Objet et développement d'applications
Cours 9 - Patterns en vrac

Patrons de conception pour POO

Florian Richoux

2014-2015

Quelques patterns restants

Patterns de comportement

- Command
- Chain of Responsability
- Visitor

Patterns de structure

- Composite
- Bridge
- Proxy

Patterns de création

- Builder
- Object Pool

Patterns de comportement

- Command
- Chain of Responsability
- Visitor

Pattern Command

Définition

Le pattern Command (Commande) encapsule une requête comme un objet, autorisant ainsi le paramétrage des clients par différentes requêtes, files d'attente et récapitulatifs de requêtes. De plus, il permet la réversion des opérations (opération "undo").

L'idée

Command vous permet de "black-boxer" des appels de méthodes : les classes clientes donneront des ordres, mais sans même connaître le nom des méthodes à appeler.

L'intuition

Intuitivement, c'est un peu comme le pattern Façade mais avec une couche supplémentaire d'abstraction permettant en quelque sorte de faire de la *composition de façades*.

Command : exemple de la télécommande

Télécommande universelle

Considérez une télécommande universelle : avec le même bouton, vous pourrez allumer la télé, la chaîne hifi, et le four pour la cuisson d'un canard.

Command : exemple de la télécommande

En Java

```
class Tele {
  public void allumer() { ... }
}

interface Command {
  public void executer();
}

class CommandTele implements Command {
  private Tele tele_;

  public CommandTele() {
    tele_ = new Tele();
  }

  public void executer() {
    tele_.allumer();
  }
}
```

Command : exemple de la télécommande

En Java

```
class Telecommande {
  private Command slot_;

  public void setCommand(Command slot) {
    slot_ = slot;
  }

  public void bouttonOn() {
    slot_.executer(); // on ne connaît pas
                      // la méthode allumer() !
  }
}
```

Command : exemple de la télécommande

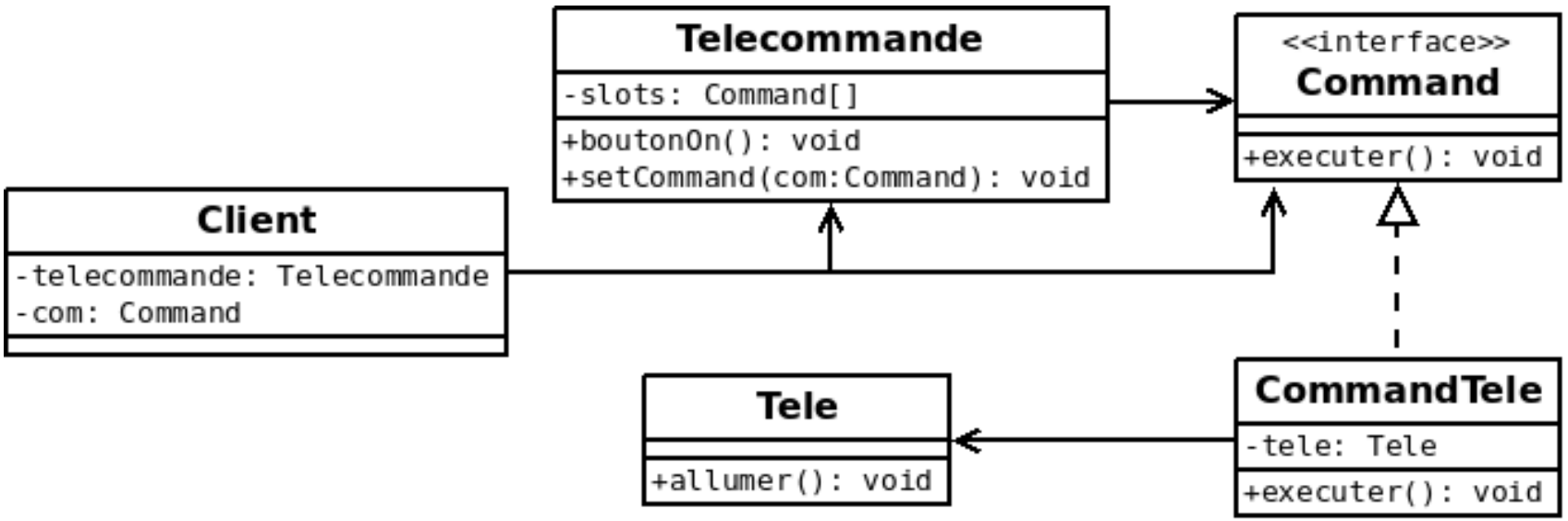
En Java

```
class Client {
  private Telecommande tc_ = new Telecommande();

  // la télécommande va contrôler la télé
  private Command com_ = new CommandTele();
  tc_.setCommand(com_);

  tc_.bouttonOn(); // allume la télé

  // la télécommande va maintenant contrôler la chaîne
  com_ = new CommandHifi();
  tc_.setCommand(com_);

  tc_.bouttonOn(); // allume la chaîne hifi
```

Command : remarques et UML

Remarques importantes

- Si `Telecommande` contient un tableau de slots, elle peut gérer plusieurs classes concrètes `Command` en même temps.
- On peut avoir un bouton `undo` pour annuler la dernière action. (Réfléchissez à son implémentation)



Command : remarques

Remarques hyper-importantes

- Découplage de la création d'un ordre et de son exécution !
- On peut créer des ordres (encapsulés par des objets), les mettre dans une file d'attente et exécuter plus tard ces ordres en piochant dans la file. Le tout peut être multi-threadé.
- Undo à la chaîne. Backup des ordres en cas de crash, et restauration des ordres.

Pattern Chain of Responsability

Définition

Le pattern Chain of Responsability (Chaîne de responsabilités) évite le couplage de l'émetteur d'une requête à ses récepteurs, en donnant à plus d'un objet la possibilité d'entreprendre la requête. Chaîner les objets récepteurs et faire passer la requête tout au long de la chaîne, jusqu'à ce qu'un objet la traite.

L'idée

Vous avez une requête à traiter (par exemple, un email). L'action à faire est différente suivant la nature de la requête. Vous passez la requête à une chaîne d'objets, chacun spécialisé pour un type d'action. La requête traversera la chaîne jusqu'à tomber sur l'objet qui va la traiter.

Chain of Responsability : exemple de l'email

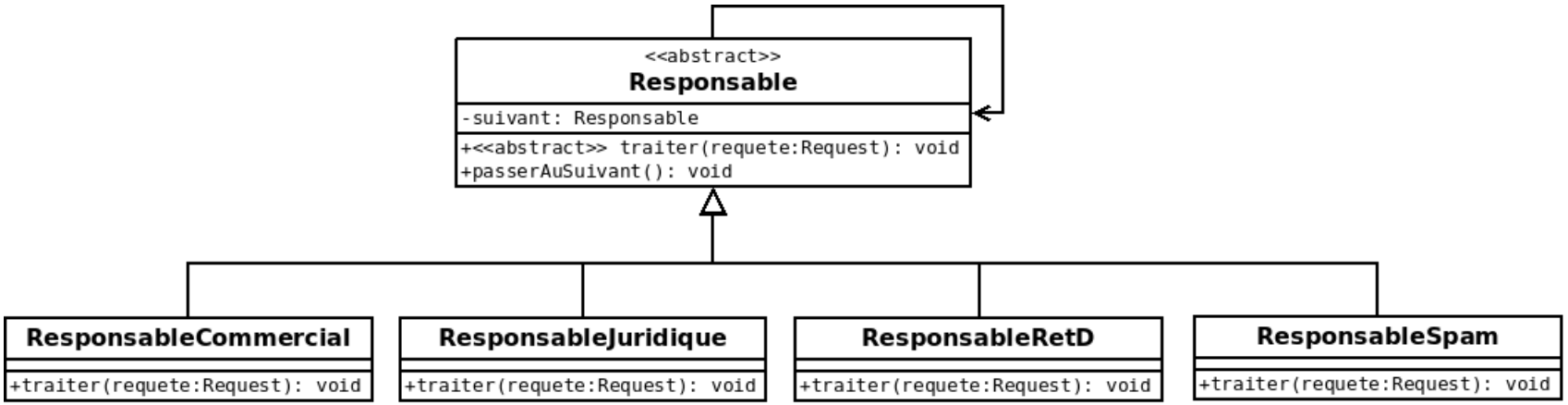
Entreprise recevant un email

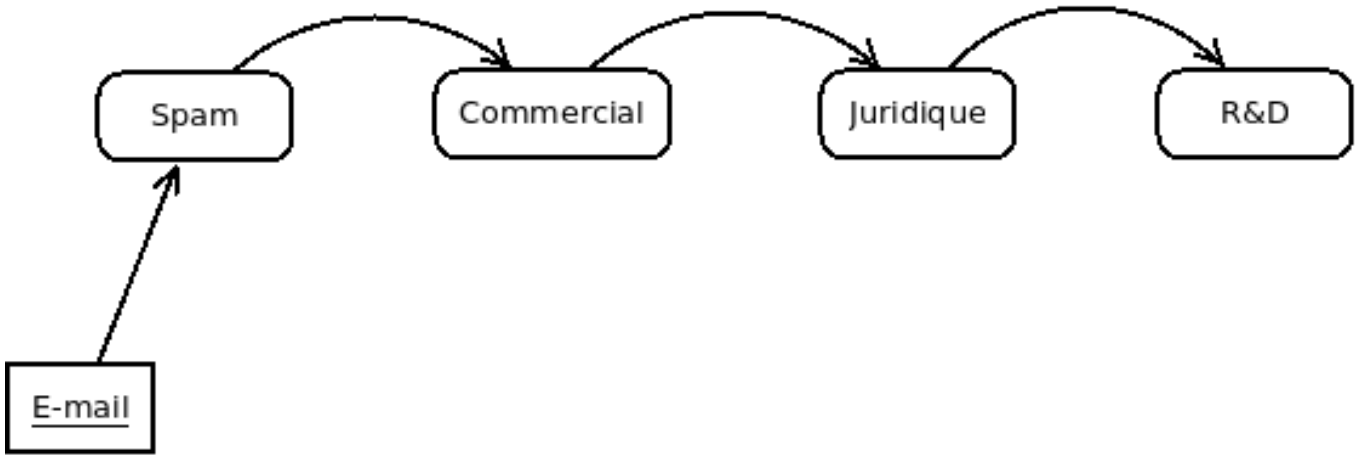
Vous travaillez dans une entreprise recevant des emails de différentes natures : des commandes, des plaintes, des idées d'améliorations et du spam.

Traitement des emails

Vous avez une IA qui peut repérer la nature des emails. Vous devez transmettre les commandes au service commercial, les plaintes au service juridique, les idées à la R&D et mettre les spams à la corbeille.

Chain of Responsability : UML et schéma





Chain of Responsability : remarques

Les plus

- Découple l'émission de la requête de son traitement (comme Command).
- Simplifie les requêtes car elles ne doivent pas connaître la structure de la chaîne.
- Ajouts et suppressions de responsabilités dynamiques.

Utilisation et inconvénient

- Souvent utilisé dans les GUI pour gérer les évènements souris / clavier.
- Peut rendre difficile la lisibilité de l'exécution, et donc du débuggage.

Pattern Visitor

Définition

Le pattern Visitor (Visiteur) fait la représentation d'une opération applicable aux éléments d'une structure d'objet. Il permet de définir une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit.

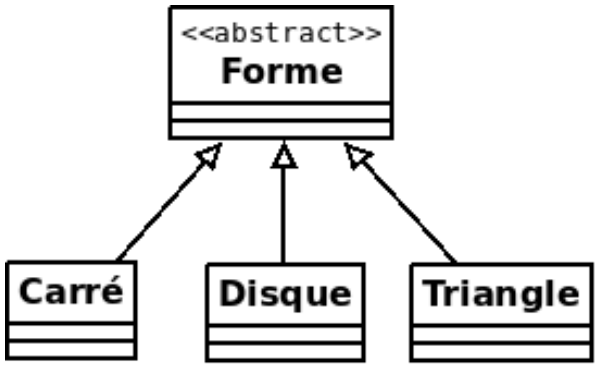
Contexte

Vous avez une collection d'objets sur lesquels vous souhaitez appliquer des actions semblables qui diffèrent légèrement selon l'objet. (comme le calcul du périmètre d'une forme). Ça vous embête d'écrire quasiment les mêmes méthodes dans chaque objet.

L'idée

Vous faites un Visitor qui peut visiter chaque objet. Vous n'avez alors qu'à écrire les opérations qu'au niveau du Visitor uniquement.

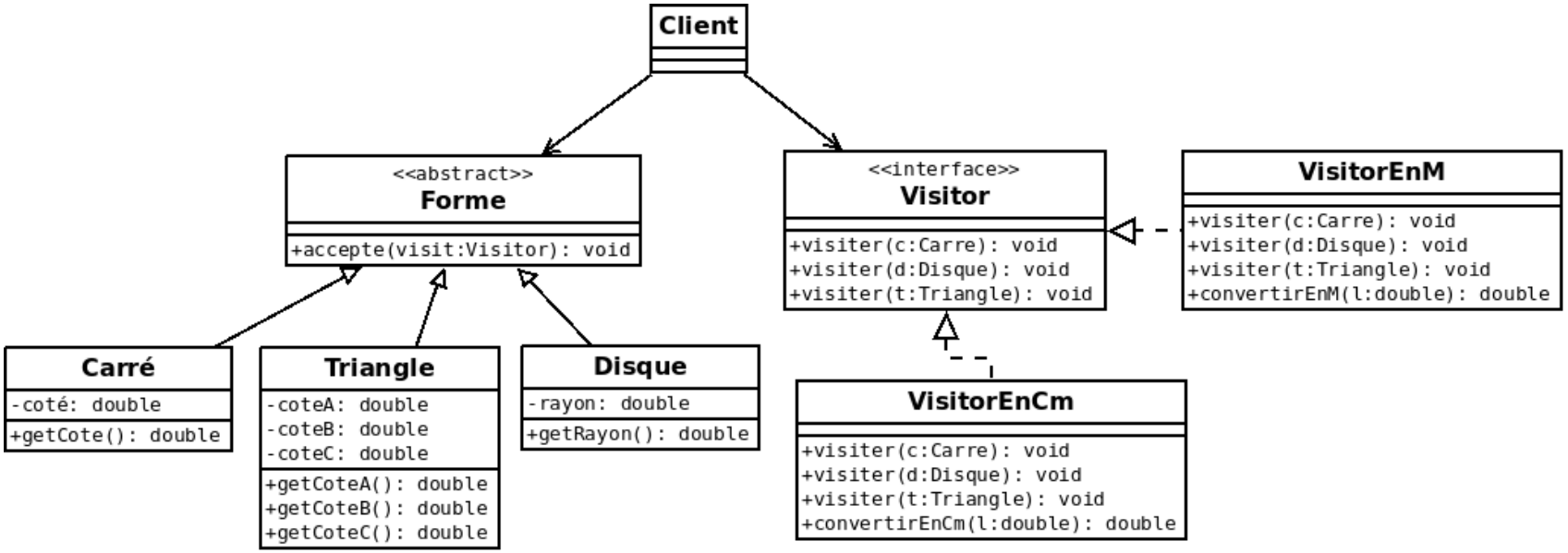
Visitor : exemple d'affichage de formes



Code pas maintenable

```
public double perimetre(Forme f) {
  if( f instanceof Carre)
    return perimetreCarre( (Carre) f );
  else if( f instanceof Disque)
    return perimetreDisque( (Disque) f );
  else if( f instanceof Triangle)
    return perimetreTriangle( (Triangle) f );
  else
    throw new RuntimeException("Forme inconnue");
}
```

Visitor : UML


Client
<<abstract>>
Forme
+accepte(visit:Visitor): void
<<interface>>
Visitor
+visiter(c:Carre): void
+visiter(d:Disque): void
+visiter(t:Triangle): void
VisitorEnM
+visiter(c:Carre): void
+visiter(d:Disque): void
+visiter(t:Triangle): void
+convertirEnM(l:double): double
Carré
-coté: double
+getCote(): double
Triangle
-coteA: double
-coteB: double
-coteC: double
+getCoteA(): double
+getCoteB(): double
+getCoteC(): double
Disque
-rayon: double
+getRayon(): double
VisitorEnCm
+visiter(c:Carre): void
+visiter(d:Disque): void
+visiter(t:Triangle): void
+convertirEnCm(l:double): double

Visitor : code

Morceau de code

```
public void accepte(Visitor visitor) {
  visitor.visiter( this );
}

class VisitorEnCm implements Visitor {
  public void visiter(Carre c) {
    print( "Perimètre du carre en cm : " +
           convertirEnCm( c.getCote() * 4 ) );
  }
  ...
}

public static void main(String[] args) {
  Visitor v = new VisitorEnCm();
  Forme[] list =
    { new Carre(), new Disque(), new Triangle() };
  for( int i = 0; i < list.length; ++i )
    list[i].accepte(v);
}
```

Patterns de structure

- Composite
- Bridge
- Proxy

Pattern Composite

Définition

Le pattern Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même et unique façon les objets individuels et les combinaisons de ceux-ci.

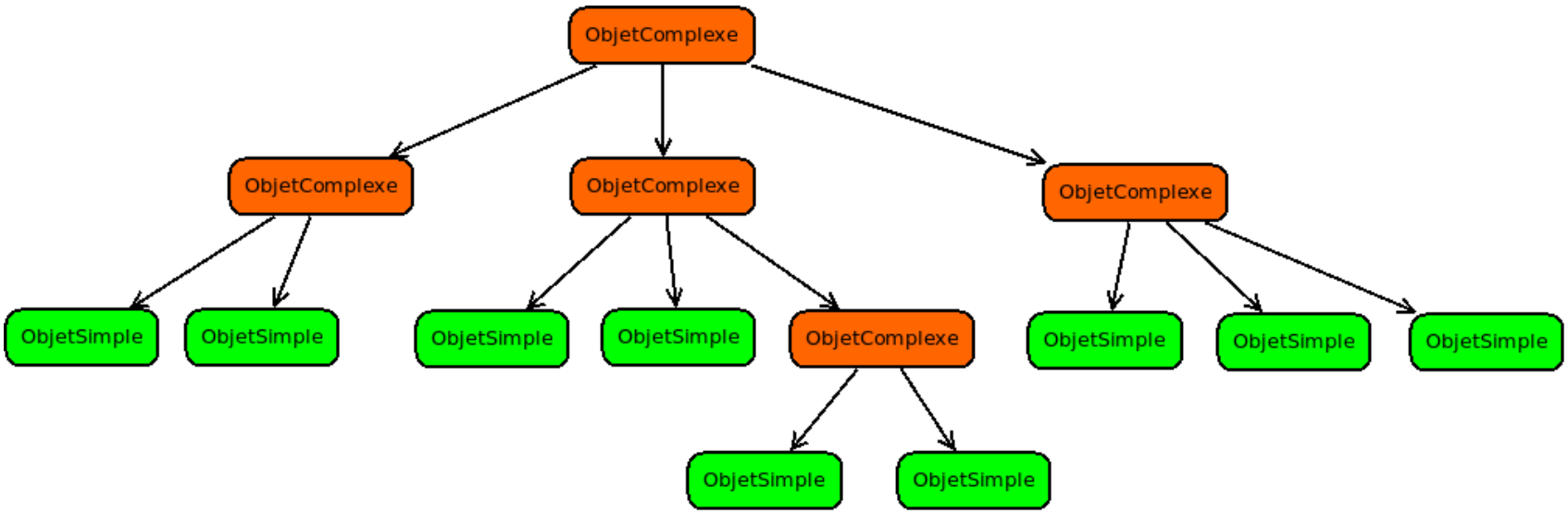
L'idée

Utiliser Composite peut vous aider à créer une structure d'arbre où les noeuds et les feuilles partagent le même type. Les noeuds et les feuilles pourront donc être traités de la même manière, ayant la même signature.

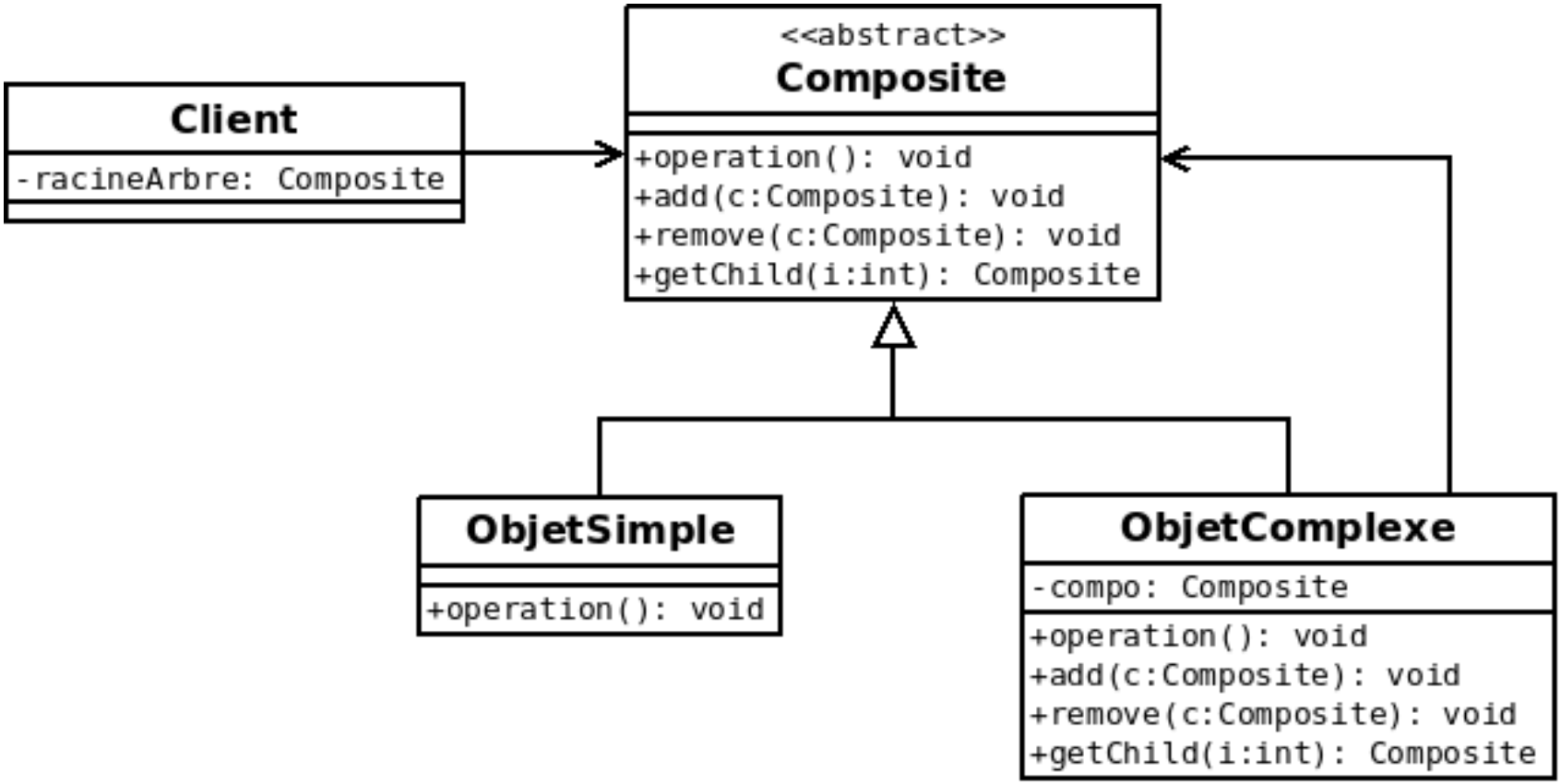
Composite : principe

Principe

Vous souhaitez manipuler un `ObjetComplexe` qui contient (dans un `ArrayList` par exemple) une collection d'`ObjetSimple`. Mais de temps en temps, il vous est plus pratique que cette `ArrayList` contienne également quelques `ObjetComplexe`.



Composite : UML



Pattern Bridge

Définition

Le pattern Bridge (Pont) découple une abstraction de ses implémentations afin que ces deux parties puissent être modifiés indépendamment l'un de l'autre.

L'idée

Si vous avez une classe abstraite et plusieurs classes en héritant, la modification de la classe abstraite entraîne la modification de ses classes descendantes.

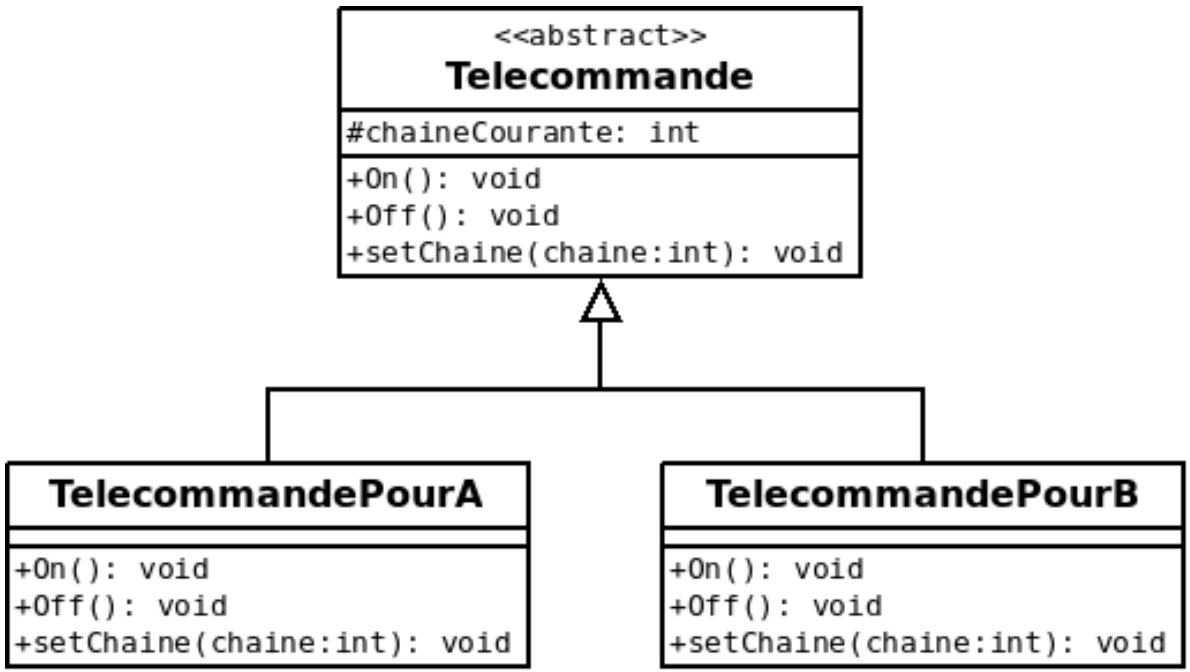
Souplesse

Il y a des cas où l'on souhaiterait plus de souplesse (donc de découplage) entre une abstraction et ses implémentations. Notamment si la structure de départ du programme n'était pas la bonne et qu'on souhaite recadrer les choses à moindre coût.

Bridge : exemple

Télécommande et télés

Imaginez que l'on a écrit une classe abstraite `Telecommande` dont dérivent des télécommandes concrètes pour une télé de marque A et une autre de marque B.



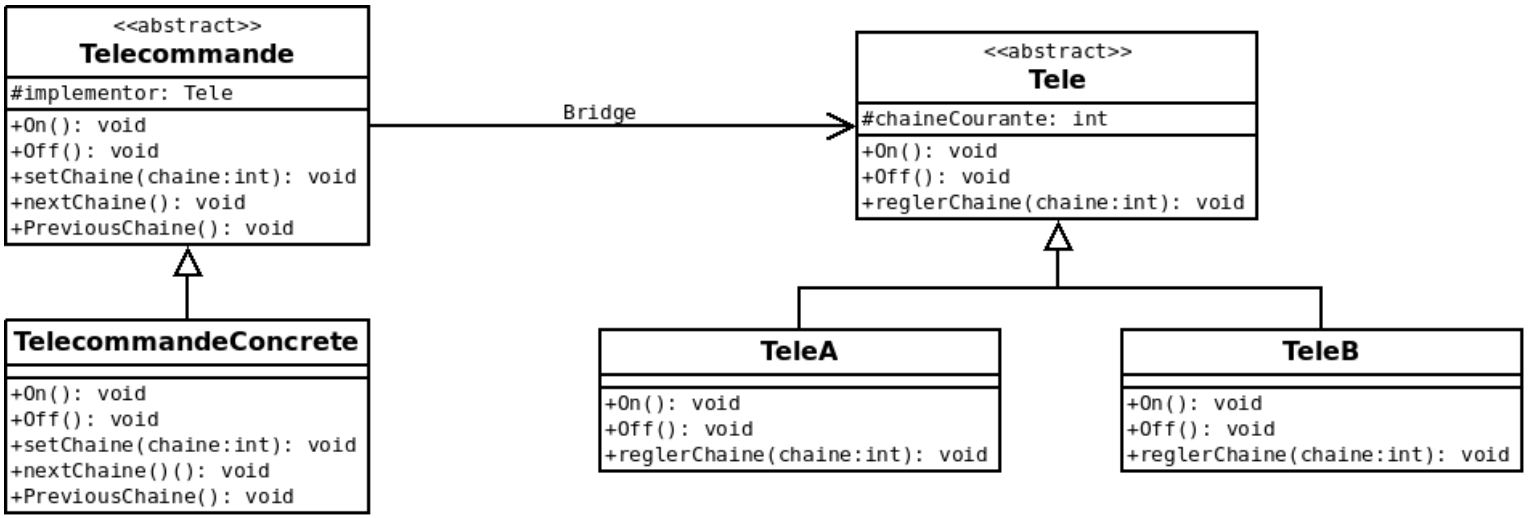
Bridge : exemple

Divergences

On s'aperçoit qu'on a plutôt besoin de télécommandes et de télés séparées. Le code de `Telecommande` et ses descendants diverge.

Bridge

Pas grave : on compose `Telecommande` avec une abstraction `Tele`, créant un "pont" entre les deux structures.



Pattern Proxy

Définition

Le pattern Proxy (Procuration ou Délégation) fournit à un tiers objet un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.

Intuitivement

Un proxy, c'est un leurre : si l'objet A fait appel à un proxy P, tout objet croyant communiquer directement avec A vont en fait communiquer avec P.

Proxy : les types

Types de proxies

Très grande diversité de proxies.

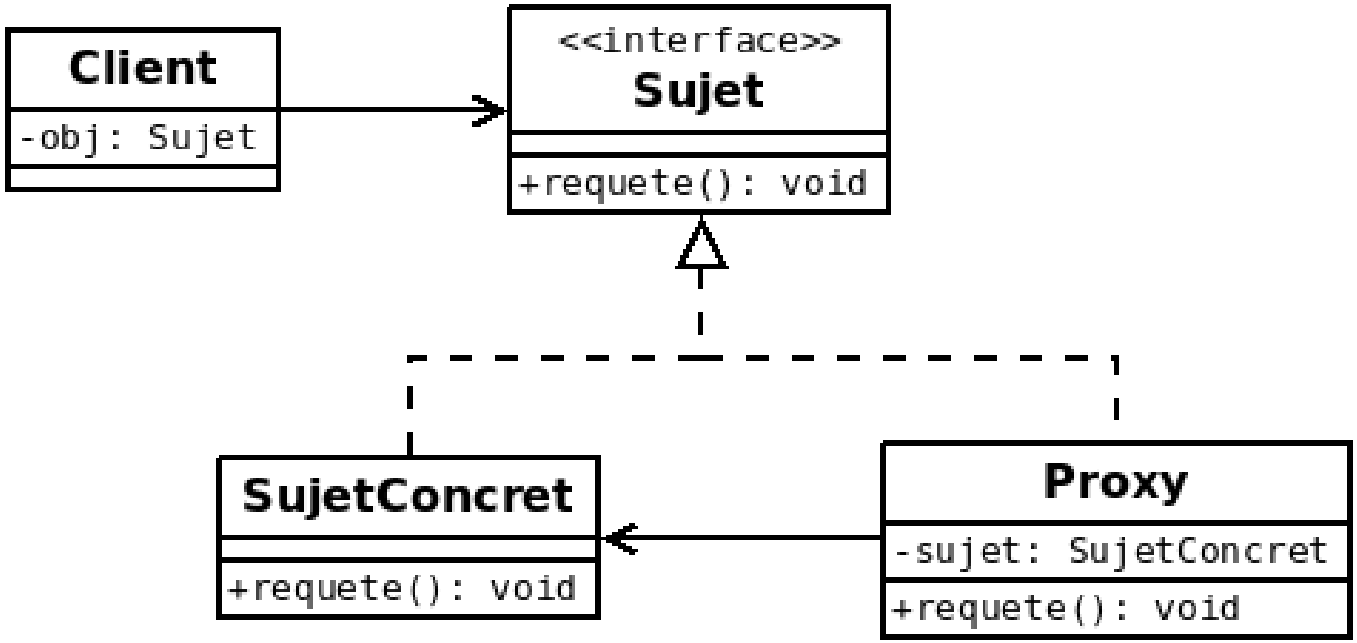
Les trois plus connus

- **Proxy à distance** : gère les interactions entre deux objets en mémoire sur différentes machines.
- **Proxy virtuel** : contrôle l'accès à des objets lourds à charger.
- **Proxy de protection** : garde du corps, filtre l'accès à l'objet protégé.

Mais aussi

- Proxy firewall,
- Proxy "smart reference",
- Proxy de synchronisation,
- ...

Proxy : UML



Proxy à distance dans l'API Java

Allez faire un tour du côté de `java.rmi`.
(RMI = Remote Method Invocation)

Patterns de création

- Builder
- Object Pool

Pattern Builder

Définition

Le pattern Builder (Monteur) dissocie la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction permette des représentations différentes.

Intuitivement

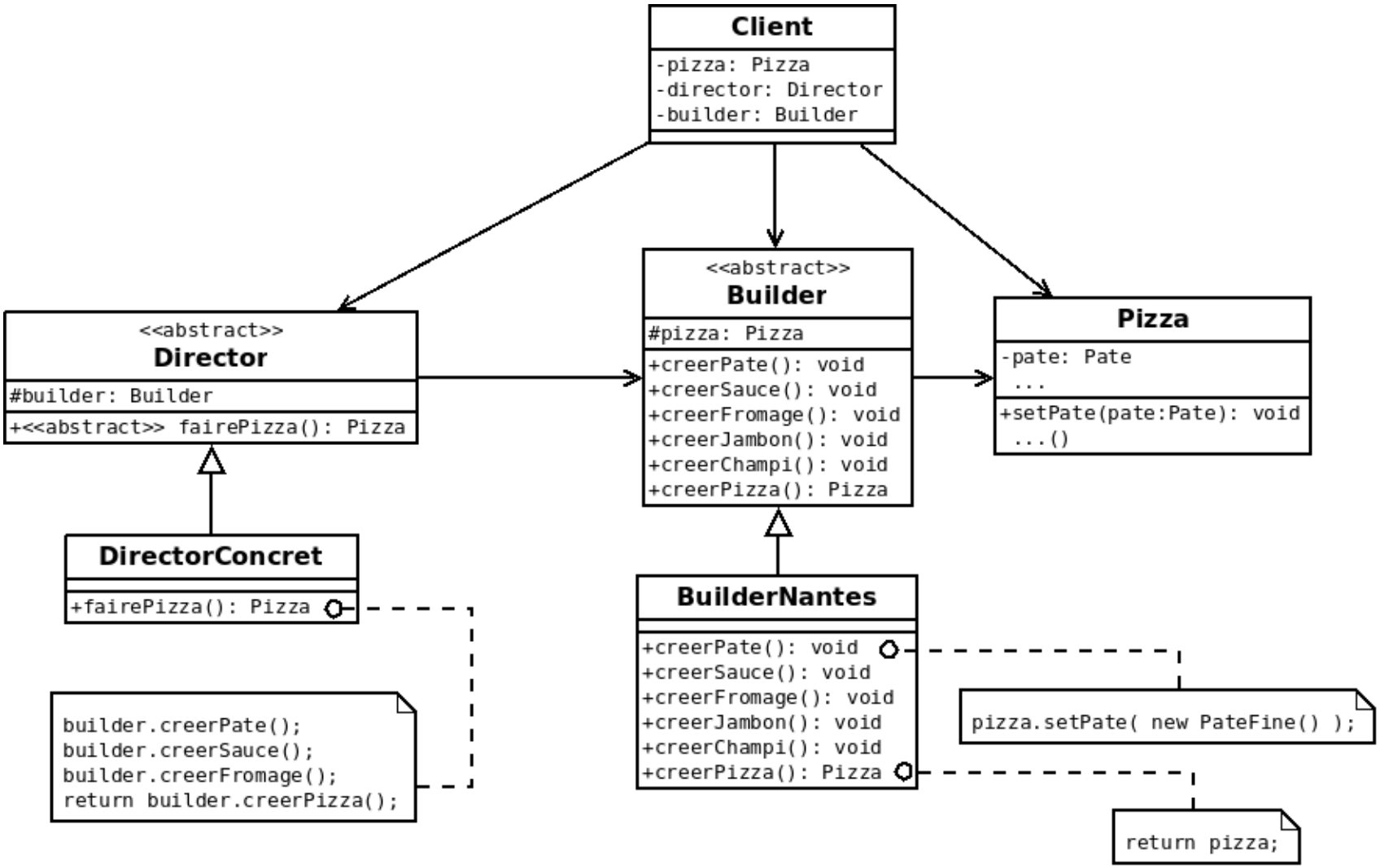
C'est comme une Abstract Factory, sauf que l'on peut choisir à la carte ce que l'on souhaite initialiser dans l'objet à créer.

Builder : les classes

Les classe du Builder

- Le **Builder abstrait** : contient l'interface des Builders, avec une méthode pour créer chaque composant et une méthode `creerObjet` pour créer l'objet comme indiqué par un Director.
- Les **Builders concrets**.
- Le **Director abstrait** : contient une méthode de création qui fera appel aux méthodes du Builder, et finalisera la création par l'appel de `creerObjet` du Builder.
- Les **Directors concrets**.
- Une class **Produit** (l'objet qui sera créé).
- Un **Client** qui jongle avec tout ça.

Builder : UML


Client
-pizza: Pizza
-director: Director
-builder: Builder
<<abstract>>
Director
#builder: Builder
+<<abstract>> fairePizza(): Pizza
<<abstract>>
Builder
#pizza: Pizza
+creerPate(): void
+creerSauce(): void
+creerFromage(): void
+creerJambon(): void
+creerChampi(): void
+creerPizza(): Pizza
Pizza
-pate: Pate
...
+setPate(pate:Pate): void
...()
DirectorConcret
+fairePizza(): Pizza
BuilderNantes
+creerPate(): void
+creerSauce(): void
+creerFromage(): void
+creerJambon(): void
+creerChampi(): void
+creerPizza(): Pizza
builder.creerPate();
builder.creerSauce();
builder.creerFromage();
return builder.creerPizza();
pizza.setPate(new PateFine());
return pizza;

Pattern Object Pool

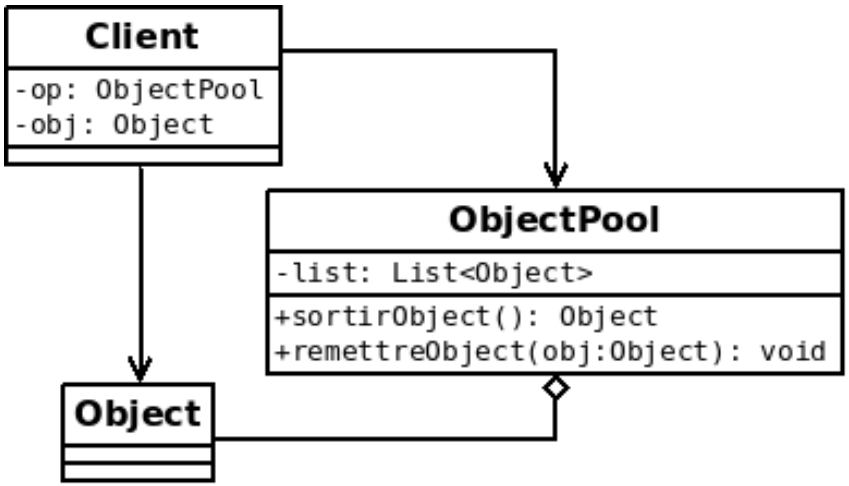
Définition semi-officielle (ce n'est pas un pattern du GoF)

Le pattern Object Pool (Groupe d'objets) met à disposition un ensemble d'objet du même type instanciés au préalable afin d'éviter leur création au moment où l'on a besoin d'un de ces objets.

L'idée

Si votre programme doit instancier des objets lourds, l'Object Pool est un moyen d'instancier un groupe d'objets au même moment et de les manipuler par la suite, en évitant d'en instancier de nouveaux.

Object Pool : UML



Principe

1. Instancier une liste d'objets, par exemple une file.
2. Défiler un objet quand on en a besoin d'un.
3. Le renfiler dans la pool après utilisation.

Object Pool : pour et contre

Pour

- Vos objets sont lourds, mais vous savez que vous n'en utilisez jamais plus de n en même temps : piochez-les dans une Object Pool quand vous en avez besoin et remettez les dedans quand vous en avez fini.
- Votre programme se décompose en deux parties : une partie "lente" et une partie où la rapidité d'exécution est critique. Instanciez la pool dans la première et utilisez vos objets dans la deuxième.

Contre

- Perd son intérêt si vous n'avez pas assez d'objets dans la pool et que vous devez en créer. Cela peut rendre sa gestion plus complexe.
- Prendre le soin de réinitialiser les objets à "vide" avant de les remettre dans la pool, sous peine d'avoir des comportements inattendus.

Oh no ! More patterns !

On ne les a pas vu mais ils existent

- Flyweight,
- Interpreter,
- Iterator,
- Mediator,
- Memento,
- Template Method,
- Null Object,
- Prototype,
- ...

Un dernier mot sur les patterns

Les 4 différents niveaux de maîtrise des patterns

- **Niveau débutant** (votre niveau) : vous avez la patternite. Vous voulez caser des patterns partout, maintenant que vous les connaissez.
- **Niveau intermédiaire** : vous voyez quand un pattern est nécessaire ou pas.
- **Niveau expérimenté** : vous savez quand un pattern s'applique naturellement dans votre programme.
- **Niveau Chuck Norris** : vous développez vos propres patterns.

Anti-patterns

Attention aux anti-patterns !
http://sourcemaking.com/antipatterns