R3.04 Conception cours n°4

I.Borne

- Principes fondamentaux de la conception par objets
- Introduction aux patrons de conception
  - création : Singleton
  - structurel : Composite

Les principes fondamentaux de la conception par objets

- Les symptômes d'intolérance au changement
- Les 5 principes SOLID
- Héritage de spécification et héritage d'implémentation
- Délégation
- Délégation et héritage dans les patrons de conception

Les symptômes d'intolérance au changement

- Rigidité
  - logiciel difficile à changer parce que chaque changement oblige beaucoup d'autres changements dans d'autres parties du système
- Fragilité
  - les changements entraînent la rupture du système dans des endroits qui n'ont aucun lien conceptuel avec la pièce qui a été modifiée
- Immobilité
  - Il est difficile de démêler le système en composants utilisables dans d'autres pièces
- Viscosité
  - Faire bien les choses est plus difficile que faire mal les choses
- Opacité
  - Il est difficile de lire et comprendre. Il n'exprime pas bien l'intention.

Les 5 principes SOLID

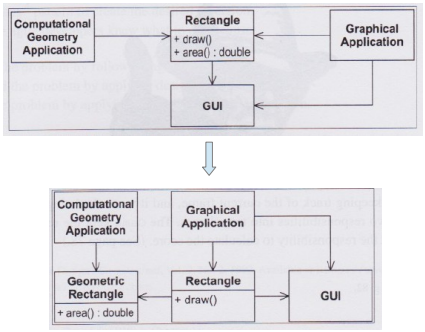
Dans le livre Agile Software Development, Pinciples, Patterns and Practices, Robert C. Martin a condensé, en 2002, cinq principes fondamentaux de conception, répondant à cette problématique d'évolutivité, sous l'acronyme SOLID :

- Single responsibility principle
- Open close principle
- Liskov principle
- Interface segregation principle
- Dependency inversion principle

SRP: the Single responsibility principle

« A class should have one reason to change »

Deux applications qui utilisent une classe Rectangle avec des objectifs différents : la classe Rectangle viole le principe



Responsabilités séparées

OCP: the Open-closed responsibility principle

« Classes, methods should be open for extension, but closed for modifications»

Open for extension : le comportement de l'entité peut être étendu

Closed for modification : l'extension du comportement n'entraîne pas de modification du code source

LSP : Liskov Substitution Principle

« Subtypes must be substituable for their base types. »

Principe :

- Si un objet de type S peut être substitué partout où un objet de type T est attendu, alors S est un sous-type de T.

Interprétation :

- Si toutes les classes sont des sous-types de leurs superclasses, toutes les relations d'héritage sont des relations d'héritage de spécification.

Une relation d'héritage qui satisfait le principe de substitution de Liskov est appelé ***héritage strict.***

ISP : The Interface-Segregation principle

« Clients should not be forced to depend on methods that they do not use. »

Le but de ce principe est d'utiliser les interfaces pour définir des contrats, des ensembles de fonctionnalités répondant à un besoin fonctionnel, plutôt que de se contenter d'apporter de l'abstraction à nos classes. Il en découle une réduction du couplage, les clients dépendant uniquement des services qu'ils utilisent

DIP : The Dependency-Inversion Principle

« High level modules should not depend on low level modules. Both should depend on abstractions. »

« Abstractions should not depend on details. Details should depend on abstractions. »

Pour quelle raison ?

prenons la définition à l'envers : les modules de haut niveau dépendent de modules de bas niveau. En règle générale les modules de haut niveau contiennent le coeur – business – des applications. Lorsque ces modules dépendent de modules de plus bas niveau, les modifications effectuées dans les modules « bas niveau » peuvent avoir des répercussions sur les modules « haut niveau » et les « forcer » à appliquer des changements.

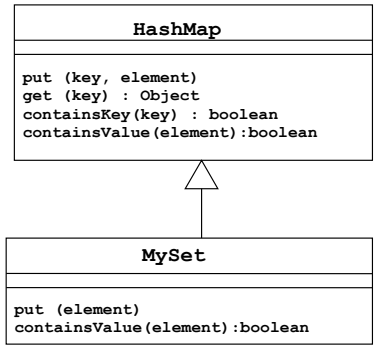
Objets de l'application et objets de la solution

- Pendant la conception objet, les objets de l'application et ceux de la solution sont détaillés et il faut aussi identifier les objets supplémentaires nécessaires.
- Diagrammes de classe UML : modélisation du domaine de l'application et du domaine de la solution.
- Objets de l'application : objets du domaine qui représentent les concepts du domaine pertinents pour le système.
- Objets de la solution : composants qui n'ont pas de contrepartie dans le domaine de l'application, tels que stockage de base données, interface utilisateur, intergiciel.

Héritage de spécification et héritage d'implémentation

- Analyse : classifier des objets en taxinomie.
- Conception : réduire la redondance et promouvoir l'extensibilité.
- Héritage d'implémentation : réutiliser du code.
- Héritage de spécification : classification des concepts en hiérarchie de types (héritage d'interface).

Exemple d'héritage d'implémentation




```
class MySet extends HashMap {

 void put (Object element) {
   if (!this.containsKey(element)) {
        this.put(element, this);
   }
  }

 boolean containsValue(Object element){
   return this.containsKey(element);
  }
}
```

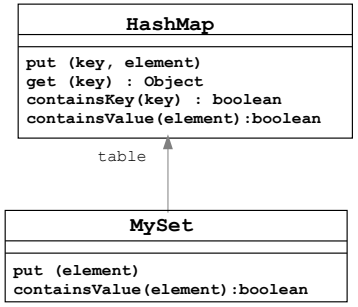
Explications

- L'objectif de l'exemple est de concevoir une nouvelle class MySet.
- La solution choisie a été de réutiliser la classe HashMap par héritage.
- Il s'agit d'un héritage d'implémentation qui ici nous oblige à ré-implémenter certaines méthodes.

La délégation

- La délégation est une alternative à l'héritage d'implémentation.
- Une classe est dite « déléguer à une autre classe » si elle implémente une opération en renvoyant un message à une autre classe.
- La délégation rend explicite les dépendances entre la classe réutilisée et la nouvelle classe.
- La délégation est préférable à l'héritage d'implémentation.
- L'héritage de spécification est préférable à la délégation dans des situations de sous-typage et conduit à une conception plus extensible.

Exemple avec une délégation




```
class MySet  {
  private HashMap table;

  MySet() {
         table = new HashMap();
  }
  void put (Object element) {
   if (!this.containsValue(element)) {
         table.put(element, this);
   }
  }

  boolean containsValue (Object element) {
   return table.containsKey(element);
  }
}
```


Délégation et héritage dans les patrons de conception

- Quand utiliser l'héritage ou la délégation ?
- Différentes combinaisons de délégation et d'héritage pour :
  - découpler des interfaces abstraites de leur implémentation,
  - découpler les classes qui fournissent une politique de celles qui fournissent des mécanismes,
  - envelopper du code ancien (*legacy code*).
- En développement par objets, les patrons de conception sont des solutions de schémas que les développeurs ont éprouvés et qui résolvent des problèmes récurrents.

Retour sur l'exemple

- Le problème est d'écrire une classe `Set` comme implémentant une nouvelle classe qui satisfait une interface existante (interface Java pour `Set`) et en réutilisant le comportement fourni par une classe existante (`HashMap`).
- L'interface de `Set` et la classe `HashMap` sont fournies et ne peuvent pas être modifiées.
- Le patron de conception ***Adapter*** est une solution à ce problème.

Introduction aux patrons de conceptions

Patron ou pattern ?

- Un patron (ou pattern en anglais) constitue une base de savoir-faire pour résoudre un problème récurrent dans un domaine particulier.
- L'expression de ce savoir-faire :
  - permet d'identifier le problème à résoudre
  - propose une solution possible et correcte pour y répondre
  - offre les moyens d'adapter cette solution

Définition

- Un patron décrit à la fois un **problème** qui se produit très fréquemment dans votre environnement et l'architecture de la **solution** à ce problème de telle façon que vous puissiez **utiliser** cette solution des milliers de fois sans jamais l'**adapter** deux fois de la même manière.

C. Alexander

Décrire avec succès des types de solutions récurrentes
à des problèmes communs dans des types de situations

Documentation d'un patron

La forme générale de documentation des patrons est de définir des items tels que :

- La motivation ou contexte dans lequel s'applique ce patron.
- Les pré-requis qui doivent être satisfaits avant de décider d'utiliser un patron.
- Une description de la structure du programme que le patron définira.
- Une liste des participants nécessaires pour compléter un patron.
- Les conséquences d'utilisation du patron à la fois positives et négatives.
- Exemples

Formalisme (complet)

1- nom du patron et sa **classification**

2- intention : explique ce que fait le patron, sa raison d'être, son objectif

3- alias : les autre noms connus du patron s'ils en existent

4- motivation : un scénario d'application du patron, les problèmes particuliers

5- indications d'utilisation : les situations dans lesquelles ce patron peut être utilisé

6- structure : une représentation graphique du patron utilisant la notation OMT/UML

7- participants : les classes et/ou les objets participants et leurs responsabilités

8- collaborations : comment les participants collaborent

9- conséquences : décrit les résultats d'utilisation du patron

10- implantation : les astuces, les variantes et les conseils d'implantation

11- exemples de code : fragments de code illustrant l'implantation du patron

12- utilisations remarquables : des exemples d'utilisations réelles de ce patron

13- patrons apparentés : les autres patrons qui l'utilisent ou sont utilisés par lui

Les patrons de conception du GoF

purpose →

scope ↓

	Creational	**Structural**	**Behavioral**
class	Factory Method	Adapter (class)	Interpreter Template Method
object	Abstract Factory Responsibility	Adapter (object)	Chain of
	Builder	Bridge	Command
	Prototype	Composite	Iterator
	Singleton	Decorator	Mediator
		Facade	Memento
		Flyweight	Observer
		Proxy	State, Strategy, Visitor

1. Les patterns de création

- Ils permettent d'abstraire le processus d'*instanciation*.
- Ils rendent les applications indépendantes de la façon dont leurs objets sont créés, composés et représentés.
- Ils fournissent les possibilités suivantes :
  - **instanciation générique** : création d'un objet sans avoir à identifier un type de classe spécifique dans le code
  - **simplicité** : évite d'avoir à écrire du code compliqué
  - **contraintes de création** : sur le nombre et le type des objets à créer

1.1 Singleton (création)

Intention

Garantir qu'une classe n'aura qu'une seule instance et fournir un accès global à cette instance unique.

Motivation

Sur un système il ne doit y avoir qu'un seul système de fichier et qu'un seul gestionnaire de fenêtres. Un système de comptabilité est destiné à servir une seule entreprise.

Comment faire ? Une variable globale rend un objet accessible, mais n'empêche pas de définir plusieurs objets.

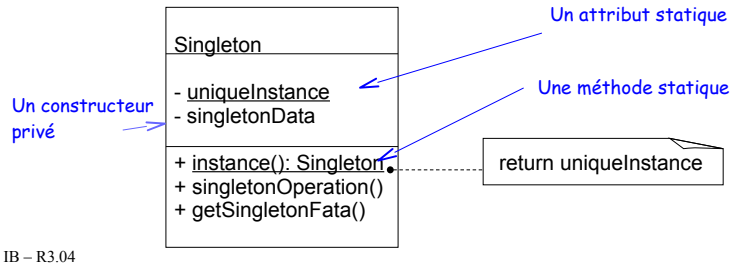
Une meilleure solution est de rendre la classe elle-même responsable de son instance unique. La classe doit garantir qu'aucune autre instance ne peut être créée, en interceptant les requêtes de création, et elle doit fournir une chemin d'accès à son instance.

Quand appliquer Singleton ?

Quand il doit y avoir exactement une instance d'une classe et qu'elle doit être accessible par les clients de la classe grâce à un point d'accès connu.

Quand l'instance unique doit être « étendue » dans des sous-classes, les clients doivent pouvoir l'utiliser sans avoir à modifier leur code.

Structure



Participants

Singleton

- Définit une méthode `instance` qui permet aux clients d'accéder à l'instance unique.
- `instance` est une méthode de classe (statique en Java).
- Peut être responsable de la création de son instance unique.

Collaborations

Les clients accèdent à l'instance de Singleton uniquement à l'aide de la méthode de classe `instance` de la classe Singleton.

Singleton : Conséquences

Les bénéfices du pattern Singleton :

1. Accès contrôlé à l'instance unique, parce que Singleton encapsule son instance.

2. Réduction de l'espace de nommage. Le pattern Singleton est une amélioration de l'utilisation de variables globales.
   Il évite de polluer l'espace des noms avec des variables globales qui stockent des instances uniques.

3. Autorise le raffinement des méthodes et de la représentation. La classe Singleton peut être sous-classée, et il est facile de configurer une application avec une instance d'une classe étendue. Vous pouvez configurer l'application avec une instance de la classe dont vous avez besoin à l'exécution.

Singleton : **Conseils utiles**

Implémentation

Deux cas :

1. *Garantir une instance unique* .
2. *Sous-classer la classe Singleton.*

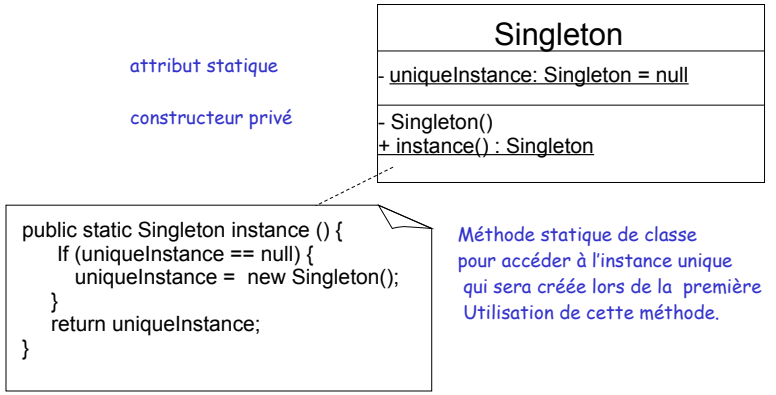
Exemple de code

Usages connus

Patterns liés

Beaucoup de patterns peuvent être implémentés en utilisant le pattern Singleton. Voir Abstract Factory (87), Builder (97), et Prototype (117).

Singleton (implémentation 1)



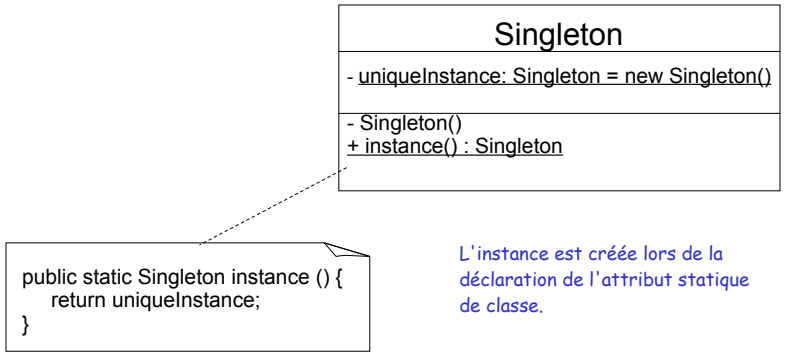
Exemple de la classe **`HistoryList`** (liste d'historique unique

```
import java.util.*;
public class HistoryList {
   private List history =
               Collections.synchronizedList(new ArrayList());
   private static HistoryList uniqueInstance;

   private HistoryList();

   public static HistoryList instance(){
        if( uniqueInstance == null){
             uniqueInstance = new HistoryList();
        }
        return uniqueInstance;
   }
```

Variable de classe

Constructeur privé

Singleton (implémentation 2)



```
import java.util.*;
public class HistoryList {
   private List history =
               Collections.synchronizedList(new ArrayList());

   private static HistoryList uniqueInstance = new HistoryList();

   private HistoryList();

   public static HistoryList instance(){
        return uniqueInstance;
   }
```

Variable de classe initialisée

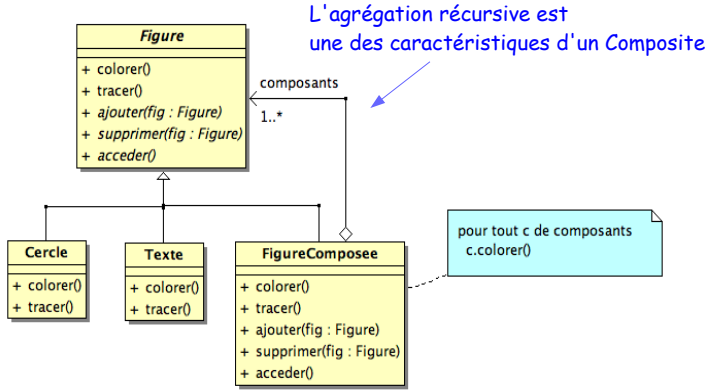
2. Les patterns structurels

- L'objectif des patterns structurels est de proposer des solutions pour composer des classes et des objets afin d'obtenir des structures plus complexes.
- On distingue :
  - les patterns structurels de classe qui utilisent l'héritage pour composer des interfaces ou des implémentations.
  - les patterns structurels d'objets décrivent comment composer des objets pour réaliser de nouvelles fonctionnalités.

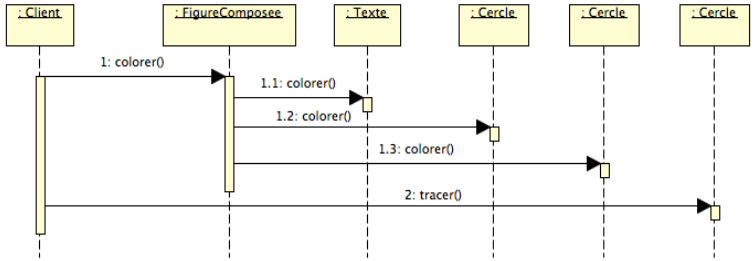
2.1 Le pattern Composite

- **Classification** : structure - Objet
- **Nom** : Composite
- **Intention** : décrire des compositions récursives et permettre à des clients de traiter des objets individuels ou des compositions d'objets uniformément.
- **Motivation** : dans les applications graphiques, par exemple les éditeurs graphiques, le système autorise l'élaboration de figures composites à partir de figures simples et prédéfinies, mais aussi à partir de figures composites élaborées précédemment.

Exemple de Composite

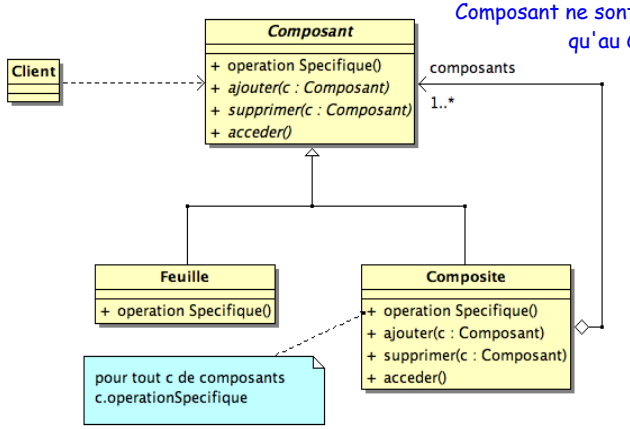


Fonctionnement du Composite

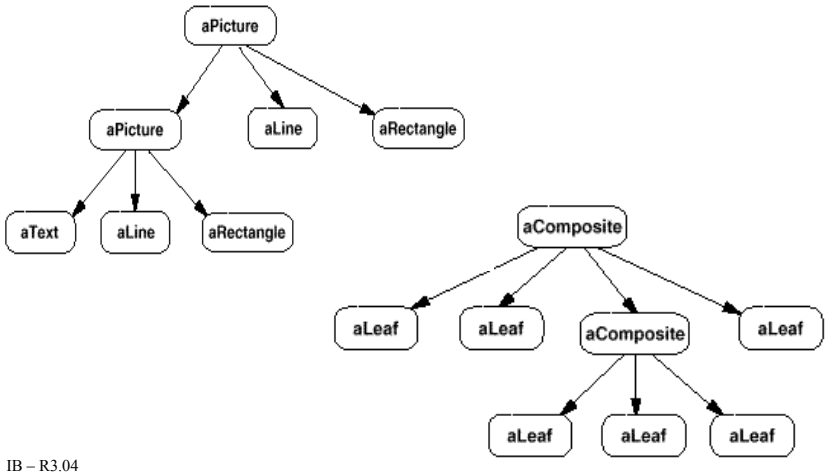


Solution du Composite (structure)

Les méthodes abstraites de Composant ne sont destinées qu'au Composite



Structure typique des objets composés récursivement d'objets graphiques composés



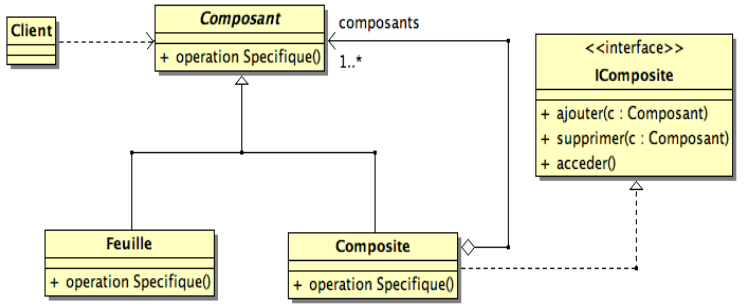
Exemple de code

```
public abstract class Component {
    public abstract void add(Component c) {}
    public abstract void remove(Component c) {}
    public abstract void operation();
  }

 public class Leaf extends Component {
    public void operation() { /* do something */ }
  }
```

```
import java.util.*;
```

```
public class Composite {
    private List<Component> children = new LinkedList();

    public void add(Component c) {children.add(c);}
    public void remove(Component c) {children.remove(c);}

    public void operation() {
      for (Component item : children) {
            item.operation();
      }
    }
}
```


Composite avec une interface



Qu'avez-vous retenu ?

- Que signifie le sigle SOLID ?
- Différence entre héritage de spécification et héritage d'implémentation
- Que permet le pattern Singleton ?
- Que permet le pattern Composite ?