Université Nice Sophia Antipolis

Patron de création: Usine, Singleton

F. Mallet

miage.m1@gmail.com

http://deptinfo.unice.fr/~fmallet/

Université Nice Sophia Antipolis

Motivation

- Les besoins pour une bonne conception et du bon code :
  - Extensibilité
  - Flexibilité
  - Facilité à maintenir
  - Réutilisabilité
  - Les qualités internes
  - Meilleure spécification, construction, documentation

Université Nice Sophia Antipolis

Historique

- MVC
- Gang of Four : Erich Gamma, Richard Helm,Ralph Johnson, John Vlissides
  - Définition de 23 patterns
- Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley, 1994
- Un Design Pattern nomme, abstrait et identifie les aspects essentiels d'une structuration récurrente, ce qui permet de créer une modélisation orientée objet réutilisable

Université Nice Sophia Antipolis

Classification

- 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)

Université Nice Sophia Antipolis

Factory / Usine

- Intention
  - Choisit la bonne sous-classe en fonction de certains paramètres
  - Permet la délégation d'instanciation : cache la classe concrète utilisée (permet l'évolution)
- Exemple
  - Instancier des classes en connaissant uniquement les classes abstraites
  - **`java.util.Calendar`** et **`java.util.GregorianCalendar`**
- *Synonyme : constructeur virtuel*

Université Nice Sophia Antipolis

java.util.Calendar

- La classe **`Calendar`**
  - Constructeur *protected* et classe abstraite !
  - Plusieurs classes concrètes (Gregorian, Japanese-traditional)
  - La configuration d'un calendrier dépend de choix locaux
    - Premiers jours de la semaine, début de l'ère, TimeZone, heure d'été
- Créer une instance de calendrier
  - Calendar cal = Calendar.getInstance()
    - Obtient le calendrier configuré avec les paramètres locaux

Codec

- Erreur si rien ne convient !
- Ex:

```
class UsineDecodeurImage {
  DecodeurImage getDecodeur(InputStream is) {
    byte[] buf = new byte[3];
    is.read(buf);
    if (buf[0]=='G' && buf[1]=='I' && buf[2]=='F')
      return new DecodeurGif(is);
    if (buf[0]=='0x42' && buf[1]=='0x4D')
      return new DecodeurBMP(is);
    throw new RuntimeException("Type inconnu !");
  }
}
```

```
interface DecodeurImage {
  Image decode(InputStream is);
}
```

```
class DecodeurGif implements
    DecodeurImage { … }

class DecodeurBMP implements
    DecodeurImage
   { … }
```

Ecosystème

- Champs d'application
  - On se sait pas quel constructeur invoqué !
  - Cas par défaut si rien ne convient

- Ex:

```
class UsineBebete {
  IBebete getBebete(String type) {
    if (type.equals("elephant"))
      return new Elephant();
    if (type.equals("oiseau"))
      return new Oiseau();
    return new BebeteStupide();
  }
}
```

```
interface IBebete {
  void agit();
}

class Elephant implements IBebete
   { … }

class Oiseau implements IBebete
   { … }

class BebeteStupide implements
   IBebete { … }
```

Ecosystème

- Champs d'application
  - Toutes les bébêtes évoluent dans le même champ !

- Ex:

```
class UsineBebete {
  ChampBebete champ;
  IBebete getBebete(String type) {
    if (type.equals("elephant"))
      return new Elephant(champ);
    if (type.equals("oiseau"))
      return new Oiseau(champ);
    return new BebeteStupide(champ);
  }
}
```

```
interface IBebete {
  void agit();
}

class Elephant implements IBebete
   { … }

class Oiseau implements IBebete
   { … }

class BebeteStupide implements
   IBebete { … }
```

Nombres complexes

- Variante :

```
class UsineComplexe1 {
  static IComplexe getReIm(double re, double im) {
    return new ComplexeReIm(re, im);
  }
  static IComplexe getNoPh(double no, double ph) {
    return new ComplexeReIm(no*Math.cos(ph),
                            no*Math.sin(ph));
  }
}
```

```
interface IComplexe {
  double getReelle();
  double getImaginaire();
}

class ComplexeReIm implements IComplexe
    { double re, im; }
```

Nombres complexes

- Variante :

```
class UsineComplexe2 {
  static IComplexe getReIm(double re, double im) {
    return new ComplexeReIm(re, im);
  }
  static IComplexe getNoPh(double no, double ph) {
    return new ComplexeNoPh(no, ph);
  }
}
```

Est-ce que le static est nécessaire ?

```
interface IComplexe {
  double getReelle();
  double getImaginaire();
}

class ComplexeReIm implements IComplexe
    { double re, im; }

class ComplexeNoPh implements IComplexe
   { double norme, phase; }
```

Abstract Factory / Usine abstraite (1/7)

- Intention
  - Fournir une interface pour créer des familles d'objets dépendants ou associés sans connaître leur classe réelle
  - Fabriquer des fabriques.
- Exemple :
  - Génération d'écosystème pour les bébêtes
    - Un écosystème contient plus d'éléphants que d'oiseau, seulement des peureux, …
  - Fabriquer des *widgets* qui ont tous le même *look&feel*
- *Synonymes : Kit, Fabrique abstraite, Usine abstraite*

Abstract Factory / Usine abstraite (2/7)

- Variante :

```
interface IUsineComplexe {
  IComplexe getReIm(double re, double im);
  IComplexe getNoPh(double no, double ph);
}
class UsineComplexeReIm implements IUsineComplexe {
  IComplexe getReIm(double re, double im) {
    return new ComplexeReIm(re, im);
  }
  IComplexe getNoPh(double no, double ph) {
    return new ComplexeReIm(no*Math.cos(ph),
                            no*Math.sin(ph));
  }
}
```

Abstract Factory / Usine abstraite (3/7)

- Variante :

```
interface IUsineComplexe {
  IComplexe getReIm(double re, double im);
  IComplexe getNoPh(double no, double ph);
}
class UsineComplexeNoPh implements IUsineComplexe {
  IComplexe getReIm(double re, double im) {
    return new ComplexeNoPh(…, …);
  }
  IComplexe getNoPh(double no, double ph) {
    return new ComplexeNoPh(no, ph);
  }

}
```

Abstract Factory / Usine abstraite (4/7)

- Champ d'application
  - Une usine à usines

- Ex:

```
class AUsineEcosysteme implements
   IUsineEcosysteme{
  IEcosysteme getEcosysteme(int type)
   {
    if (type==1)
      return new Ecosysteme1();
    if (type==2)
      return new Ecosysteme2();
    return new EcosystemeDefaut();
  }
}
```

```
interface IUsineEcosysteme {
  IEcosystem getEcosysteme(int type);
}
}
interface IEcosysteme {
  IBebete getBebete();
}
class Ecosystem1 implements
    IEcosysteme {
// Au moins 60% d'éléphant
}

class Ecosystem2 implements
    IEcosysteme {
// Plus de 80% d'oiseaux
}

class EcosystemDefaut implements
    IEcosysteme {
// autant d'oiseaux que d'éléphants
}
```

Abstract Factory / Usine abstraite (5/7)

- Champ d'application
  - Look & Feel

- Ex:

```
class abstract UsineIHM {
  static UsineIHM getUsine() {
    switch(readOsType()) {
    case Os.WINDOWS:
      return new UsineIHMWindows();
    case Os.MOTIF:
      return new UsineIHMMotif();
    default:
      return new UsineIHMMetal();
  }
  abstract IButton createButton();
}
```

```
interface IButton {
}

class UsineIHMWindows extends UsineIHM
{
  IButton createButton() {
    return new ButtonWin();
  }
}
class UsineIHMMotif extends UsineIHM {
  IButton createButton() {
    return new ButtonMotif();
  }
}
class UsineIHMMetal extends UsineIHM {
  IButton createButton() {
    return new ButtonMetal();
  }
}
```

Abstract Factory / Usine abstraite (6/7)

- Conséquences
  - Isolation des classes concrètes (seules les classes abstraites sont connues)
  - Échange facile des familles de produit
  - Encouragement de la cohérence entre les produits
  - Prise en compte difficile de nouvelles formes de produit

Abstract Factory / Usine abstraite (7/7)

- Implantation
  - Les fabriques sont souvent des singletons
  - Ce sont les sous-classes concrètes qui font la création, en utilisant le plus souvent une Factory
  - Si plusieurs familles sont possibles, la fabrique concrète utilise Prototype

Singleton (1/5)

- Intention
  - S’assurer qu’une classe a une seule instance, et fournir un point d’accès global à celle-ci.
  - Variable globale « améliorée »
- Exemple
  - Un seul window manager, un seul point d’accès à une base de donnée, etc.
- Champs d’application
  - Lorsque l’instance unique doit être extensible par héritage, et que les clients doivent pouvoir utiliser cette instance étendue sans modifier leur code

Singleton (2/5)

- Structure
  - un champs *public* static INSTANCE
  - un constructeur *private* (pas d'héritage)
  - Ex:

```
class A {
  final public static A INSTANCE = new A();
  private A() { … }
}
```

A.INSTANCE

- Problème
  - Instance unique immutable
  - Pas d'extension possible !

Singleton (3/5)

- Structure
  - un champs *private* static **`INSTANCE`**
  - un constructeur *private/protected* (si héritage)
  - une méthode public **`getInstance()`**
  - Ex:

```
class A {
  final private static A INSTANCE = new A();
  private A() { … }
  static public A getInstance() { return INSTANCE; }
}
```

`A.getInstance()`

Singleton (4/5)

- Variante
  - Un nombre connu d'instances >1
  - Ex:

```
class A {
  final private static A[] INSTANCES = new A[10];
  final private static int count = -1;
  private A() { … }
  static public A getInstance() {
    ++ count;
    if(A[count%INSTANCES.length] == null)
      A[count % INSTANCES.length] = new A();
    return A[count % INSTANCES.length];
  }
}
```

```
A.getInstance()
```

Singleton (5/5)

- Implémentation
  - Assurer l'unicité
  - Sous-classer (demander quelle forme de singleton dans la méthode **getInstance()**)
    - Une seule bébête de type `Dieu`
    - 10 bébêtes de type `Elephant` (espèce en voie d'extinction)
    - => Factory
- Utilisations connues
  - DefaultToolkit en AWT/Java : java.awt.DefaultToolkit.getDefaultToolkit()