

Architecture Logicielle



Les patrons de construction

Florent Nicart

Université de Rouen

2017–2018

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

- 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.

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

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

Introduction

Factory Method

Structure
Exemples
Conclusion

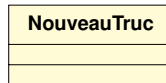
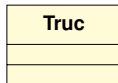
Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

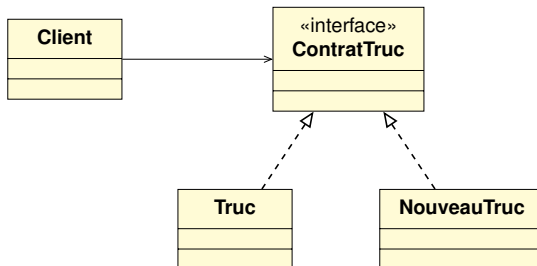
- L'OCP quant à lui nous invite à ajouter sans modifier **et** à concevoir pour ne pas avoir besoin de modifier dans le futur :



- 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 **et** à concevoir pour ne pas avoir besoin de modifier dans le futur :



- 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

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

Les patrons constructions ont pour objectif :

- de faire disparaître les instanciations avec `new` qui rigidifie le code ¹,
- et découpler le code client du type dynamique.
- Faire varier l'instanciation de manière polymorphe plutôt que de manière codée en dure (Fabriques).
- Produire un assemblage complexe d'objets (Builder).

1. Ce n'est pas un objectif systématique !

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**

Introduction

Factory Method

Structure

Exemples

Conclusion

Abstract Factory

Structure

Exemples

Conclusion

Builder

Introduction

Structure

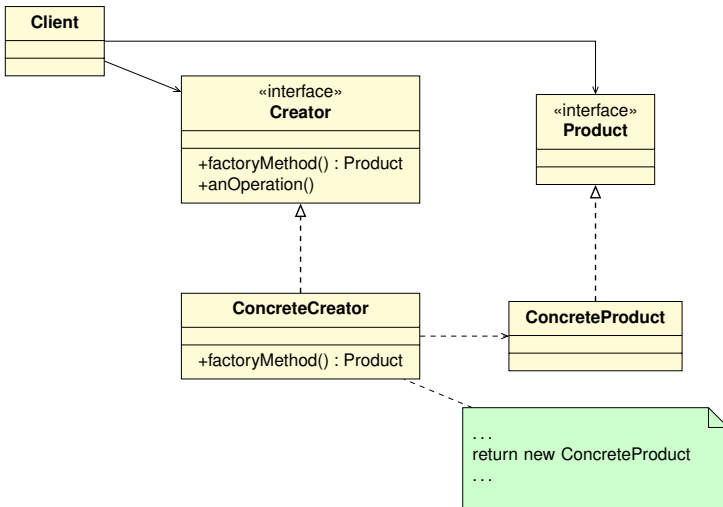
Exemples

Conclusion

- **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

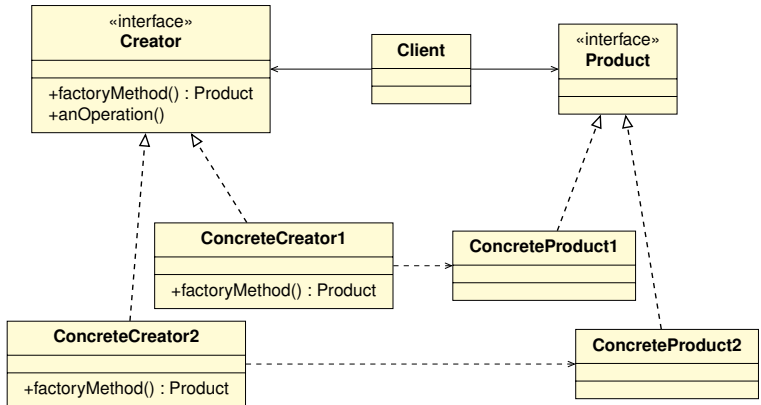
Fonctionnement

« 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 instantiation statique.
- Un créateur concret est une classe métier ayant déjà une autre fonction.

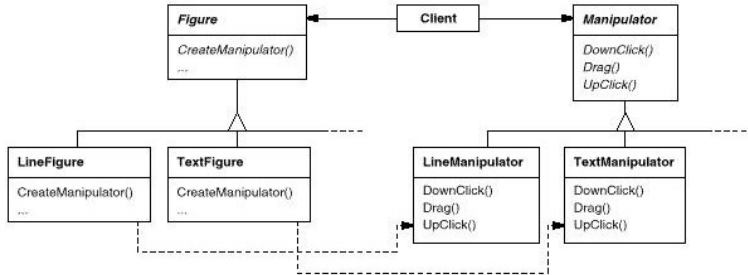
Schéma de principe

Plus précisément



Exemple 1

Manipulateur graphique (GOF)



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

Exemple 2

API Java

Introduction

Factory Method

Structure

Exemples

Conclusion

Abstract Factory

Structure

Exemples

Conclusion

Builder

Introduction

Structure

Exemples

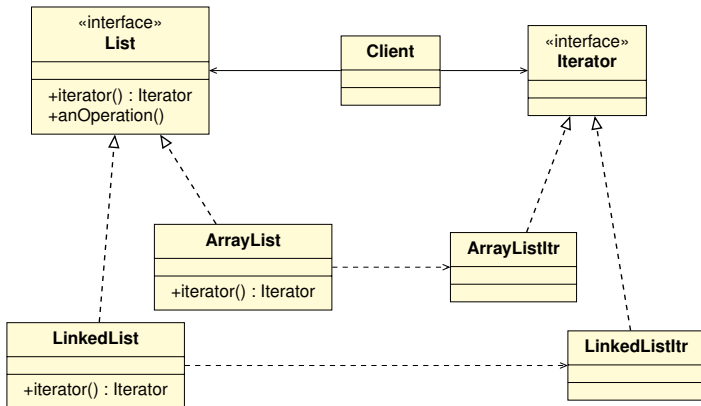
Conclusion

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

Exemple 2

API Java

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



- Factory Method* : `iterator()`.

Exemple 3

The labyrinth

Introduction

Factory Method

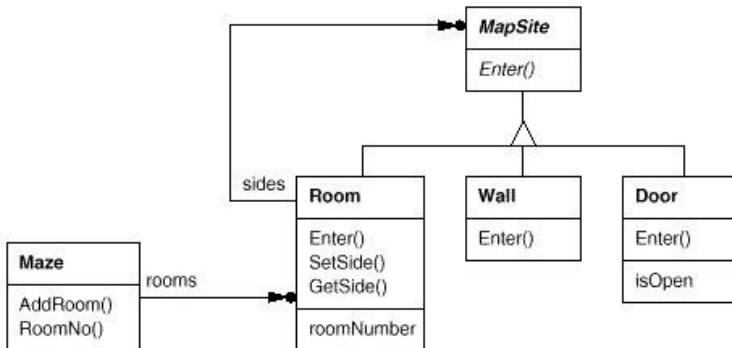
Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion



Exemple 3

The labyrinth : creation

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

```
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

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

- 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

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

- 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

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

- 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

Introduction

Factory Method

Structure

Exemples

Conclusion

Abstract Factory

Structure

Exemples

Conclusion

Builder

Introduction

Structure

Exemples

Conclusion

- **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

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

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**

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

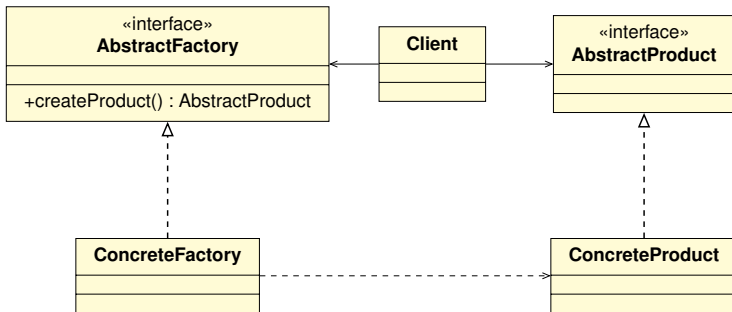
Builder

Introduction
Structure
Exemples
Conclusion

- **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** : Implémentation d'une `AbstractFactory`.

Abstract Factory

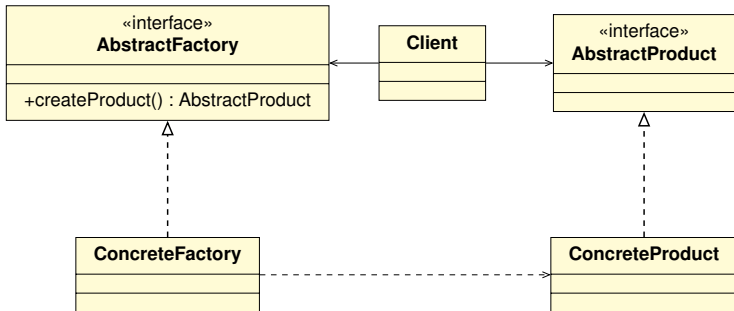
Schéma de principe



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

Abstract Factory

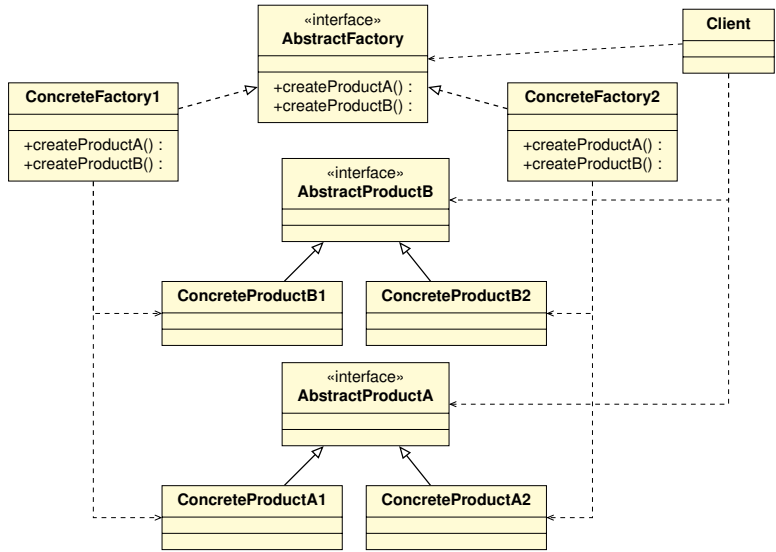
Schéma de principe



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

Abstract Factory

Schéma de principe : et là ?



Introduction

Factory
Method

Structure
Exemples
Conclusion

Abstract
Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

Principe

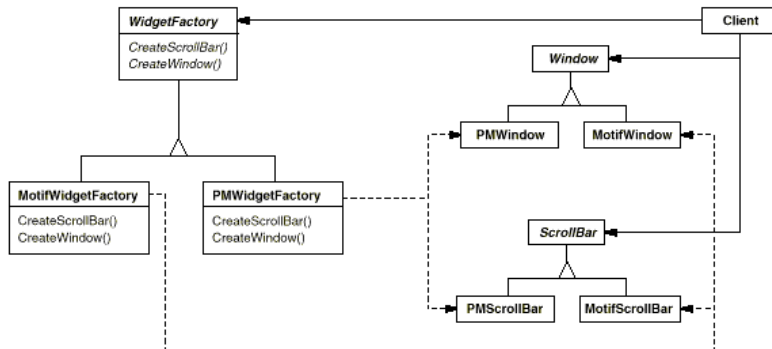
Fonctionnement

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 *ConcreteFactory* 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();^^! // Labyrinthe normale.
5
6         MazeGame mg=new EnchantedMazeGame();
7         Maze mz=mg.createMaze();^^! // Labyrinthe enchanté.
8     }
9 }
```

Principes respectés

Introduction

Factory Method

Structure

Exemples

Conclusion

Abstract Factory

Structure

Exemples

Conclusion

Builder

Introduction

Structure

Exemples

Conclusion

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

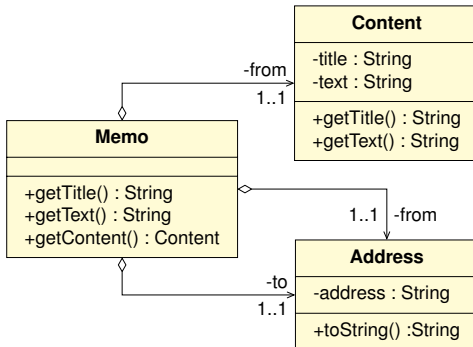
Le patron Builder

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

Situation de départ

Un exemple (très) simple

- On considère la classe `Memo` :



Situation de départ

Un exemple (très) simple - Code 1/2

- Code des composants de Memo :

```
1 package memo;
2
3 private class Content {
4     private String title;
5     private String text;
6     public Content(String title , String text) {
7         if ( title==null || text==null || title.length()>=20) throw new
            InvalidParameterException (...);
8
9         this.title = title;
10        this.text = text;
11    }
12    public String getTitle() { return title; }
13    public String getText() { return text; }
14 }
```

```
1 package memo;
2
3 public class Address {
4     private String address;
5     public Address(String adr) {
6         if (!ValidAddress.match(adr)) throw new InvalidParameterException
            (...);
7         address=adr;
8     }
9 }
```

Situation de départ

Un exemple (très) simple - Code 1/2

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

- Code de Memo :

```
1 package memo;
2
3 public final class Memo {
4     private Address from;
5     private Address to;
6     private Content content;
7
8     public Memo(Address from, Address to, Content content) {
9         if (from==null || to==null || content==null) throw new
10             IllegalArgumentException (...);
11
12         this.from=from;
13         this.to=to;
14         this.content=content;
15     }
16 }
```

- Pour instancier un mémo, ses composants doivent avoir été préparé au préalable.

Situation de départ

Chargement d'un flux XML

- On souhaite instancier un mémo en mémoire à partir d'un flux XML correspondant au schéma suivant :

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

- Par exemple :

```
1 <mémo xmlns="http://memocorp.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>
```


Situation de départ

Chargement d'un flux XML

- Un tel *parser* peut-être codé « facilement » à l'aide de l'API SAX.
- Où placer le code de chargement ?
- dans un constructeur de `memo` ? :

```
1 package memo;  
2  
3 public final class Memo {  
4     private Address from;  
5     private Address to;  
6     private Content content;  
7  
8     public Memo(Address from, Address to, Content content) {  
9         ...  
10    }  
11  
12    public Memo(String XMLFileName) {  
13        ...  
14    }  
15 }
```

- Bon, ... admettons² ...

2. Attention ! Derrière vous, une chauve-souris enragée !

Situation de départ

Chargement d'un flux XML

- Un tel *parser* peut-être codé « facilement » à l'aide de l'API SAX.
- Où placer le code de chargement ?
- dans un constructeur de `memo` ? :

```
1 package memo;  
2  
3 public final class Memo {  
4     private Address from;  
5     private Address to;  
6     private Content content;  
7  
8     public Memo(Address from, Address to, Content content) {  
9         ...  
10    }  
11  
12    public Memo(String XMLFileName) {  
13        ...  
14    }  
15 }
```

- Bon, ... admettons² ...

2. Attention ! Derrière vous, une chauve-souris enragée !

Situation de départ

Inclusion du code du parser XML

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

```
1 public class Memo {
2     ...
3
4     public Memo(String XMLFileName) throws ParserConfigurationException ,
        SAXException {
5         InputSource is= new InputSource(new BufferedInputStream(new
            FileInputStream(XMLFileName)));
6
7         SAXParserFactory spf = SAXParserFactory.newInstance();
8         SAXParser sp = spf.newSAXParser();
9         XMLReader xr = sp.getXMLReader();
10        System.out.println("Parser : " + xr.getClass().getName());
11        HandlerImpl handler = new HandlerImpl();
12        xr.setContentHandler(handler);
13        xr.setErrorHandler(handler);
14        try {
15            xr.parse(is);
16        } catch (Exception e) {
17            System.out.println(e);
18        }
19
20        return new Memo(handler.getFrom(), handler.getTo(), handler.getContent());
21    }
22    ... Wait it is not finished ! ...
```

Situation de départ

Inclusion du code du parser XML

```
1  ...
2  private class HandlerImpl extends DefaultHandler implements ContentHandler ,
    ErrorHandler {
3      private String title ;           // Champs à collecter
4      private String text ;
5      private String from ;
6      private String to ;
7
8      private List<String> stack ; // Pile d'analyse
9
10     public void startDocument() {
11         stack = new ArrayList<String>() ;
12     }
13
14     public void endDocument() throws SAXException {
15         stack.clear() ;
16     }
17
18     public void startElement(String uri , String localName , String qName,
19         Attributes atts) throws SAXException {
20         stack.add(qName) ;
21     }
22
23     public void endElement(String uri , String localName , String qName) throws
24         SAXException {
25         stack.remove(stack.size() - 1) ;
26     }
27     ...
```

Situation de départ

Inclusion du code du parser XML

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

```
1    ...
2
3    public void characters(char[] ch, int start, int length) throws
        SAXException {
4        String buf = new String(ch, start, length);
5        String top = stack.get(stack.size() - 1).trim();
6        if ("titre".equals(top)) title = buf;
7        else if ("texte".equals(top)) text = buf;
8        else if ("de".equals(top)) from = buf;
9        else if ("à".equals(top)) to = buf;
10       }
11   }
12 }
```

Situation de départ

Inclusion du code du parser XML

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

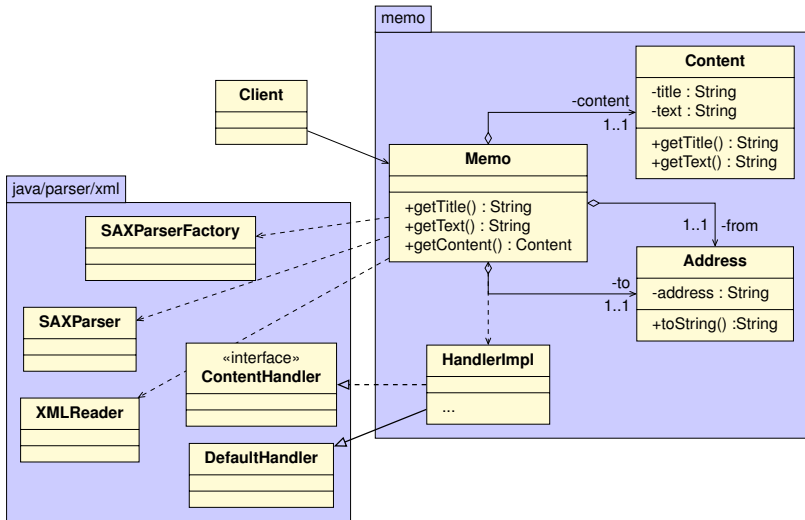
- Programme de test :

```
1  import memo;  
2  
3  public class Main {  
4      public static void main(String[] args) {  
5          try {  
6              // Ça c'est du constructeur :  
7              Memo memo = new Memo("memo.xml");  
8              System.out.println(memo);  
9          } catch (Exception e) {  
10             e.printStackTrace();  
11         }  
12     }  
13 }
```

Situation de départ

Inclusion du code du parser XML

- Ce que l'on a fait :



Situation de départ

Inclusion du code du parser XML - bilan

- Le code de la classe `Memo` se retrouve surchargé avec un code de construction complexe,
- Ce cas d'utilisation (chargement d'un flux XML) ne reflète pas un besoin de tous les clients de la bibliothèque `memo`,
- tous les clients se retrouvent cependant couplés à la bibliothèque SAX de parsing XML.
- Enfin, on pourrait imaginer avoir besoin d'autres techniques de sérialisation (ex : JSON, format maison). Faut-il imposer tous ces formats à tous les clients ?
- Le patron *Builder* va nous aider en externalisant la construction d'un agrégat...

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**

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

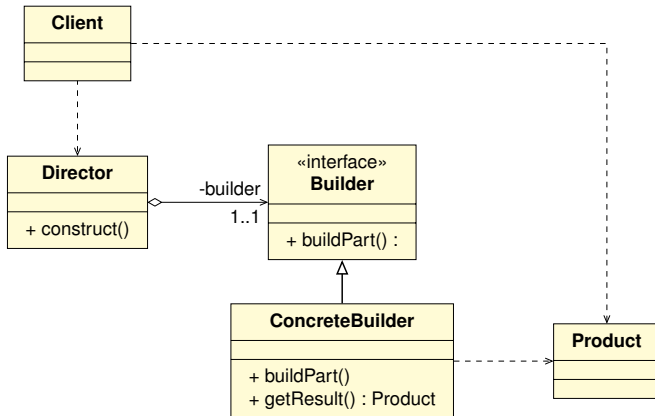
Builder

Introduction
Structure
Exemples
Conclusion

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

Builder

Schéma de principe



Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

Builder

Principe

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

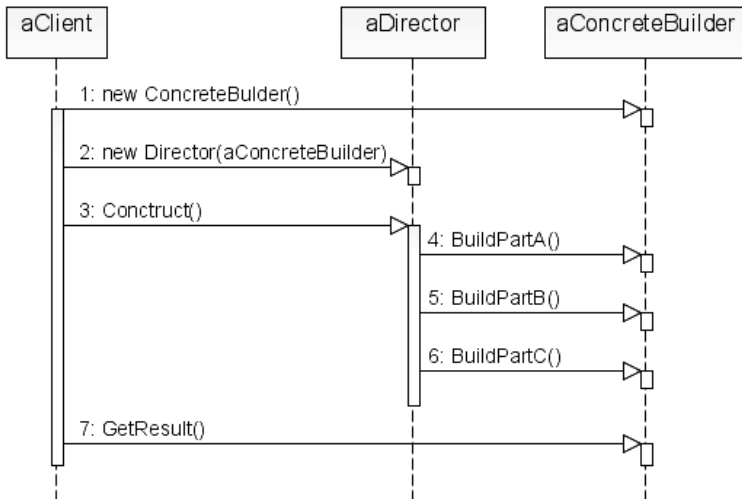
Builder

Introduction
Structure
Exemples
Conclusion

- Le client cherche à produire un objet `Product` qui représente en fait la tête d'un agrégat plus complexe.
- Le client se tourne vers un objet de type `Director` qui orchestrera toute la construction pour lui. Celui-ci encapsule toute la logique de collecte des informations de construction.
- Le directeur à son tour se tourne vers un objet du type `Builder` pour produire les composants (`buildPart()`) et les assembler (`getResult()`).

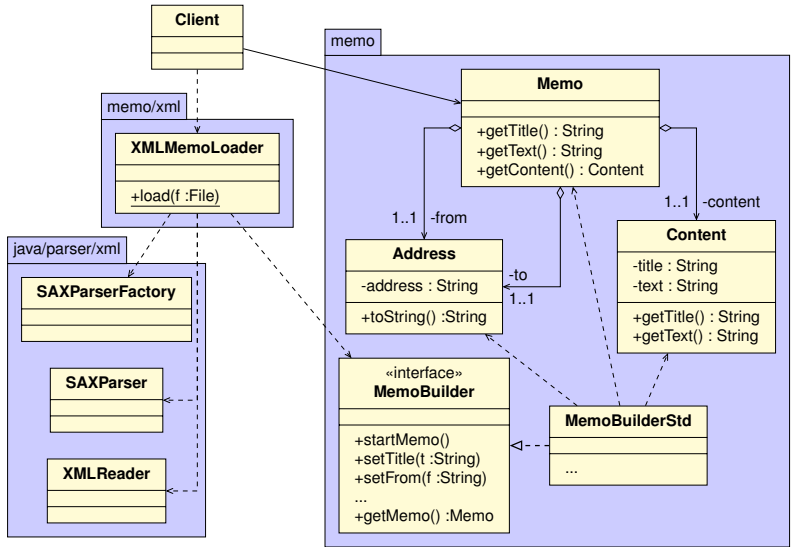
Builder

Coopérations



Retour à l'exemple

Externalisation parser XML



Retour à l'exemple

Externalisation parser XML

```
1 package memo ;
2
3 public interface MemoBuilder {
4     public void startMemo() ;
5     public void setTitle(String title) ;
6     public void setText(String text) ;
7     public void setFrom(String from) ;
8     public void setTo(String to) ;
9     public Memo getMemo() ;
10 }
```

- `MemoBuilder` définit une interface de construction.
- Note : les composants sont produits de façon abstraite à partir des données élémentaires (ex : `String` au lieu de `Address`).
- Remarque : nom des méthodes `buildPart()` (`addXXX()`, `setXXX()`).

Retour à l'exemple

Externalisation parser XML

```
1 package memo;
2
3 public class MemoBuilderStd {
4     private String title; private String text;
5     private String from; private String to;
6
7     public void startMemo() {
8         title=null; text=null; from=null; to=null;
9     }
10    public void setTitle(String title) {
11        if (title !=null) throw new InvalidOperationException("Title_àlready_
12            defined");
13        this.title=title;
14    }
15    public void setText(String text) { ... // idem }
16    public void setFrom(String from) { ... // idem }
17    public void setTo(String to) { ... // idem }
18    public Memo getMemo() {
19        if (title==null || text==null || from==null || to==null) throw new
20            InvalidOperationException("Component_àmissing...");
21        return new Memo(new Address(from), new Address(to), new Content(title ,
22            text)
23    );
24 }
```

- MemoBuilderStd est garant de la construction.

Retour à l'exemple

Externalisation parser XML

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

```
1 package memo.xml ;
2 import org.xml.sax.* ;
3
4 public class XMLMemoLoader {
5     public static void load(String XMLFileName, MemoBuilder mb) throws
6         ParserConfigurationException , SAXException {
7         InputSource is= new InputSource(new BufferedInputStream(new
8             FileInputStream(XMLFileName)) ;
9         SAXParserFactory spf = SAXParserFactory.newInstance() ;
10        SAXParser sp = spf.newSAXParser() ;
11        XMLReader xr = sp.getXMLReader() ;
12        HandlerImpl handler = new HandlerImpl(mb) ;
13        xr.setContentHandler(handler) ;
14        xr.setErrorHandler(handler) ;
15        try {
16            xr.parse(is) ;
17        } catch (Exception e) {
18            System.out.println(" " + e) ;
19        }
20    }
21    ...
22 }
```

Retour à l'exemple

Externalisation parser XML

```
1  ...
2  private class HandlerImpl extends DefaultHandler implements ContentHandler,
   ErrorHandler {
3      private List<String> stack;
4      MemoBuilder mb;
5
6      public HandlerImpl(MemoBuilder mb) {
7          this.mb = mb;
8      }
9
10     public void characters(char[] ch, int start, int length) throws SAXException
        {
11         String buf = new String(ch, start, length);
12         String top = stack.get(stack.size() - 1).trim();
13         if ("titre".equals(top)) mb.setTitle(buf);
14         else if ("texte".equals(top)) mb.setText(buf);
15         else if ("de".equals(top)) mb.setFrom(buf);
16         else if ("à".equals(top)) mb.setTo(buf);
17     }
18     public void endDocument() throws SAXException { stack.clear(); }
19     public void endElement(String uri, String localName, String qName) throws
        SAXException {
20         stack.remove(stack.size() - 1);
21     }
22     public void startDocument() { stack = new ArrayList(); }
23     public void startElement(String uri, String localName, String qName,
        Attributes atts) throws SAXException {
24         stack.add(qName);
25     }
26 }
```

Exemple 1

Chargement d'un flux XML

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

- Programme de test :

```
1  import memo.xml; // Parce que je le veux bien !
2
3  public class Main {
4      public static void main(String[] args) {
5          MemoBuilder memoBuilder = new MemoBuilderStd();
6
7          try {
8              XMLMemoLoader.load("memo.xml", memoBuilder);
9              Memo memo = memoBuilder.getMemo();
10             System.out.println(memo);
11         } catch (Exception e) {
12             e.printStackTrace();
13         }
14     }
15 }
```

Principes respectés

Introduction

Factory Method

Structure
Exemples
Conclusion

Abstract Factory

Structure
Exemples
Conclusion

Builder

Introduction
Structure
Exemples
Conclusion

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

Quelques références

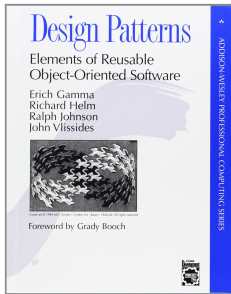
Design Patterns : Elements of Reusable

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.