(MazeGame.North, makeWall());
9      r1.setSide(MazeGame.East, door);
10     r1.setSide(MazeGame.South, makeWall());
11     r1.setSide(MazeGame.West, makeWall());
12     r2.setSide(MazeGame.North, makeWall());
13     r2.setSide(MazeGame.East, makeWall());
14     r2.setSide(MazeGame.South, makeWall());
15     r2.setSide(MazeGame.West, door);
16     return maze;
17 }
```

# Exemple 3

## The labyrinth : creation

- La classe `EnchantedMaze()` n'a plus qu'à définir ses propres *Factory Methods* :

```
1  public class EnchantedMazeGame extends MazeGame {
2      public Room makeRoom(int n) {
3          return new EnchantedRoom(n);
4      }
5      public Wall makeWall() {
6          return new EnchantedWall();
7      }
8      public Door makeDoor(Room r1, Room r2){
9          return new EnchantedDoor(r1, r2);
10     }
11 }
```

- La même méthode `createMaze()` produira des labyrinthes des deux types :

```
1  public class Test  {
2       public void test() {
3            MazeGame mg=new MazeGame();
4            Maze mz=mg.createMaze();      // Labyrinthe normal.
5
6            MazeGame mg2=new EnchantedMazeGame();
7            Maze mz2=mg2.createMaze();    // Labyrinthe enchanté.
8       }
9  }
```

# Principes respectés

- **S.R.P** : ?
- **O.C.P** : ?
- **L.S.P** : ?
- **I.S.P.** : ?
- **D.I.P.** : ?

# Le patron Abstract Factory (Fabrique Abstraite)

Fournir une interface permettant de créer des objets ayant un lien ou interdépendants sans avoir à spécifier leur classe.

# Le patron **Abstract Factory**

## Aussi connu comme

Kit, Fabrique Abstraite

## Intention

Définir une interface pour créer un objet et laisse les sous-types décider de la classe à instancier.

## Motivation

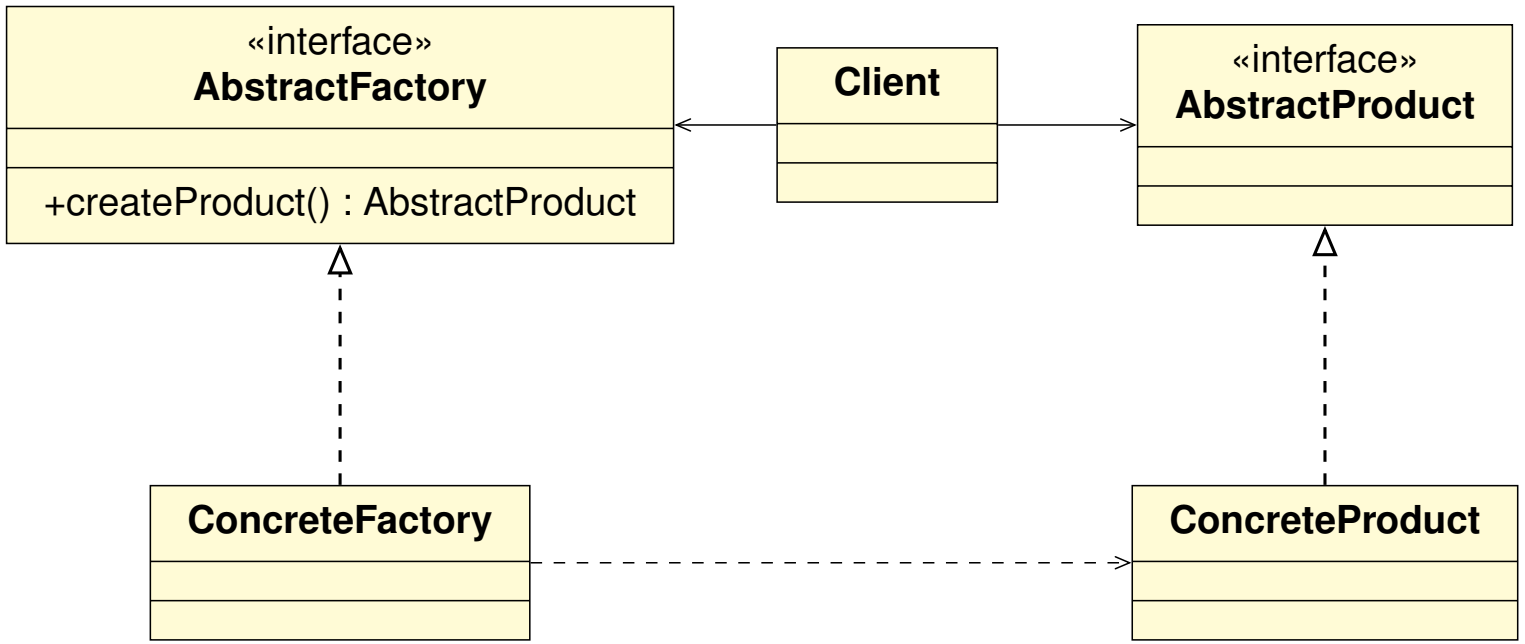
- Une bibliothèque permet à ses clients de produire différents types d'objets (ex : widgets).
- Il n'est pas souhaitable que les clients instancient eux-même les concepts de la bibliothèques car ceux-ci peuvent varier (ex : look-and-feel).
- *Abstract Factory* centralise la production d'une famille de produits.

# Participants du patron **Abstract Factory**

- **AbstractProduct** : Interface qui spécifie les produits.
- **ConcreteProduct** : Une classe correspondant à l'implémentation d'un produit.
- **AbstractFactory** : L'interface spécifiant les opérations de création.
- **ConcreteFactory** : Implémenation d'une `AbstractFactory`.

# Abstract Factory

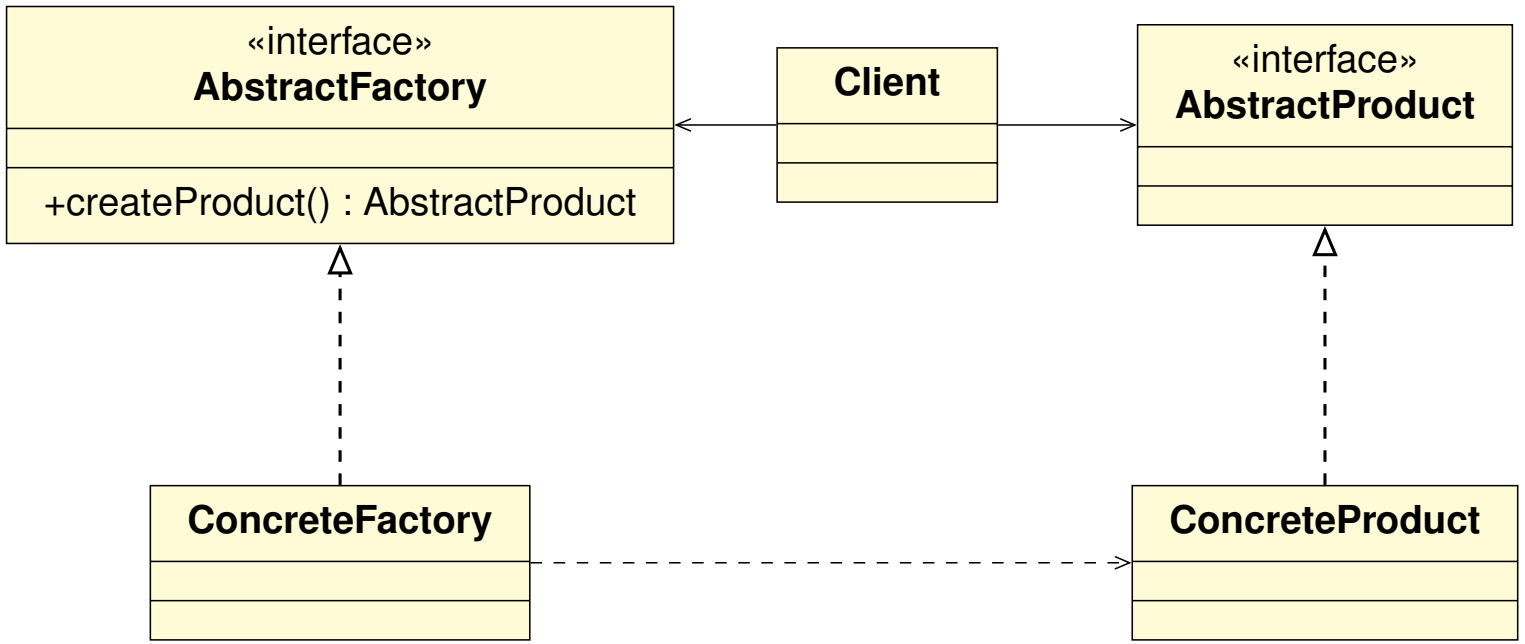
## Schéma de principe




- Euh ... c'est la même chose que *Factory Method* ! non ???
- Non, pas du tout...

# Abstract Factory

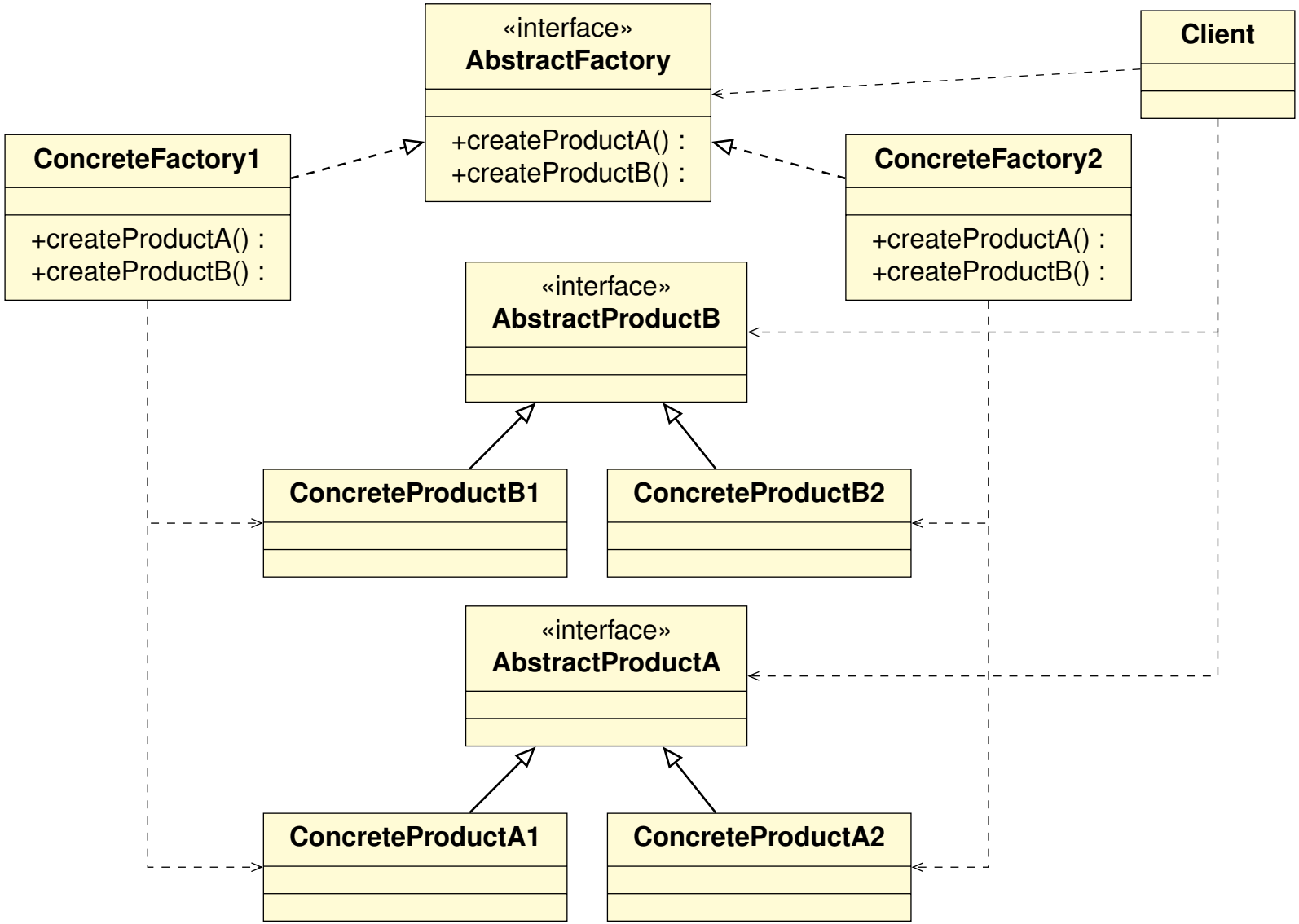
## Schéma de principe




- Euh ... c'est la même chose que *Factory Method* ! non ???
- Non, pas du tout...

# Abstract Factory

## Schéma de principe : et là ?



# Principe

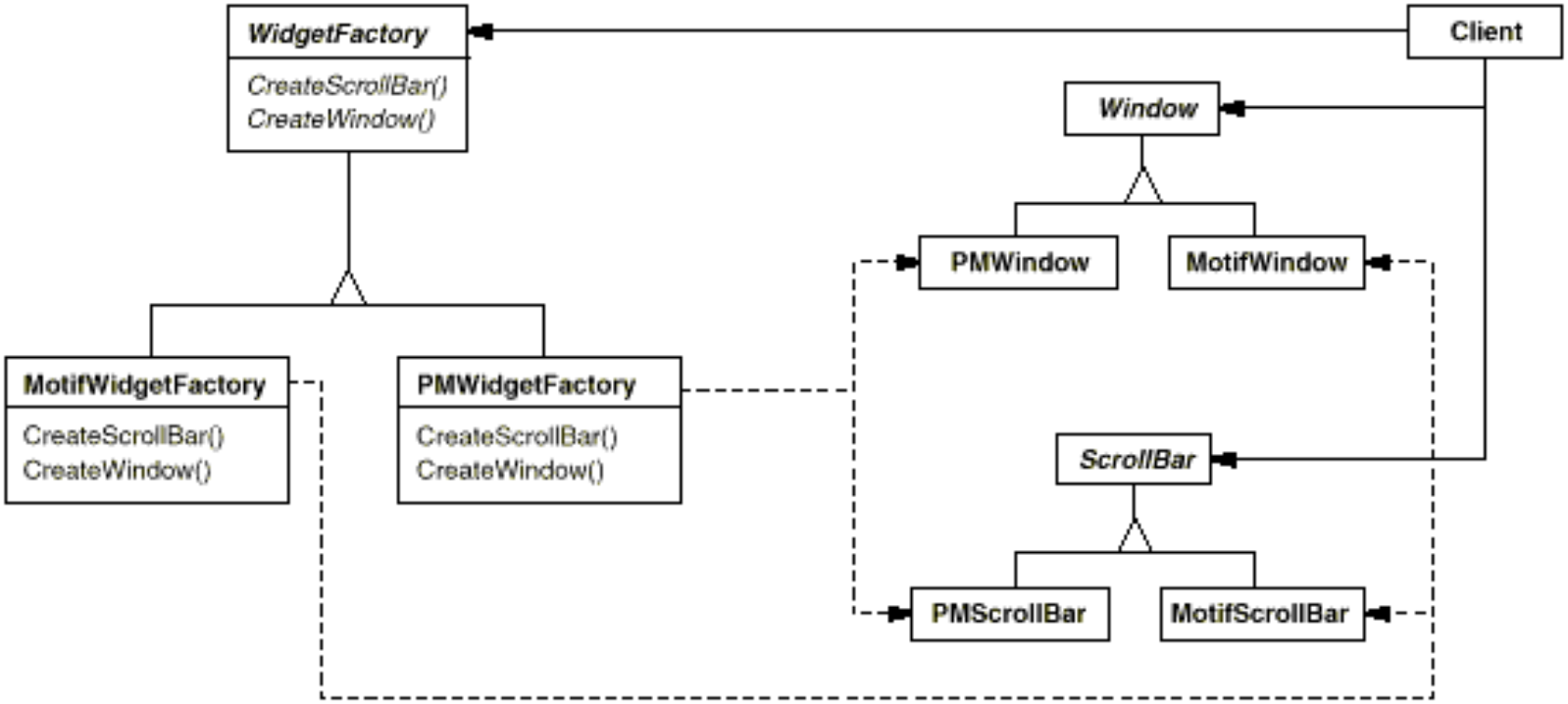
## Fonctionnement

le patron *Abstract Factory* peut être vu comme l'association de plusieurs *Factory Method*, mais :

- La *Abstract Factory* n'a pas de rôle métier. Elle ne sert qu'à produire des objets,
- en général, les classes portent le nom du patron (`somethingFactory`).
- Les *ContreteFactory* peuvent être vides,
- Elles sont nécessaires pour avoir le polymorphisme (il faut un objet).
- Une seule instance est en général nécessaire pour une `ConcreteFactory` donnée (*Singleton*).
- Pour changer de famille de produit, le client change de `ConcreteFactory`.

# Exemple 1

## Widget Factory (GOF)



# Exemple 2

## Retour au labyrinthe

- Mettons en place une fabrique abstraite :

```
1  public interface MazeFactory {
2      public Maze makeMaze();
3      public Room makeRoom(int n);
4      public Wall makeWall();
5      public Door makeDoor(Room r1, Room r2);
6  }
7
8  public class StandardMazeFactory implements MazeFactory {
9      public Maze makeMaze() { return new Maze(); }
10     public Room makeRoom(int n) { return new Room(n); }
11     public Wall makeWall() { return new Wall(); }
12     public Door makeDoor(Room r1, Room r2) {
13         return new Door(r1, r2);
14     }
15 }
16
17 public class EnchantedMazeFactory implements MazeFactory {
18     public Maze makeMaze() { return new Maze(); }
19     public Room makeRoom(int n) { return new EnchantedRoom(n); }
20     public Wall makeWall() { return new EnchantedWall(); }
21     public Door makeDoor(Room r1, Room r2) {
22         return new EnchantedDoor(r1, r2);
23     }
24 }
```

# Exemple 2

## The labyrinth : creation

- Cette fois `createMaze()` reçoit une factory :

```
1  public class MazeGame {
2      public Maze createMaze(MazeFactory factory) {
3          Maze maze = factory.makeMaze();
4          Room r1 = factory.makeRoom(1);
5          Room r2 = factory.makeRoom(2);
6          Door door = factory.makeDoor(r1, r2);
7          maze.addRoom(r1);
8          maze.addRoom(r2);
9          r1.setSide(MazeGame.North, factory.makeWall());
10         r1.setSide(MazeGame.East, door);
11         r1.setSide(MazeGame.South, factory.makeWall());
12         r1.setSide(MazeGame.West, factory.makeWall());
13         r2.setSide(MazeGame.North, factory.makeWall());
14         r2.setSide(MazeGame.East, factory.makeWall());
15         r2.setSide(MazeGame.South, factory.makeWall());
16         r2.setSide(MazeGame.West, door);
17         return maze;
18     }
19 }
```

- `createMaze()` produit des labyrinthes des tout type :

```
1  public class Test  {
2      public void test(MazeGame mg) {
3          MazeFactory smf=new StandardMazeFactory();
4          Maze mz=createMaze(smf);        // Labyrinthe normal.
5
6          MazeFactory emf=new EnchantedMazeFactory();
7          Maze mz2=createMaze(emf);       // Labyrinthe enchanté
```

# Exemple 3

## The labyrinth : creation

- La classe `EnchantedMaze()` n'a plus qu'à définir ses propres *Factory Methods* :

```
1  public class EnchantedMazeGame extends MazeGame {
2      public Room makeRoom(int n) {
3          return new EnchantedRoom(n);
4      }
5      public Wall makeWall() {
6          return new EnchantedWall();
7      }
8      public Door makeDoor(Room r1, Room r2){
9          return new EnchantedDoor(r1, r2);
10     }
11 }
```

- La même méthode `createMaze()` produira des labyrinthes des deux types :

```
1  public class Test  {
2       public void test() {
3            MazeGame mg=new MazeGame();
4            Maze mz=mg.createMaze();^^I// Labyrinthe normale.
5
6            MazeGame mg=new EnchantedMazeGame();
7            Maze mz=mg.createMaze();^^I// Labyrinthe enchanté.
8       }
9  }
```

# Principes respectés

- **S.R.P** : ?
- **O.C.P** : ?
- **L.S.P** : ?
- **I.S.P.** : ?
- **D.I.P.** : ?

# Le patron Builder

Déplacer la logique de construction d'un objet en dehors de la classe à instancier.

# Le patron **Builder**

## Aussi connu comme

/

## Intention

Déplacer la logique de construction d'un objet en dehors de la classe à instancier.

## Motivation

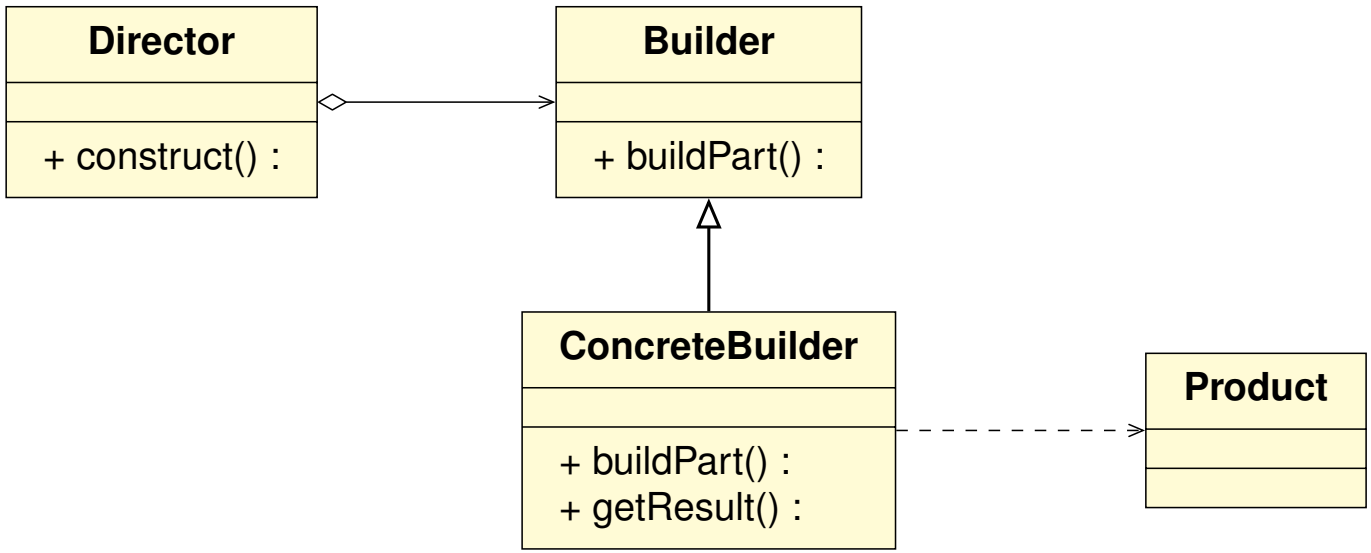
- Il n'est pas toujours souhaitable que la logique de construction d'un objet soit placée dans son constructeur :
- celle-ci peut être complexe,
- être l'objet d'un cas d'utilisation particulier de la bibliothèque,
- peut coupler la classe à des bibliothèques externes spécifiques (ex : parseur).

# Participants du patron **Builder**

- **Builder** : Interface spécifiant les méthodes de création des parties.
- **ConcreteBuilder** : Implémente les méthodes de constructions, mémorise éventuellement l'état de la cosntruction
- **Director** : Construit le produit à l'aide de l'interface `Builder`
- **Product** : Représente la structure complexe à produire

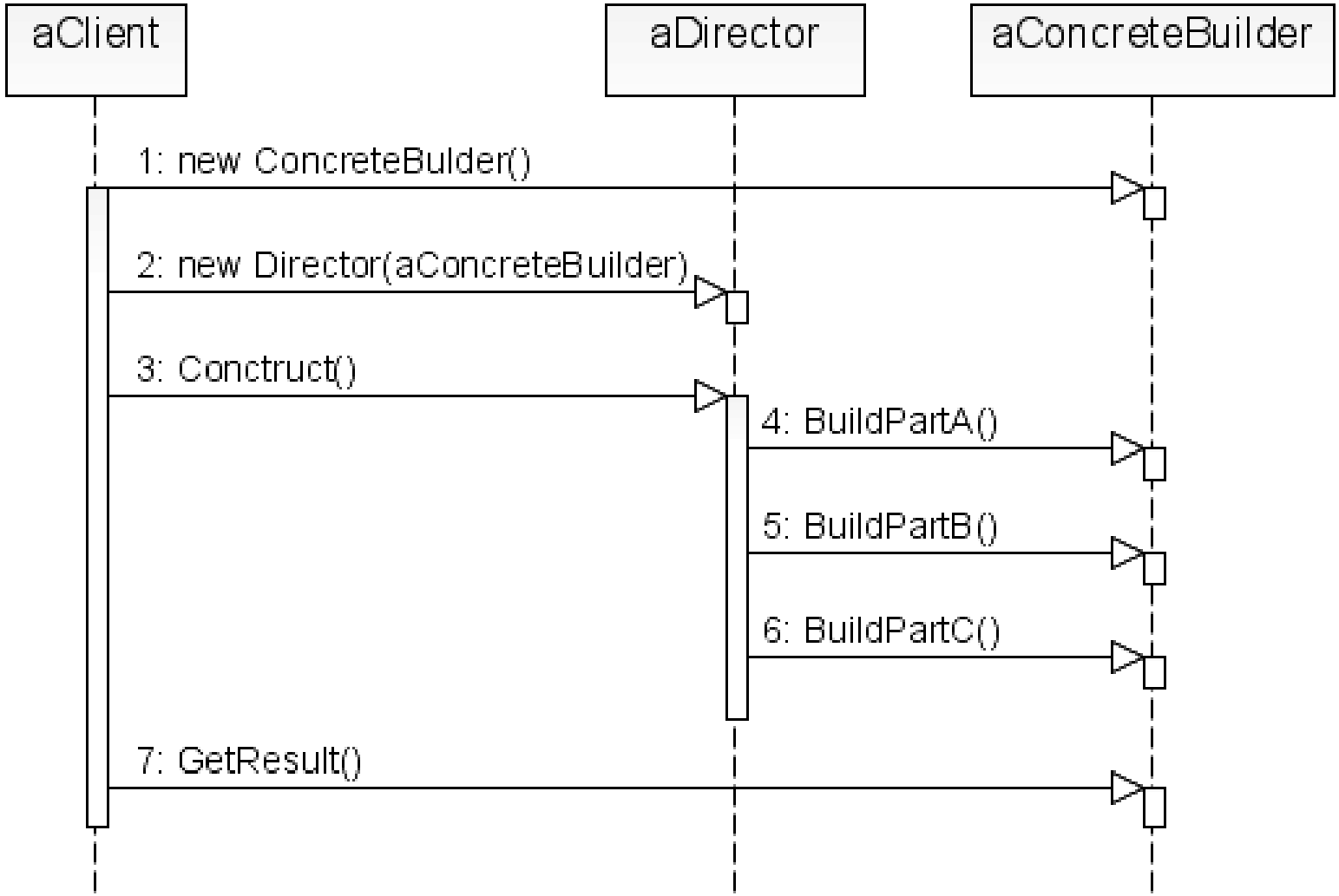
# Builder

## Schéma de principe



# Builder

## Coopérations



# Exemple 1

## Chargement d'un flux XML

- On souhaite créer un graphe d'objets en mémoire correspondant au schéma XML suivant :

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <!DOCTYPE mémo [
3  <!ELEMENT mémo (de,à,contenu)>
4  <!ATTLIST mémo
5  xmlns CDATA #FIXED "http://apiacoa.org/ns">
6  <!ELEMENT de (#PCDATA)>
7  <!ELEMENT à (#PCDATA)>
8  <!ELEMENT contenu (titre,texte)>
9  <!ELEMENT titre (#PCDATA)>
10 <!ELEMENT texte (#PCDATA)>
11 ]>
```

- Par exemple :

```
1  <mémo xmlns="http://apiacoa.org/ns">
2  <de>John</de>
3  <à>Bob</à>
4  <contenu>
5  <titre>Bonjour</titre>
6  <texte>Ca va ?</texte>
7  </contenu>
8  </mémo>
```

# Exemple 1

## Chargement d'un flux XML

- On dispose de la classe `Memo` :

```
1  public final class Memo {
2      final String nl = System.getProperty("line.separator");
3      private class Content {
4          String title;
5          String text;
6          Content(String title, String text) {
7              this.title = title;
8              this.text = text;
9          }
10         public String toString() {
11             StringBuffer buf = new StringBuffer();
12             buf.append("titre : " + title + nl).append("texte : " +
                   text + nl);
13             return buf.toString();
14         }
15     }
16     String from;
17     String to;
18     Content content;
19     ...
20 }
```

# Exemple 1

## Chargement d'un flux XML

- Classe `Memo` (suite) :

```
1       ...
2       Memo(String from, String to, String title, String text) {
3           this.from=from;
4           this.to=to;
5           this.content = new Content(title,text);
6       }
7       public String toString() {
8           StringBuffer buf = new StringBuffer();
9           buf.append("message" + nl).append("de " + from + nl)
10              .append("à " + to + nl).append(content.toString());
11          return buf.toString();
12      }
13  }
```

# Exemple 1

## Chargement d'un flux XML

```
1  public interface MemoBuilder {
2      public Memo build() throws Exception;
3  }
4
5  public class MemoBuilderImpl implements MemoBuilder {
6      private InputSource is;
7      private String title;
8      private String text;
9      private String from;
10     private String to;
11     public MemoBuilderImpl(InputSource is) {
12         this.is = is;
13     }
14     public Memo build() throws ParserConfigurationException, SAXException {
15         SAXParserFactory spf = SAXParserFactory.newInstance();
16         SAXParser sp = spf.newSAXParser();
17         XMLReader xr = sp.getXMLReader();
18         System.out.println("Parser : " + xr.getClass().getName());
19         HandlerImpl handler = new HandlerImpl();
20         xr.setContentHandler(handler);
21         xr.setErrorHandler(handler);
22         try {
23             xr.parse(is);
24         } catch (Exception e) {
25             System.out.println("" + e);
26         }
27         return new Memo(from,to,title,text);
28     }
29     ...
```

# Exemple 1

## Chargement d'un flux XML

```
1       ...
2       private class HandlerImpl extends DefaultHandler implements
            ContentHandler, ErrorHandler {
3       private List<String> stack;
4       public void characters(char[] ch, int start, int length) throws
            SAXException {
5           String buf = new String(ch, start, length);
6           String top = stack.get(stack.size() - 1).trim();
7           if ("titre".equals(top)) title = buf;
8           else if ("texte".equals(top)) text = buf;
9           else if ("de".equals(top)) from = buf;
10          else if ("à".equals(top)) to = buf;
11      }
12      public void endDocument() throws SAXException {
13          stack.clear();
14      }
15      public void endElement(String uri, String localName, String qName) throws
            SAXException {
16          stack.remove(stack.size() - 1);
17      }
18      public void startDocument() {
19          stack = new ArrayList();
20      }
21      public void startElement(String uri, String localName, String qName,
            Attributes atts) throws SAXException {
22          stack.add(qName);
23      }
24  }
```

# Exemple 1

## Chargement d'un flux XML

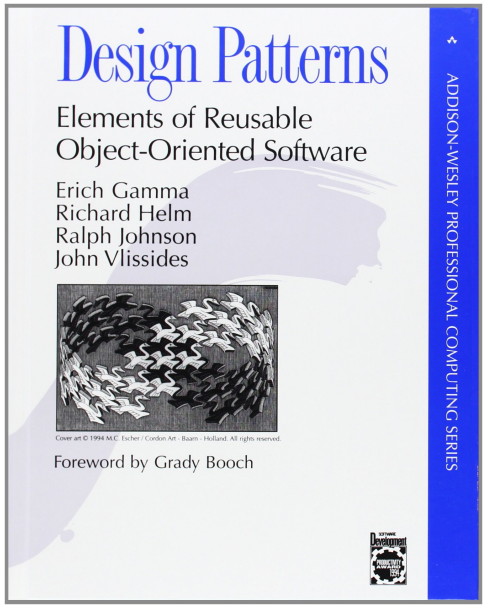
- Programme de test :

```
1  import org.xml.sax.InputSource;
2
3  public class Main {
4      public static void main(String[] args) {
5          MemoBuilder memoBuilder =
6              new MemoBuilderImpl(new InputSource("memo.xml"));
7          try {
8              Memo memo = memoBuilder.build();
9              System.out.println(memo);
10         } catch (Exception e) {
11             e.printStackTrace();
12         }
13     }
14 }
```

# Principes respectés

- **S.R.P** : ?
- **O.C.P** : ?
- **L.S.P** : ?
- **I.S.P.** : ?
- **D.I.P.** : ?

# Quelques références




**Design Patterns : Elements of Reusable Object-Oriented Software.**, *Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides*, Addison Wesley (1994).
ISBN-13 : 978-0201633610.


**Les Design patterns en Java : Les 23 modèles de conception fondamentaux**, *Steven John Metsker, William C.Wake* , Pearson (2009).
ISBN-13 : 978-2744023965.