

Patrons de création/construction

- La création des objets :
 - Sources de dépendances
 - Classes concrètes (sauf réflexion)
 - Tout est static
 - Constructeurs parfois complexes à appeler
 - La construction / l'initialisation des objets peut être une vraie responsabilité (exemple MK2)
 - Parfois dangereux : interdit de créer plusieurs objets



Patrons de création/construction

- C'est mieux
- Mais avez vous toujours besoin de Meetic pour avoir des amis, le site est-il toujours disponible ?
- Une meilleure solution est une interface de création bien définie
- une factory pour la factory !**

```
CopainProvider copp = CopainProvider.newInstance();  
...  
...  
Copain f = copp.newCopain(Eyes.blue, Hair.Black);
```



Patrons de création/construction

- Copain f = new Fred(); // facile la création
 - mais avez vous toujours besoin de Fred ?
 - est il toujours disponible ?
- Ce new est bloquant d'un point de vue syntaxique
 - le type de la classe est une constante
 - Le code doit être modifié "à la main" (hard-coded).
- Utilisons une classe pour cacher ce new.

```
Meetic m = new Meetic();  
...  
...  
Copain f = m.newCopain();
```



Patrons de création/construction

- Factory (Usine) : une classe responsable du type d'instance effectivement créée.
 - Pas dans le Gof, version générique
- Method Factory: Une méthode de classe pour créer les instances.
- Abstract Factory (Kit de construction): une classe qui permet de créer une famille d'instances de classes liées.
- Builder : Quand cela devient compliqué de construire une instance, demander un spécialiste.
- Prototype : Quand copier est une bonne idée
- Singleton : Il y a des objets qui ne supportent pas d'être à deux



Patrons de création/construction

- Factory (Usine) : une classe responsable du type d'instance effectivement créée.
- Abstract Factory: une classe qui gère une multitude d'instances de classes.
- Builder : Quand on a une bonne idée d'une instance à construire.
- Prototype : une classe qui ne supporte pas d'être à l'origine d'un marigot.

Rappel
une responsabilité => un objet

Factory (! GoF)

- Factory V1 : une nouvelle classe possède des méthodes statiques qui créent les nouvelles instances,
 - Remarque sur les dépendances: le « constructeur » et les classes créées sont dans le même paquetage.
- Factory V2 : une nouvelle classe possède des méthodes non statiques qui créent les nouvelles instances,

Factory/Factory method : quand ?

- "new" impose le type
 - laisser le client instancier l'objet avec new augmente la rigidité
- le type réel a-t-il besoin d'être connu ?
 - Token t3 = new IntToken(3);
 - Token tlf = new KeywordIfToken();
- La hiérarchie se doit-elle d'être publique et stable ?
 - Token t3 = new IntToken(3);
 - ou
 - Token t3 = new TokenConstant(3);
- La classe a instancier existe-t-elle/est-elle connue ?
 - (quand je compile mon code ?)

Factory method : comment ?

- Factory method :
 - Fournir une méthode abstraite 'newXX()', en laissant la responsabilité aux sous-classes d'instancier l'objet du bon type
 - Remarque sur les dépendances: les classes dérivées, et donc le newXX() ne sont pas forcément dans le même paquetage.

Factory method : exemple

```
public class HouseStd {
    Wall createWall(...) {
        return new WallStd(...);
    }
}
```

Implémentation par défaut optionnelle

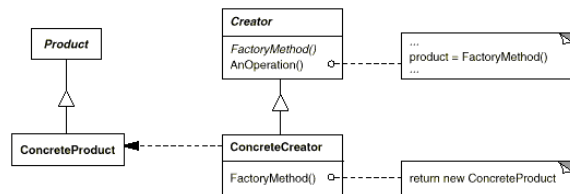
```
public interface Wall
{
    ...
}
```

```
public class WoodHouse extends HouseStd {
    Wall createWall(...) {
        return new WoodWall(...);
    }
}
```

Abstract Factory (Kit) : quand ?

- Plus compliqué : nous travaillons avec un ensemble de classes ou d'objets interdépendants
- 1) if (user.pref == "rouge") return new RougeWall ...
 - 2) RougeFactory.newWall();
 - 3) ElementFactory kit =
ElementFactories.newInstance(user.pref);
kit.newWall();
kit.newRoof();

Factory method



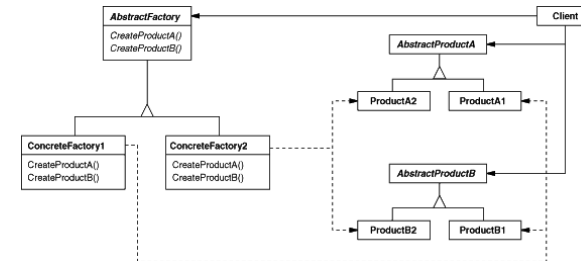
Abstract Factory : quand ?

- On veut pouvoir choisir une famille d'implémentations en fonction du choix de l'utilisateur ou du système
- LookAndFeel, AWT
- On veut pouvoir choisir en fonction du service demandé
 - JDBC : tout est fait à partir de l'objet Driver (qui dépend du type du SGBD) qui fournit une interface commune quelle que soit la base (MySQL, PostgreSQL, Oracle...)
- La manière d'utiliser l'ensemble des classes ne doit pas dépendre du choix d'implémentation, qui peut même être fait à *runtime*

Abstract Factory : comment ?

- Déclarer l'interface de l'abstract factory : un ensemble de méthode create qui permettent d'instancier les différents objets
- Afin de respecter le principe d'encapsulation, les méthodes create retournent un type abstrait
- À chaque cas de figure correspondra une implémentation de l'abstract factory, et la création de tous les objets sera déléguée à cette instance
- Les méthodes create ne sont a fortiori plus statiques
- L'autre nom est kit de construction

Abstract Factory



Abstract Factory : exemple

```

public interface ElementFactory {
    Wall createWall(...);
    Room createRoom(...);
    Door createDoor(...);
}
    
```

```

class RedElementFactory
implements ElementFactory {
    public Wall createWall(...) {
        return new RedWall(...);
    }

    public Room createRoom(...) {
        return new RedRoom(...);
    }

    public Door createDoor(...) {
        return new RedDoor(...);
    }
}
    
```

```

public interface Wall
{
    ...
}
    
```

```

public interface Room
{
    ...
}
    
```

```

public interface Door
{
    ...
}
    
```

Prototype : quand ?

- Ce qui compte, c'est la valeur des champs des instances
- Door door1 = factory.createDoor(3,3,56,7.6,4.5,8);
- Door door2 = factory.createDoor(3,3,56,7.6,4.5,8);
- En plus, le client ne connaît pas les bonnes valeurs
- Door door1 = factory.createDoor(config.x,config.y,...)
- On se dit, pour door2, je ne vais pas copier-coller
- Door door2 = (Door)door1.clone();
- Et oui ! une responsabilité en plus : la copie
- Souvent couplé à une Factory qui gère les prototypes à utiliser

Prototype : quand ?

- Quand ce sont surtout les valeurs d'initialisation des objets créés qui importent
- Et que, comme toujours, le client les ignore (et pour cause, les valeurs des champs d'une implémentation donnée)



Prototype : exemple

```
class PrototypeElementFactory {  
    private final Wall3D wallPrototype;  
    private final Room3D roomPrototype;  
    private final Door3D doorPrototype;  
  
    public Wall createWall(Direction dir) {  
        Wall3D wall = wallPrototype.clone();  
        wall.setDirection(dir);  
        return wall;  
    }  
    public Room createRoom(String info) {  
        return roomPrototype.clone();  
    }  
    public Door createDoor() {  
        return doorPrototype.clone();  
    }  
    public PrototypeElementFactory() {  
        // créer les prototypes  
    }  
}
```



Prototype : comment

- Implémenter une factory ou une abstract factory qui copie un objet de référence pour instancier un nouvel objet
 - Ces objets de référence peuvent être passés en paramètres à la création de la factory
- Éventuellement, les méthodes de création peuvent prendre des paramètres à changer sur la copie

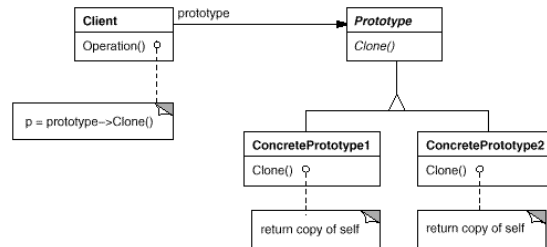


Prototype : exemple

- Dans une interface graphique, l'utilisateur configure l'aspect d'un bouton (fonte, type du cadre, position de l'icône, ...)
- Pour construire la barre d'outil, le bouton prototype est dupliqué, et seul l'icône, le texte et l'action sont changés



Prototype



Singleton

- Assurer qu'il n'existe qu'une et une seule instance d'une classe
- Fournir un point d'accès global à cette instance
- Permettre l'héritage
- Permettre l'instanciation à la demande ou à l'initialisation



Singleton : quand

- Certaines classes ne doivent avoir qu'une seule instance
 - Application
 - Player
- Il y a des classes « boîtes à outils » (en général plein de méthodes statiques)
 - Mais on ne peut pas en hériter (ajout/modification)



Singleton : comment

- une méthode static contrôle l'instanciation et l'accès à l'unique instance (lazy ou non)
- on peut étendre à doubleton...

```

public abstract class Player {
    abstract void move(Direction d);
    abstract void yell(String message);

    private static final Player INSTANCE = new PlayerImpl();

    public static Player getInstance() {
        return INSTANCE;
    }
}

public class PlayerImpl extends Player { ... }
    
```



Singleton : exemple (lazy)

- Ne choisir le mode lazy que quand c'est utile (pas souvent !) :
 - quand l'instance est un gros objet ou long à initialiser,
 - et qu'on en ait pas toujours besoin

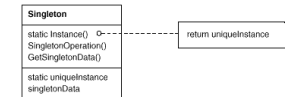
```
public abstract class TranslateService {
    public abstract String translate(Language from, Language to, String word)
        throws UnsupportedOperationException;

    private static TranslateService INSTANCE; /* = null */

    public synchronized static TranslateService getInstance() {
        if (INSTANCE == null)
            INSTANCE = new TranslateServiceImpl();
        return INSTANCE;
    }
}
```



Singleton



Singleton ...

- Attention à la singleton-mania ! Des singletons partout sans prendre le temps de réfléchir
- A-t-on vraiment besoin du Singleton ?
 - Se rappeler des besoins principaux
 - instance unique
 - point d'accès global (pas de propriétaire « naturel »)
 - Initialisation lazy (possibilité de le faire maintenant ou plus tard)
 - Sinon, le Singleton peut devenir une classe cachée
- Facile d'adapter le Singleton pour permettre un petit nombre d'instances



Singleton's : correct ?

```
public class KpiService {
    private static KpiService instance = null;

    public static KpiService getInstance() {
        return instance;
    }

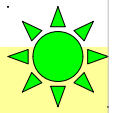
    public KpiService() {
        instance = this;
    }
}

public class DB {

    public static DB getDB() {
        return Cache.singleton;
    }

    private DB() {
    }

    private static class Cache {
        static final DB singleton = new DB();
    }
}
```



Builder : quand

- Construire un objet devient trop complexe
- Les différents cas possibles sont trop nombreux (combien de maisons différentes ?)
- On veut tout de même pouvoir fournir différentes implémentations
- On veut tout de même contrôler la création



Builder : exemple

```
public interface MazeBuilder {
    Maze getMaze();
    void addRoom(Location location, Room room);
    void addDoor(Location location, Door door);
    void addWall(Location location, Wall wall);
    void setDecorType(DecorFactory factory);
    void createPath(Location from, Location to, ElementFactory factory);
}
```

```
MazeBuilder mazeBuilder = MazeBuilderFactory.newInstance();
mazeBuilder.setDecorType(decorMap.get(userPref.get(Pref.DECOR)));
mazeBuilder.addRoom(to, monsterRoom);
mazeBuilder.addRoom(from, entranceRoom);
mazeBuilder.createPath(from, to, elementFactory);
```

```
Maze = mazeBuilder.getMaze();
```



Builder : comment

- On confie à une classe appelée Builder la responsabilité de construire l'objet
- Contrairement à la factory ou l'abstract factory, il peut posséder un état
- La construction de l'objet correspond à l'appel de plusieurs méthodes ; la dernière étant en général celle qui retourne l'instance
 - Construction par étapes
- En revanche, c'est bien le client du builder qui contrôle la création (choisit les caractéristiques de l'objet créé)



Builder : exemple

```
public interface MazeBuilder {
    Maze getMaze();
    void setEntranceLocation(Location entrance);
    void setMinibossesLocation(Location[] locations);
    void setBossLocation(Location boss);
    void setMonsterTypeFrequency(Map<Monster, Double> map);
    void setElementFactory(ElementFactory factory);
    void setSmoothness(double smoothness);
    void setDungeonType(DungeonType type);
}
```

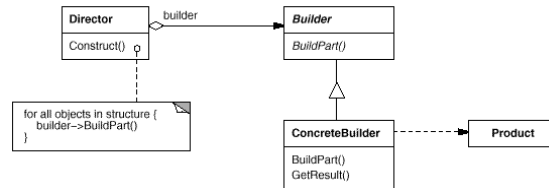
```
public interface DocumentBuilder {
    Text createTextField(String text);
    List createListItems(Items[] items);
    ...
}
```

```
public class StringBuilder { ... }
```

```
public class WordBuilder implements DocumentBuilder { ... }
public class ASCIIBuilder implements DocumentBuilder { ... }
public class HTMLBuilder implements DocumentBuilder { ... }
public class TeXBuilder implements DocumentBuilder { ... }
```



Builder



Copyright GoF



Builder

convertir du RTF vers différents formats

