Design Patterns, patrons de conception

Bernard Hugueney

Contents

1 Objectifs Pédagogiques **1**

2 Qu'est-ce qu'un *design pattern* ? **2**

3 Différents niveaux conceptuels **3**

4 Styles architecturaux **3**

5 Patrons architecturaux **4**

6 Design patterns proprement dits **4**
6.1 Optional . . . 4
6.2 Construction polymorphique → Factory (Usine) . . . 4
6.2.1 Exercice . . . 5
6.3 Changement de comportement d'un objet : *Stratégie* . . . 5
6.3.1 Exercice . . . 5
6.4 Une seule instance : le Singleton (anti-pattern ?) . . . 6
6.4.1 Exercice . . . 6
6.5 Agir sur un ensemble de valeurs : Visiteur . . . 6
6.5.1 Exercices . . . 7
6.6 Réaction à des modifications : Observer . . . 7
6.6.1 Exercice . . . 8
6.7 Injection de dépendance . . . 8

7 Design patterns ? **8**
7.1 Limitation d'effets de bords . . . 8
7.2 Programmation Orientée Valeurs (immutables) . . . 8
7.3 Programmation Orientée Mathématiques ? . . . 8

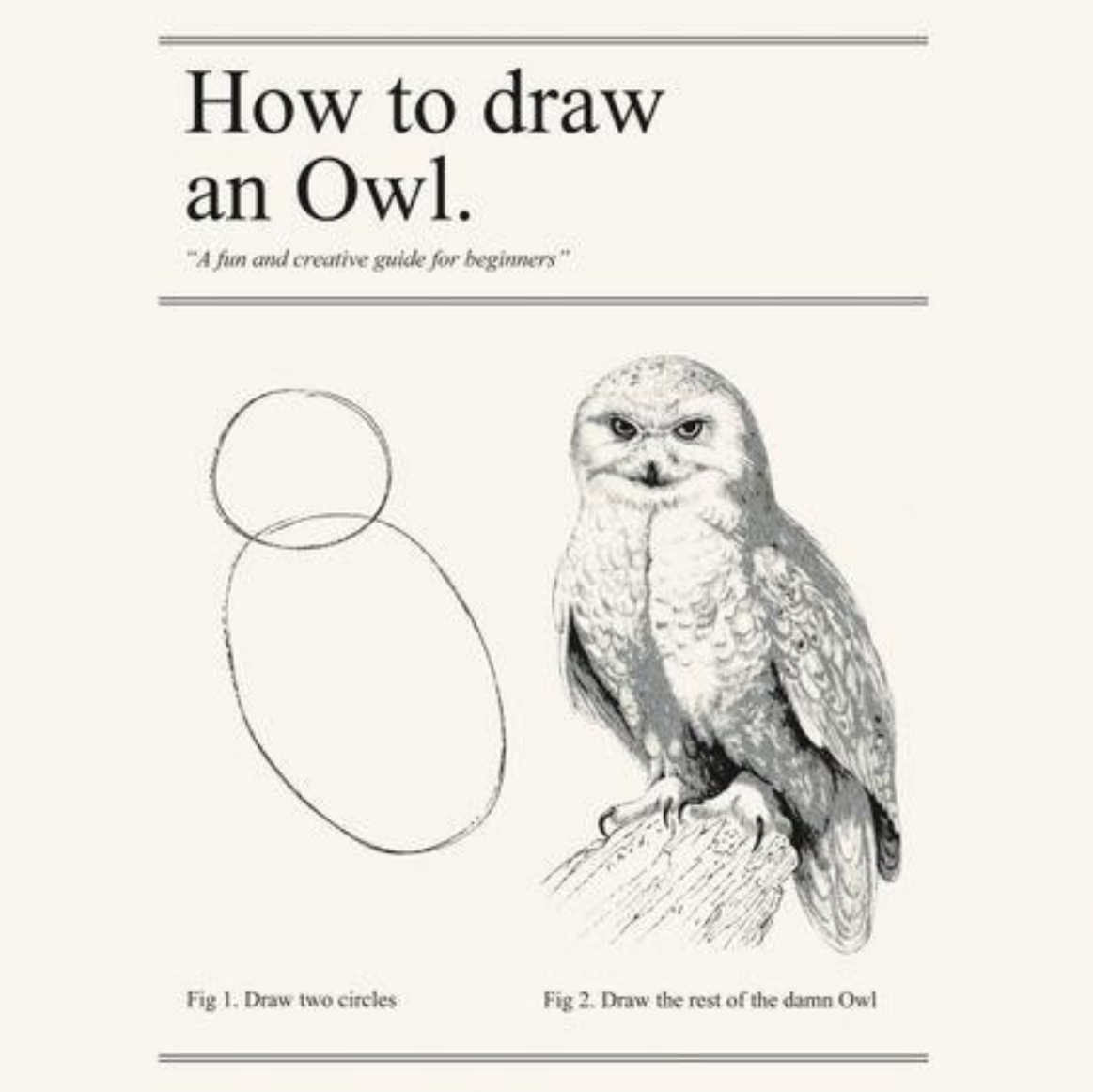
1 Objectifs Pédagogiques

Les objectifs pédagogiques de cette session sont théoriques et pratiques (pas forcément dans cet ordre !) :

- sur le plan théorique, comprendre ce que sont des *design patterns* et surtout à quoi il servent
- sur le plan pratique, connaître quelques *design patterns* et surtout savoir utiliser

2 Qu'est-ce qu'un *design pattern* ?

En fait, il y a plusieurs définitions et plusieurs sous-catégories de *design patterns*. De façons générale, les *designs pattern* sont des modèles pour aider à décider comment coder. En effet, connaître le(s) langages de programmation et même les bibliothèques disponibles n'est pas suffisant pour écrire un programme un peu conséquent. De même que savoir utiliser un crayon pour tracer des formes élémentaires n'est pas suffisant pour pouvoir dessiner quelque chose de complexe :



De même, pour construire des bâtiments, il ne suffit pas de savoir assembler des briques / pierres / parpaings. Comme les possibilités sont infinies, il est utile de se référer à un "catalogue" de solutions aux différentes échelles :

- types de bâtiments :
  - maison individuelle ou non
  - immeuble
  - garage / atelier
  - ...

- types de pièces :
  - chambres
  - salon
  - salle de bain
  - ...
- ...

À chaque fois, des besoins ont été identifiés et des solutions développées font partie d'un "répertoire" de bonnes (?) pratiques. Le double avantage est que non seulement ses solutions sont éprouvées, mais elles sont facilement comprises par ceux/celles qui les connaissent.

On distingue le niveau "architecture" du niveau "réalisation", même s'il s'agit en fait d'un continuum de décisions prises sur la façon de réaliser la solution.

L'architecture, c'est ce qui relève de décision prises en **amont** et qui impacte le reste. Donc ce qui relève de l'architecture, c'est ce qui est coûteux (difficile et long) à changer. D'où l'intérêt de s'inspirer de solutions éprouvées !

3 Différents niveaux conceptuels

On peut distinguer les trois niveaux (de l'amont à l'aval en matière de conception / développement) suivants dans le concept général de *design pattern* :

- Styles architecturaux (*architectural styles*)
- Patrons architecturaux (*architectural patterns*)
- Patrons de conception (*design patterns*) proprement dits

Si les *design patterns* ne peuvent pas être implémentés de façon générale (c'est-à-dire réutilisable) dans un langage de programmation (lorsqu'ils le peuvent, il ne sont plus considérés comme des *design patterns*), ils le sont parfois dans des *frameworks*.

4 Styles architecturaux

Il s'agit de l'organisation très générale de l'application, au plus haut niveau d'abstraction, avec souvent des sous-catégories :

- application monolithique
- client / serveur
  - client léger ou lourd
  - 2-tiers, 3-tiers, n-tiers
- pair-à-pair
  - hiérarchisés ou non
- ...

5 Patrons architecturaux

Il s'agit de façons d'organiser le code pour implémenter le style architectural, souvent pour répondre à des aspects précis de réutilisabilité et d'extensibilité :

- Model View Control
- Services Controller DataAccess
- Architecture Hexagonale
- ...

6 Design patterns proprement dits

Par définition, chaque *desing pattern* correspond à quelque chose que l'on ne peut pas implémenter directement dans le langage, par exemple avec un appel de méthode en POO (Programmation Orientée Objet).

6.1 Optional

Une fonction qui peut retourner ou non une valeur pourrait retourner une référence à `null` dans ce cas. Il faudrait alors toujours penser à vérifier si la référence est `null` avant de faire quoi que ce soit avec sous peine d'avoir une `NullPointerException`.

java.util.Optional rend obligatoire cette vérification (évidemment, il ne faut pas utiliser directement la méthode get, sinon on échange juste une exception contre une autre !).

6.2 Construction polymorphique → Factory (Usine)

On rappelle que le polymorphisme, c'est le fait que différentes implémentations puissent être appelées de la même façon, c'est à dire avec le même code. Cela permet donc de **réutiliser** ce code pour plusieurs implémentations. En POO, cela est réalisé avec la notion d'héritage :

```
1  import java.sql.Connection; // une interface !
2  import java.sql.Statement; // (idem)
3
4  public class A {
5      private Connection conn;
6      public A(String arg){
7          conn = new AClassImplementingConnection(arg);
8      }
9      public void f(){
10         Statement st= conn.createStatement();
11         //...
12     }
13     public void g(){
14         Statement st= conn.createStatement();
15         //...
16     }
17 }
```

Toutes les méthodes de la classe `A` sont indépendantes de la classe de l'attribut `conn` puisque celui-ci n'est manipulé qu'à travers l'interface `Connection`. On pourrait donc choisir une autre classe d'implémentation de cette interface sans avoir modifier aucune de ces méthodes.

En revanche, le constructeur de l'attribut est lui forcément lié à la classe.

Pour résoudre ce problème et achever le découplage, on ajoute (comme souvent) un niveau d'indirection en construisant l'instance à l'aide d'une 'usine' (*Factory*). C'est d'ailleurs la solution choisie en Java

qui permet de récupérer une instance d'une classe implémentation l'interface Connection à l'aide de la méthode DataSource.getConnection().

6.2.1 Exercice

Faire une Factory qui puisse retourner une instance de Connection issue d'un driver `org.h2.Driver` ou d'un driver `org.postgresql.Driver` en ne chargeant (avec `Class.forName()`) que le driver utilisé.

Souvent, la *Factory* est utilisée avec/pour de l' *injection de dépendance* (cf. infra) afin de ne pas voir à mettre dans le code même les arguments qui vont déterminer la classe instanciée.

6.3 Changement de comportement d'un objet : *Stratégie*

Les différents comportements possibles sont souvent implémentés par différentes définitions dans différentes classes. Par exemple :

```
1  public interface Animal{
2      public void seDeplace();
3  }
```

```
1  public class AnimalRampant implements Animal{
2      public void seDeplace(){
3          System.out.println("Je rampe");
4      }
5  }
```

```
1  public class AnimalVolant implements Animal{
2      public void seDeplace(){
3          System.out.println("Je vole");
4      }
5  }
```

Le problème survient quand on voudrait implémenter une classe qui pourrait changer en cours de vie. La encore, un niveau d'indirection peut nous aider avec le *design pattern* Stratégie :

```
1  public class Papillon implements Animal{
2      Animal forme;
3      public Papillon(){
4          forme= new AnimalRampant();
5      }
6      public void mue(){
7          forme= new AnimalVolant();
8      }
9      public void seDeplace(){
10         forme.seDeplace();
11     }
12 }
```

En pratique, on aura plutôt une interface qui ne concerne que la ou les méthodes concernée qui sera donc différente (plus 'petite', ici juste la locomotion) que celle implémentée par la classe utilisant un attribut *Stratégie* (ici l'interface `Animal`).

6.3.1 Exercice

Soit une classe `User` implémentant, notamment, une méthode `access()`. On veut que cette méthode ait différentes implémentations suivant que le `user` soit :

- non authentifié

- simple utilisateur enregistré
- administrateur

Réalisez une implémentation à l'aide du *design pattern* Strategie.

6.4 Une seule instance : le Singleton (anti-pattern ?)

Si l'on veut assurer qu'il n'y a qu'une seule instance d'une classe donnée, par exemple un seul objet qui gérera les connections à la base de données, on doit interdire l'accès direct aux constructeurs et imposer l'utilisation d'une méthode (de classe) pour récupérer une référence vers la seule instance qui sera crée.

6.4.1 Exercice

Soit une classe quelconque `A`, implémentez, avec une méthode statique `create()`, le *design (anti-)pattern Singleton* :

```
1  public class A{
2
3  }
```

Quels snt les problèmes qui peuvent êtres rencontrés ? (Se souvenir des spécificités d'implémentation de serveurs par exemple). Veut-on vraiment être absolument limité à une seule instance ? (Penser aux tests !)

6.5 Agir sur un ensemble de valeurs : Visiteur

Lorsqu'un objet 'contient' (y compris implicitement), un grand nombre de valeurs et qu'on veut faire une opération (éventuellement composée) sur un (sous-)ensemble de ces valeurs, la solution la plus facile est de faire une méthode qui retourne cet ensemble, par exemple sous forme d'un tableau ou d'une Collection.

Ce n'est pas forcément désirable si la construction de cette collection est trop coûteuse, en temps ou surtout en mémoire.

```
1  public class IntegerRange{
2      int b;
3      int e;
4      public IntegerRange(int b, int e){
5          // TODO : si b > e, les échanger !
6          this.b= b;
7          this.e= e;
8      }
9      public Integer[] data(){
10         Integer[] res= new Integer[e-b];
11         for(int i=0; i != res.length; ++i){
12             res[i]= b+i;
13         }
14         return res;
15     }
16 }
```

```
1  public class Sum {
2      public static void main(String[] args){
3          IntegerRange ir= new IntegerRange(0, 1000000);
4          long res=0;
5          for(Integer i : ir.data()){
6              res+= i;
7          }
8          System.out.println(res);
9      }
10 }
```

On remarque que la création de l'ensemble de valeurs pourrait être évitée si l'on pouvait passer l'opération à effectuer à l'objet qui contient les valeurs. C'est le principe du *design pattern Visiteur*.

```java
public interface Visitable<Data>{
    public void accept(Visitor<Data> v);
}
```

```java
public interface Visitor<Data>{
    public void visit(Data v);
}
```

```java
public class IntegerRange implements Visitable<Integer>{
    int b;
    int e;
    public IntegerRange(int b, int e){
        // TODO : si b > e, les échanger !
        this.b= b;
        this.e= e;
    }
    public void accept(Visitor<Integer> v){
        for(int i=b; i != e; ++i){
            v.visit(i);
        }
    }
}
```

```java
public class Sum implements Visitor<Integer> {
    private long result=0;
    public void visit(Integer i){
        result+= i;
    }
    public long getResult(){
        return result;
    }
    public static void main(String[] args){
        IntegerRange ir= new IntegerRange(0, 1000000);
        Sum v= new Sum();
        ir.accept(v);
        System.out.println(v.getResult());
    }
}
```

6.5.1 Exercices

Implémenter une classe qui calcule le produit des valeurs de l'intervalle d'une instance de `IntegerRange`. Quelle(s) modification(s) pourriez-vous apporter pour terminer l'opération avant d'avoir traiter tous les éléments, par exemple dès que le produit est nul ?

Si l'on veut aussi pouvoir récupérer explicitement l'ensemble des valeurs, par exemple dans une liste, doit-on modifier la classe `IntegerRange` ?

6.6 Réaction à des modifications : Observer

Parfois, l'on veut pouvoir réagir à une modification de l'état d'un objet. Avec le *design pattern Observer*, un objet implémentant l'interface java.util.Observer peut s'enregistrer (et se désenregistrer) auprès d'un objet implémentant l'interface java.util.Observable. L'observer pourra alors être notifié par l'observable lors d'un changement d'état.

6.6.1 Exercice

Reprendre l'exercice du *design pattern Strategy* et ajouter la possibilité de notifier les changement de statut (d'identification) en faisant de `User` un Observable. Implémenter un java.util.Observer qui écrive une message sur la sortie d'erreur lors de chaque changement d'état d'identification du user.

6.7 Injection de dépendance

Les *design patterns* peuvent se composer, ainsi l'injection de dépendance, vue avec @Inject repose sur une sorte de *Factory* qui permet de construire l'instance à injecter. On ajoute encore un niveau d'indirection pour mettre les paramètres de la construction dans un fichier de configuration XML et/ou dans des annotations.

7 Design patterns ?

Il y a beaucoup de design patterns , dont certains sont tombés en désuétude (comme le Singleton) alors que de nouvelles pratiques apparaissent (le texte canonique date de 1994!).

7.1 Limitation d'effets de bords

Sur l'exemple de la programmation fonctionnelle, on cherche à limiter les *effets de bords* qui introduisent des dépendances cachées entre tous les codes affectés. On préfère autant que possible des fonctions *pures* dont le résultat ne dépend que des arguments et n'a pas d'autre effet que de calculer une résultat.

7.2 Programmation Orientée Valeurs (immutables)

Le meilleurs moyen de s'assurer que des objets passés en argument ne sont pas modifiés, c'est d'interdire leur modification. On peut suivre l'exemple de java.lang.String, java.lang.Integer, etc.

7.3 Programmation Orientée Mathématiques ?

De même que les fonctions *pures* suivent l'exemple des fonctions mathématiques, on a aussi intérêt à s'inspirer d'autres propriétés mathématiques lorsque l'on implémente des opérations, par exemple :

- l'existence d'un élément neutre qu'on puisse "ajouter" (comme le 0 pour l'addition, la chaîne de vide pour la concaténation) sans modifier le résultat afin d'avoir une valeur initiale.
- l'associativité pour permettre les traitement en parallèle
- la commutativité pour permettre les traitements dans le désordre

Ces derniers points sont évidemment particulièrement intéressant lorsqu'on implémente des serveurs.