Programmation Avancée et Application

Les Design Pattern

Jean-Guy Mailly

jean-guy.mailly@parisdescartes.fr

LIPADE - Université de Paris (Paris Descartes)
http://www.math-info.univ-paris5.fr/∽jmailly/

Les Design Pattern

1. Introduction
2. Patrons de création
3. Patrons de structure
4. Patrons de comportement

Introduction

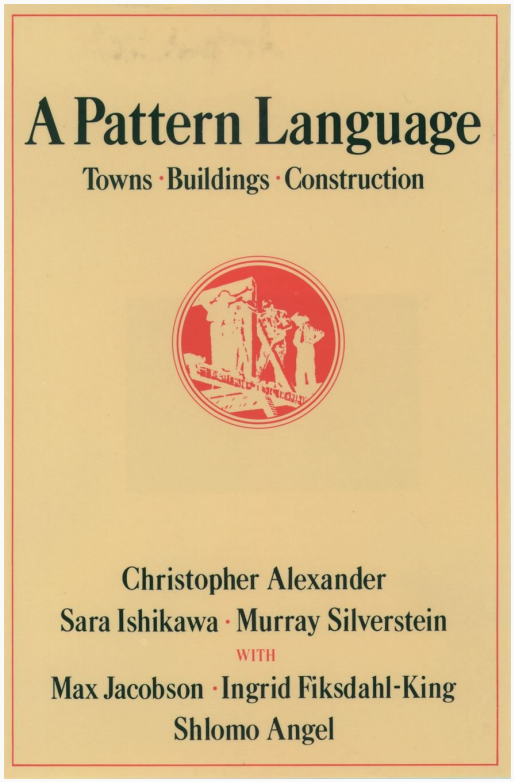
Design Pattern

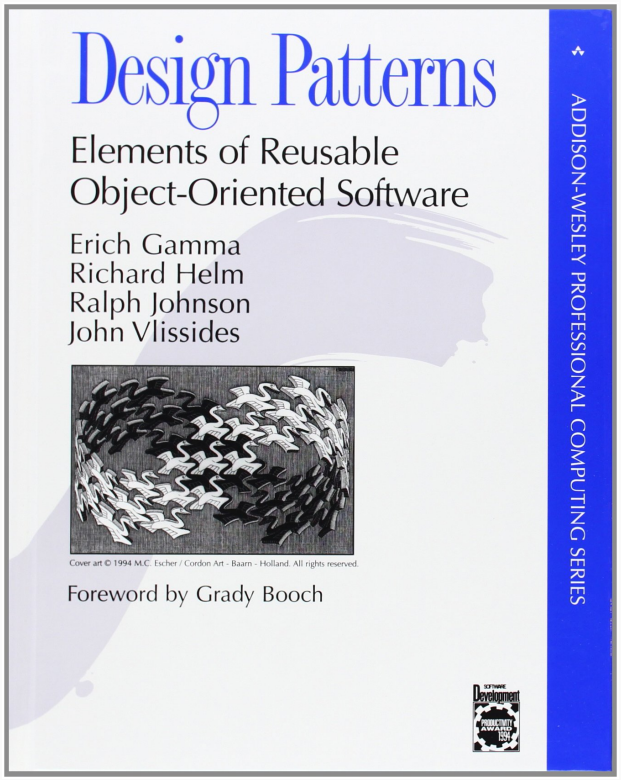
- En français : patrons de conception
- Intuition : différentes situations rencontrées par un développeur correspondent à un problème similaire, seul le contexte change
- Une solution similaire peut donc être appliquée
- Les design patterns forment un « catalogue » de solutions prêtes à l'emploi : l'expérience des générations précédentes de développeurs mise à disposition des nouveaux développeurs
- Nommage d'une structure de haut niveau qu'on ne peut pas exprimer directement sous forme de code $\Rightarrow$ Vocabulaire commun à tous les développeurs

Un peu de culture générale

- Origine du concept : architecture (1977 : *A Pattern Language : Towns, Buildings, Construction*, C. Alexander, S. Ishikawa et M. Silverstein)
- Années 80 : premières adaptations du concept au génie logiciel
- 1994 : Publication du *Gang of Four (GoF)*

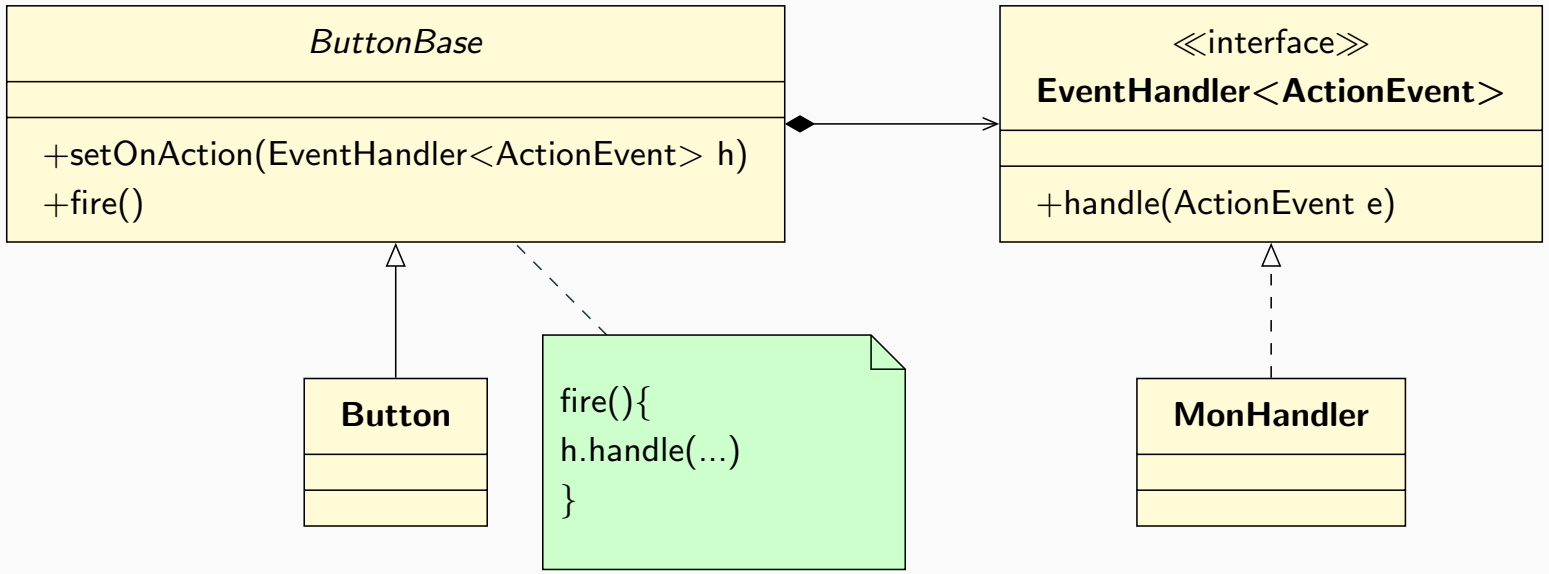
  Design Patterns : Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson et J. Vlissides





Premier exemple : gestion d'événements en JavaFX

- Rappel : lorsqu'un événement se produit dans une interface graphique, le composant lié à cet événement fait automatiquement appel à la méthode handle() de son Handler



- On peut avoir d'autres classes à la place du Button (*e.g.* CheckBox) ou de MonHandler (gestion différente d'un même événement)
- On retrouve la même structure pour les autres types d'événements (MouseEvent, KeyEvent,...)

Deuxième exemple : mises à jour et sauvegardes multiples (1/10 : Scénario)

Scénario simple :

- Notre application a besoin de sauvegarder les changements de l'état d'un objet dès qu'il se produit
  - Pour l'exemple, ce sera la modification d'un attribut de type **int** ou **double**
- Plusieurs modes de sauvegardes peuvent être implémentés, et utilisés au choix, voire même en parallèle
  - sauvegarde dans un fichier/une base de données
  - envoi par email
  - affichage sur la console
  - ...

Deuxième exemple : mises à jour et sauvegardes multiples (2/10 : La classe Donnees)

```
public abtract class Donnees {
  private List<Sauvegarde> sauvegardes ;

  public Donnees(){
    sauvegardes = new ArrayList<Sauvegarder>();
  }

  public void ajoutSauvegarde(Sauvegarde s){
    sauvegardes.add(s);
  }

  protected void signalerMAJ(){
    for(Sauvegarde s : sauvegardes) s.sauver(this) ;
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (3/10 : La classe DonneesInt)

```
public abtract class DonneesInt extends Donnees {
  private int donnees ;

  public DonneesInt(int d){
    super();
    donnees = d ;
  }
  public void miseAJour(int d){
    donnees = d ;
    signalerMAJ();
  }
  public String toString(){
    return System.currentTimeMillis()
                        + " : " + donnees ;
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (4/10 : La classe DonneesDouble)

```
public abtract class DonneesDouble extends Donnees {
  private double donnees ;

  public DonneesInt(double d){
    super();
    donnees = d ;
  }
  public void miseAJour(double d){
    donnees = d ;
    signalerMAJ();
  }
  public String toString(){
    return System.currentTimeMillis()
                        + " : " + donnees ;
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (5/10 : L'interface Sauvegarde)

```
public interface Sauvegarde {
  public void sauver(Donnees d) ;
}
```

- Première Sauvegarde concrète : sur la sortie standard

```
public class SauvegardeStdout implements Sauvegarde {
  public void sauver(Donnees d){
    System.out.println(d);
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (6/10 : Sauvegarde dans un fichier)

```
public class SauvegardeFichier implements Sauvegarde {
  private File fichier ;

  public SauvegardeFichier(String filename){
    fichier = new File(filename);
  }

  public void sauver(Donnees d){
    // Ouvrir le fichier
    // Ajouter d.toString() a la suite du contenu
    // Fermer le fichier
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (7/10 : Sauvegarde par email)

```
public class SauvegardeEmail implements Sauvegarde {
  private String email ;

  public SauvegardeEmail(String email){
    this.email = email ;
  }

  public void sauver(Donnees d){
    // Envoyer d.toString() par email
    // a l'adresse voulue
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (8/10 : Initialisation du système)

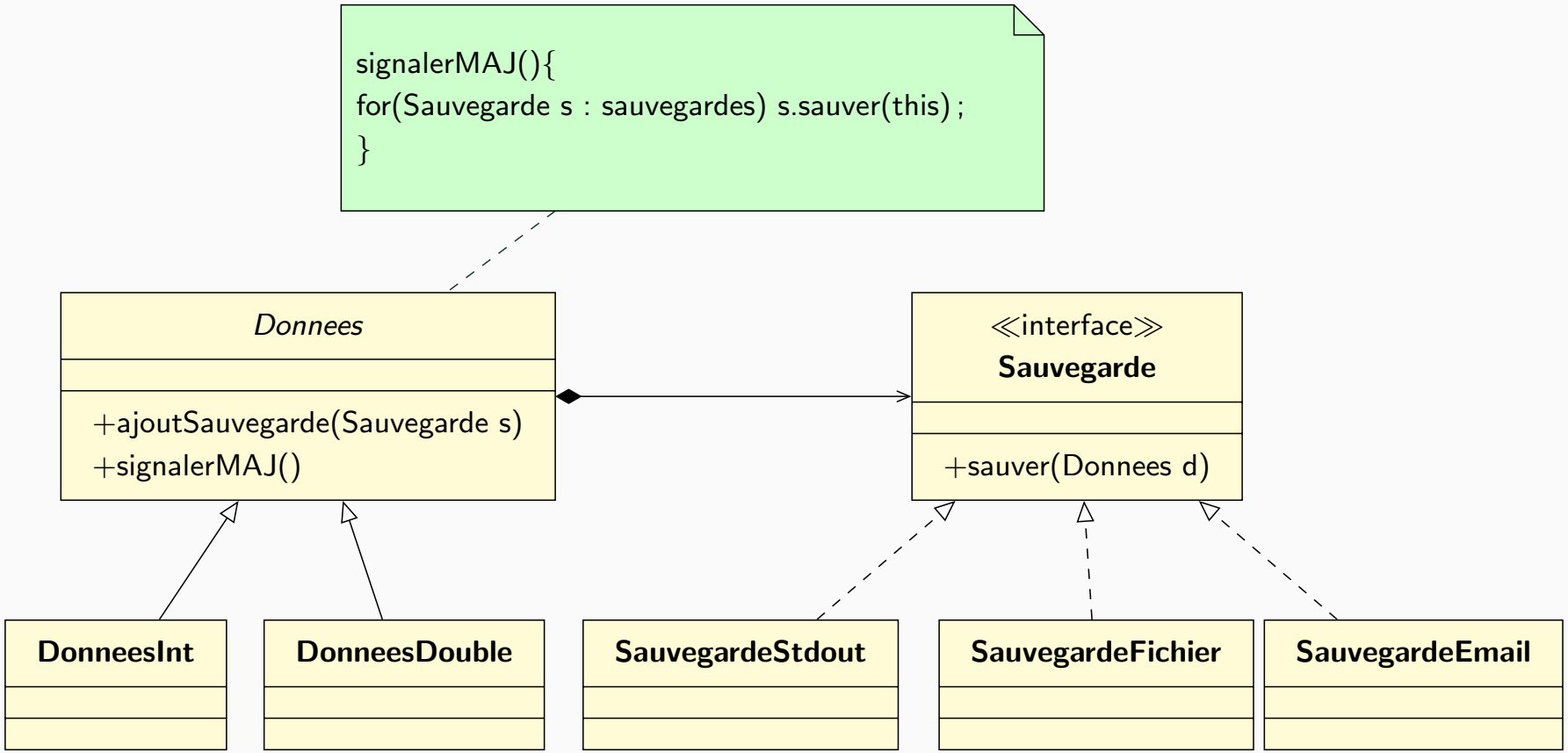
```
public class SystemeSauvegarde {
  public static void main(String[] args){
    List<Donnees> donnees = new ArrayList<Donnees>();
    donnees.add(new DonneesInt(0));
    donnees.add(new DonneesDouble(0));

    Sauvegarde stdout = new SauvegardeStdout();
    Sauvegarde fichier = new SauvegardeFichier(
                        "/chemin/vers/fichier.log");
    Sauvegarde email = new SauvegardeEmail(
              "sauvegarde@parisdescartes.fr");
```

Deuxième exemple : mises à jour et sauvegardes multiples (9/10 : Initialisation du système)

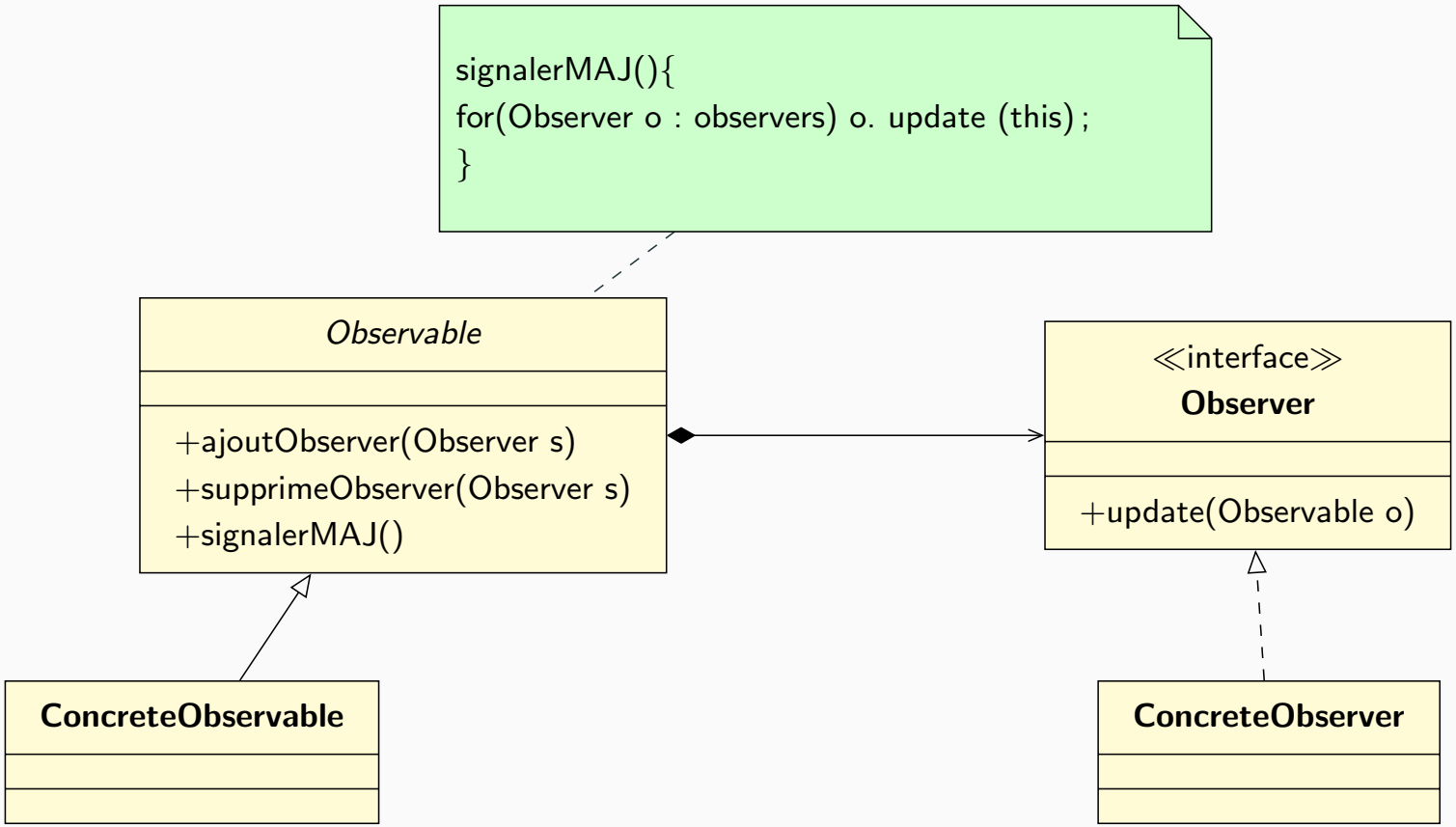
```
        for(Donnees d : donnees){
          d.ajouteSauvegarde(stdout);
          d.ajouteSauvegarde(fichier));
          d.ajouteSauvegarde(email);
        }
        // Suite des operations...
    }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (10/10 : Diagramme de classe)



Le design pattern Observer

- Ce patron de conception permet à des objets (les observateurs) d'agir en fonction de la modification de l'état d'un objet (l'observé)



Description minimale d'un design pattern

- Nom et classification
- Utilité

- Structure (UML)

Description minimale d'un design pattern

- Nom et classification $\rightarrow$ Observer, patron de comportement
- Utilité $\rightarrow$ permet à des objets (les observateurs) d'agir en fonction de la modification de l'état d'un objet (l'observé)
- Structure (UML) $\rightarrow$ voir slide précédent

Plusieurs types de patrons

- Patrons de création
  - Patrons qui permettent de créer des objets de manière adaptée à la situation
- Patrons de structure
  - Patrons qui concernent les relations entre différentes entités du programme
- Patrons de comportement
  - Patrons qui concernent la communication entre différents objets et l'adaptation de leur comportement (*e.g.* Observer)

Patrons de création

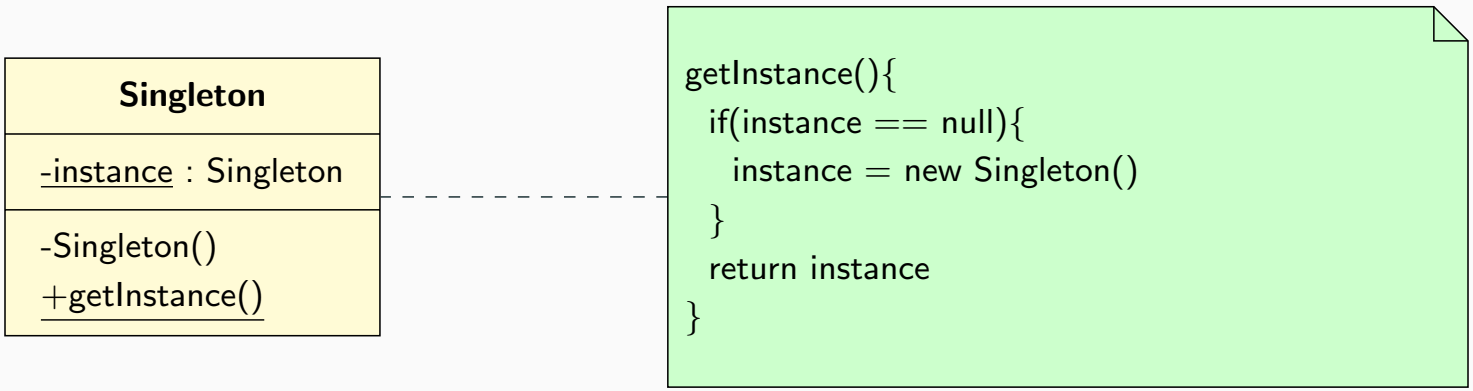
Les patrons de création

Ou *Creational design patterns*

- Abstraire le processus de création d'objets
- Rendre indépendante la façon dont les objets sont créés
- Encapsuler la connaissance de la classe concrète qui est instanciée
- Cacher ce qui est créé, quand, comment, par qui

Le Singleton

- On a besoin d'une classe dont on doit créer une seule instance, et cette instance doit être disponible n'importe où dans le programme
- Exemple d'application : gestion des ressources d'une application (file d'attente de l'imprimante), configuration des propriétés de l'application,...



Une file d'impression en Singleton

```
public class PrintQueue {
  private List<String> printQueue ;
  private static PrintQueue instance = null ;

  private PrintQueue(){
    printQueue = new LinkedList<String>();
  }

  public static PrintQueue getInstance(){
    if(instance == null)
      instance = new PrintQueue();
    return instance ;
  }

  // Methodes de la file : ajout, retrait,...
}
```

Une file d'impression en Singleton : la synchronisation

- La première version de PrintQueue pose problème si plusieurs threads peuvent l'utiliser : deux appels simultanés de getInstance() peuvent créer deux instances !

```
public class PrintQueue {
  // ...
  // Attributs et constructeur

  public static synchronized PrintQueue getInstance(){
    if(instance == null)
      instance = new PrintQueue();
    return instance ;
  }

  // Methodes de la file : ajout, retrait,...
}
```

Une file d'impression en Singleton : la synchronisation améliorée

- La synchronisation a un coût, on préfère éviter de la faire à chaque appel de getInstance()

```
public class PrintQueue {
  // Attributs et constructeur
  public static PrintQueue getInstance(){
    if(instance == null)
      synchronized(PrintQueue.class){
        if(instance == null)
          instance = new PrintQueue();
      }
    return instance ;
  }
  // Methodes de la file : ajout, retrait,...
}
```

- PrintQueue.**class** est une instance de Class<PrintQueue>

Factory et création générique de compilateurs

- Scénario : pour créer un IDE multi-langages, on doit pouvoir choisir facilement parmi plusieurs compilateurs disponibles, ajouter de nouveaux compilateurs, ou remplacer un compilateur par un autre (par exemple pour intégrer une nouvelle version du langage)
- Le design pattern Factory (ou Fabrique) peut être utilisé

Les classes des compilateurs

```
public abstract class AbstractCompiler {
  public abstract void compile(String fileName);
}

public class JavaCompiler extends AbstractCompiler {
  public void compile(String fileName){
    // Compile Java code
  }
}

public class CCompiler extends AbstractCompiler {
  public void compile(String fileName){
    // Compile C code
  }
}
```

- On peut prévoir des compilateurs pour autant de langages qu'on veut

La Factory

```
public abstract class AbstractCompilerFactory {
  public abstract AbstractCompiler
          makeCompiler(String langage) ;
}

public class CompilerFactory
           extends AbstractCompilerFactory {
  public AbstractCompiler
                 makeCompiler(String language){
    switch(language){
      case "Java": return new JavaCompiler() ;
      case "C": return new CCompiler() ;
      // ...
    }
  }
}
```

Le client

- On suppose que la ligne de commande contient le langage et le chemin du fichier

```
public static void main(String[] args){
  AbstractCompilerFactory f = new CompilerFactory() ;
  AbstractCompiler c = f.makeCompiler(args[0]);
  c.compile(args[1]);
}
```

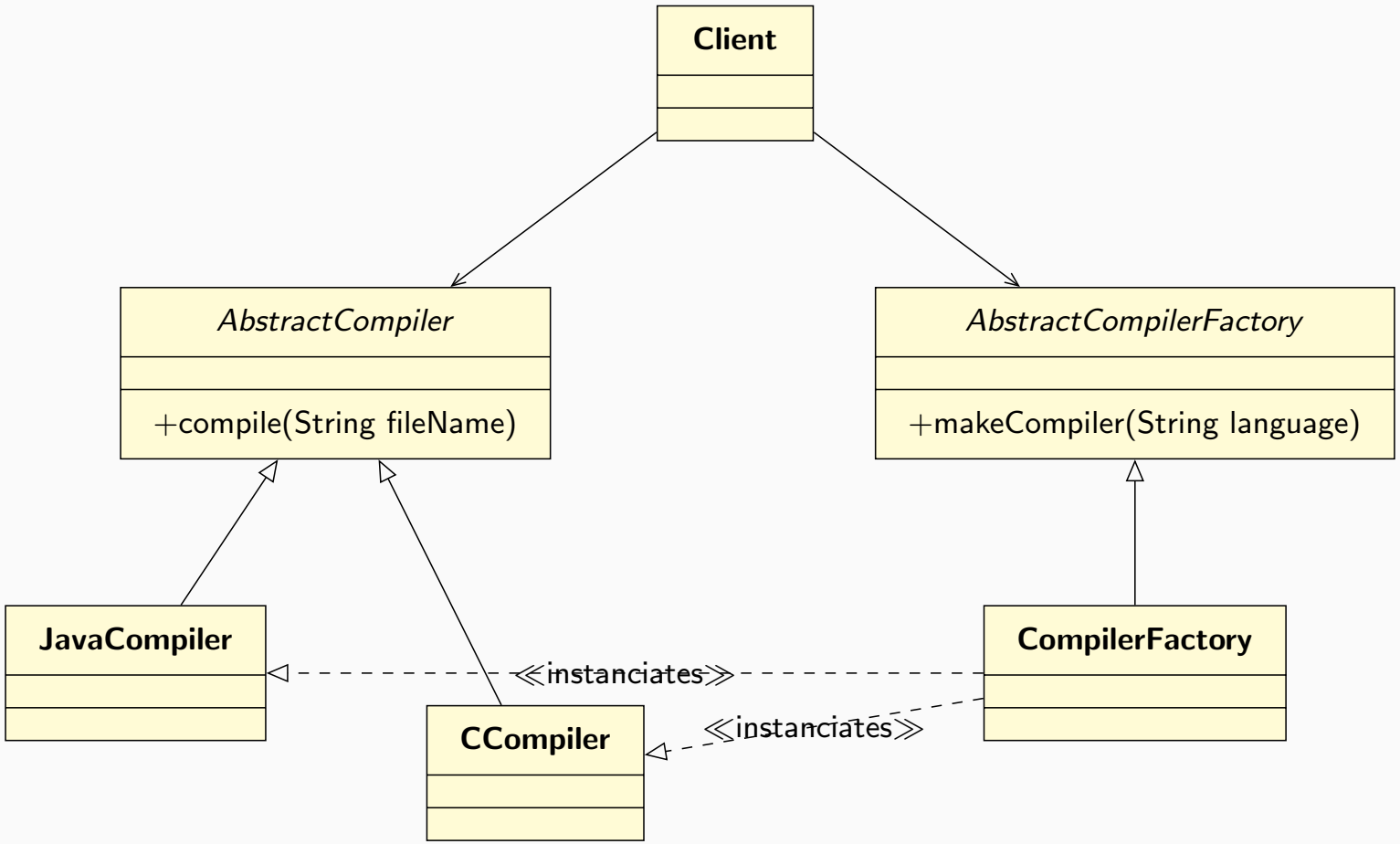
Le client

- On suppose que la ligne de commande contient le langage et le chemin du fichier

```
public static void main(String[] args){
    AbstractCompilerFactory f = new CompilerFactory() ;
    AbstractCompiler c = f.makeCompiler(args[0]);
    c.compile(args[1]);
}
```

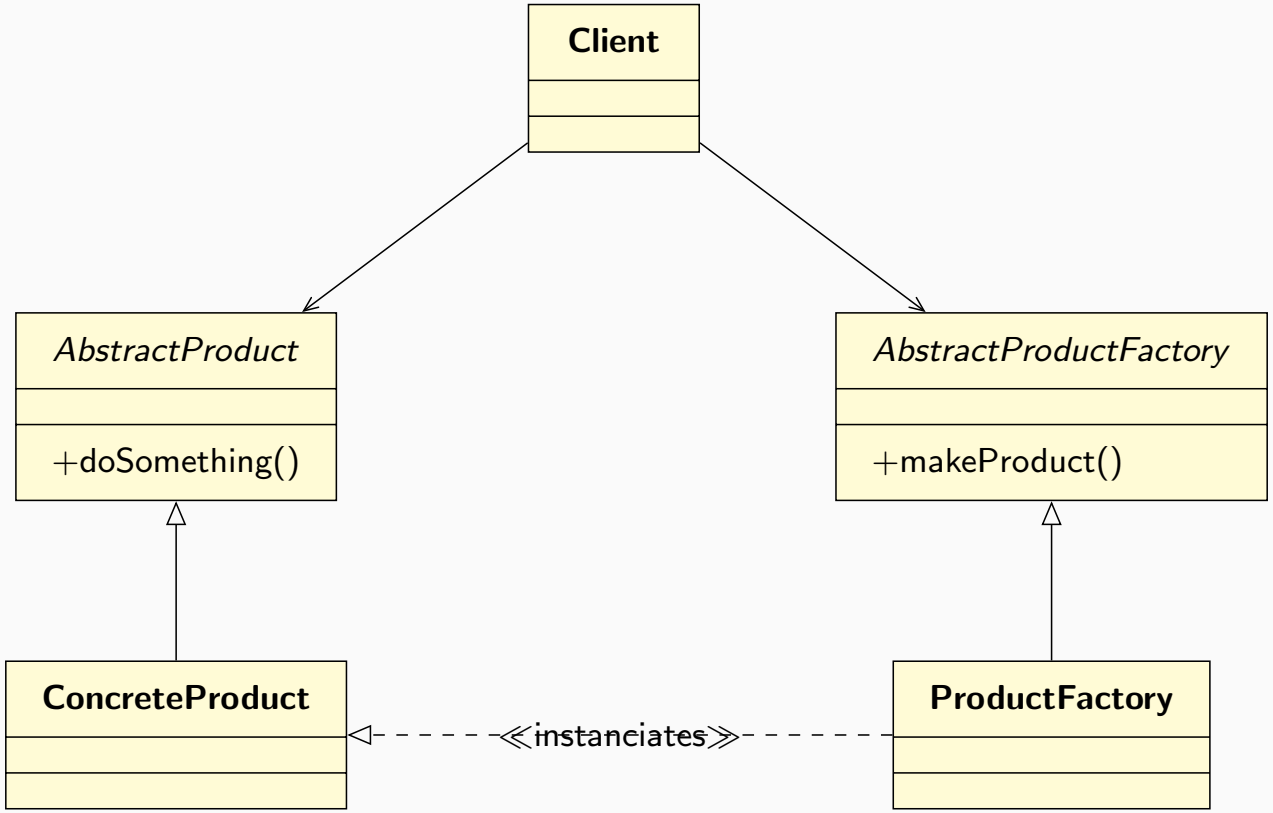
- Si on veut modifier les compilateurs disponibles, il suffit de créer une nouvelle classe qui hérite de AbstractCompilerFactory et adapter la première ligne

Le diagramme de classe des compilateurs



Le design pattern Factory

- Utilité : Définir une interface générique pour la création d'objets, faire référence aux objets créés uniquement via leur interface commune



Les autres patrons de création

Le GoF définit plusieurs patrons de création

- Abstract Factory : Fournit une interface pour créer des familles d'objets liés les uns aux autres, sans spécifier la classe concrète
- Builder : Sépare la construction d'un objet complexe de sa représentation
- Factory : déjà vu
- Prototype : Permet la création de nouvelles instances d'une classe en faisant des copies d'une classe qui sert de modèle
- Singleton : déjà vu

Patrons de structure

Les patrons de structure

Ou *Structural design patterns*

- Facilitent le design de l'application en identifiant des moyens simples de représenter les relations entre les entités
- Séparent les interfaces des implémentations

Un patron familier : Composite

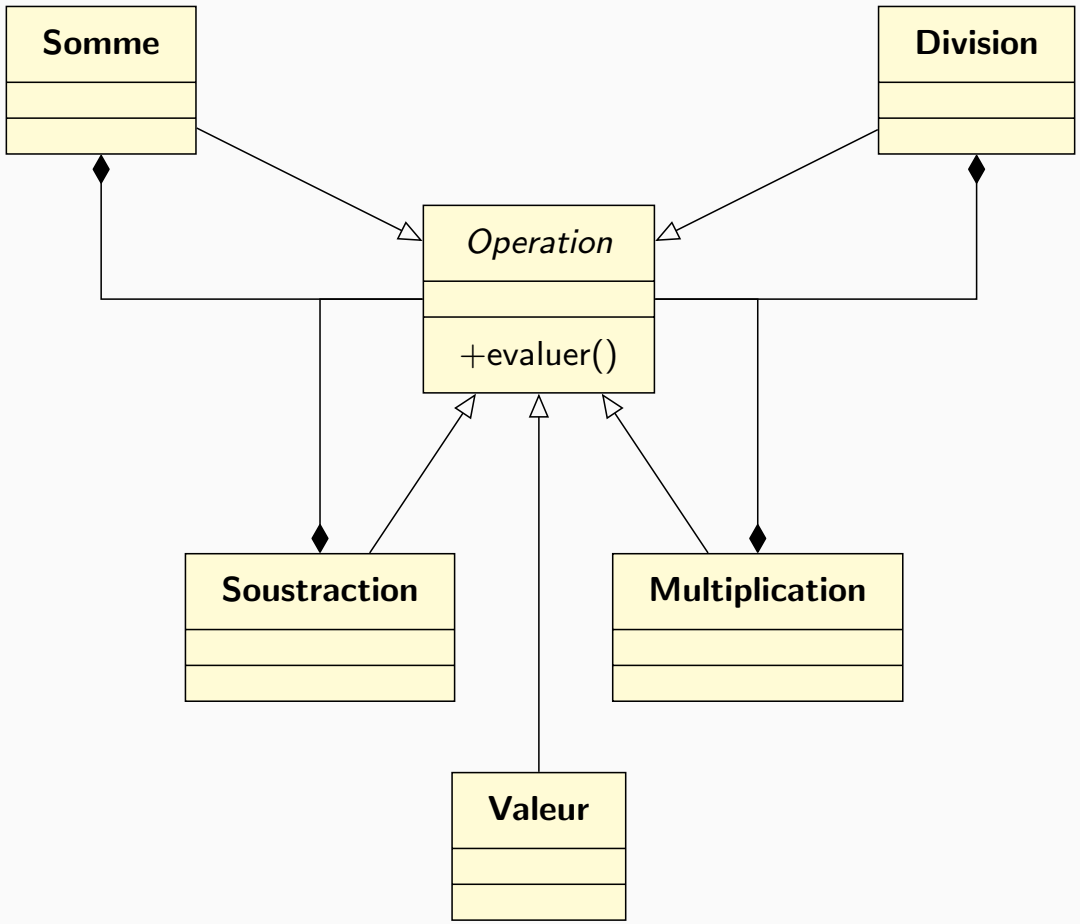
- Utilité : représenter une hierarchie d'objets, et ignorer la différence entre objets « simples » et objets « complexes »

Exercice VI (POO et opérations mathématiques)
On peut représenter une opération mathématique sous forme d'un arbre dont les noeuds sont des opérateurs, et les feuilles sont des valeurs numériques. On souhaite représenter de telles expressions et pouvoir évaluer leur valeur.

- La classe (abstraite) Operateur représente aussi bien un noeud qu'une feuille de l'arbre. Un Operateur est caractérisé par son arité, qui est 0 dans le cas des feuilles, et un entier > 0 pour les noeuds internes.
- Les opérateurs arithmétiques habituels (Somme, Soustraction, Multiplication, Division), d'arité 2, sont des classes filles d'Operateur.
- On peut également définir des classes AdditionNAire et MultiplicationNAire pour les versions générales de l'addition et la multiplication (l'arité est donc variable).
- Une Valeur a une arité nulle, et représente un nombre réel.

La classe Operateur contient une méthode abstraite **public abstract double** evaluer() ; qui retourne la valeur de cet opérateur pour ses opérandes.
Implémentez ces classes.

Diagramme de classe des opérations

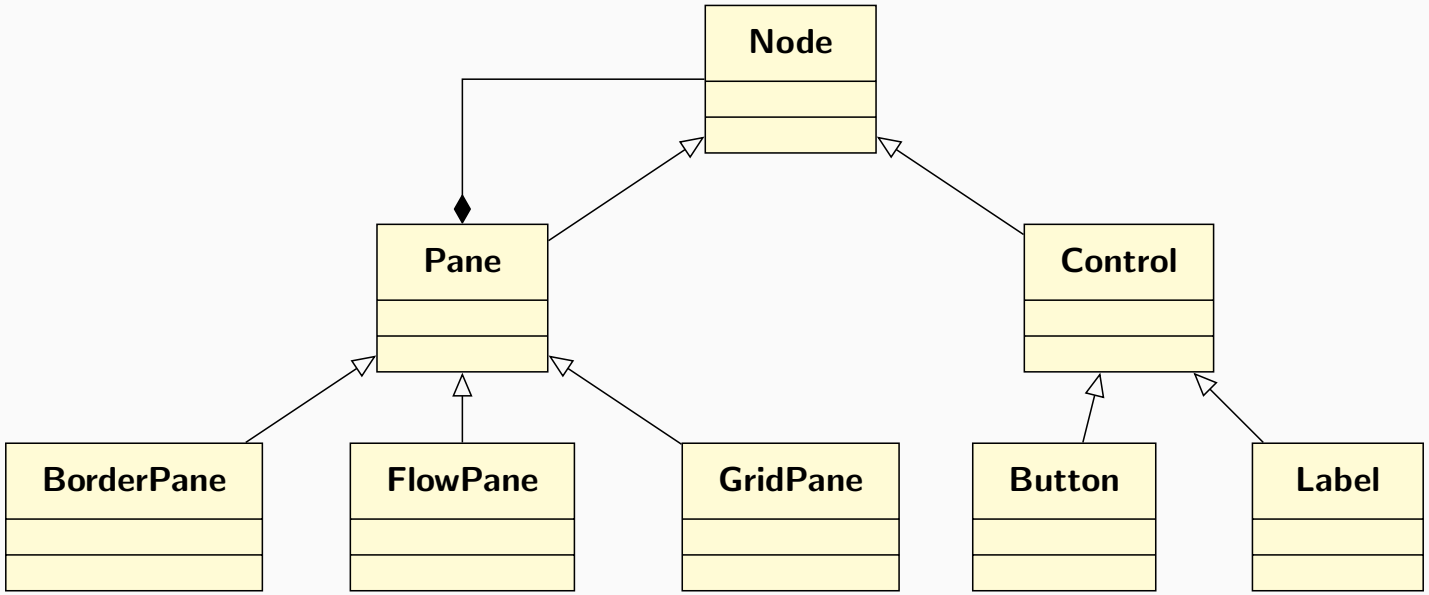


Composite et JavaFX

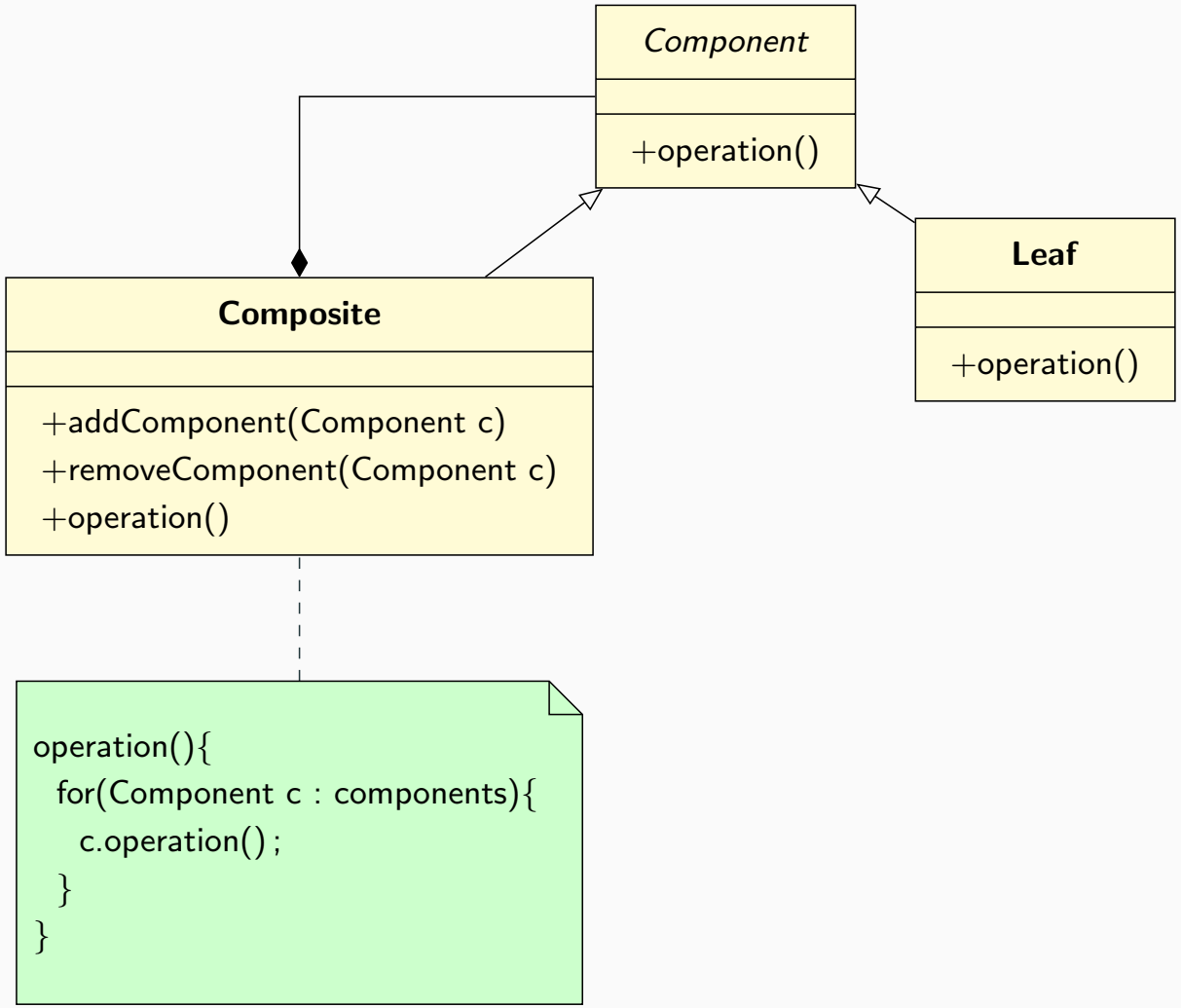
- On a vu une instance du pattern Composite en JavaFX. Une idée ?

Composite et JavaFX

- On a vu une instance du pattern Composite en JavaFX. Une idée ?
- Un Pane est un composant qui peut contenir des composants.
  Extrait du diagramme de classes

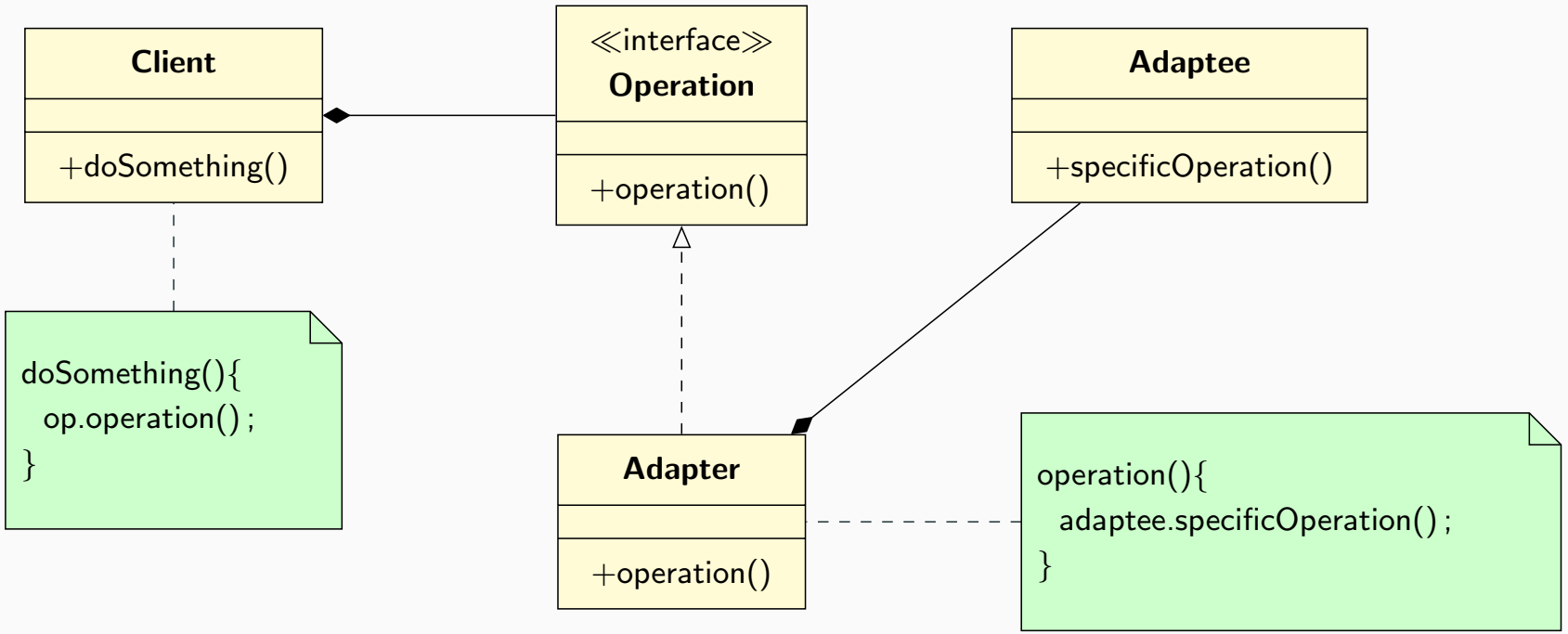


Le diagramme du patron Composite



Le pattern Adapter

- Permet de faire travailler ensemble des classes qui sont incompatibles



- On peut donc utiliser du code d'une autre API (l'adaptée) qui fournit le service dont on a besoin, même si ce code n'est pas directement compatible avec le notre

Les autres patrons de structure

Le GoF définit d'autres patrons de structure :

- Adapter : déjà vu
- Bridge : sépare une abstraction de son implémentation et permet de les faire varier indépendemment
- Composite : déjà vu
- Decorator : permet d'ajouter dynamiquement des fonctionnalités ou des propriétés à un objet
- Facade : fournit une interface simple pour un ensemble complexe d'interfaces
- Flyweight : permet d'économiser de l'espace en partageant certaines données qui sont communes à des objets similaires
- Proxy : fournit une classe intermédiaire qui sert de substitut à l'utilisation d'une classe donnée, et permet d'en contrôler l'utilisation ou d'y ajouter des fonctionnalités

Patrons de comportement

Les patrons de comportement

Ou *Behavioral design patterns*

- Modélisation du comportement des objets et de la façon dont ils peuvent communiquer
- Séparation des structures de données et des algorithmes qui manipulent ces structures

Strategy

- Le design pattern strategy permet de séparer les algorithmes des classes. Plusieurs intérêts :
  - Possibilité de modifier dynamiquement l'algorithme utilisé durant l'exécution du programme
  - Facilité de mise à jour de l'application avec un algorithme plus efficace
  - Possibilité de choisir l'algorithme le mieux adapté à la situation

Exemple : la satisfiabilité des formules logiques (1/4)

- Une formule propositionnelle est la composition de variables booléennes avec des opérateurs comme le ET, le OU ou le NON
  - c'est exactement ce qu'on met dans un **if** (...) ou un **while** (...)
- Savoir si une formule est vraie quand on connaît les valeurs des variables est facile
- Savoir s'il est possible de trouver une valeur de chaque variable qui rendra la formule vraie est par contre difficile en général
  - c'est le problème SAT, un sujet de recherche en informatique fondamentale et en intelligence artificielle
  - dans certains cas, la formule a une syntaxe particulière qui rend le problème plus facile que le cas général

Exemple : la satisfiabilité des formules logiques (2/4)

```
public class Formula {
  // ...
  private Strategy strategy ;

  public boolean isSatisfiable(){
    if(isHornFormula()){
      strategy = new HornStrategy()
    }else if (isKromFormula()){
      strategy = new KromStrategy();
    }else{
      strategy = new GeneralStrategy();
    }
    return strategy.solve(this);
  }
}
```

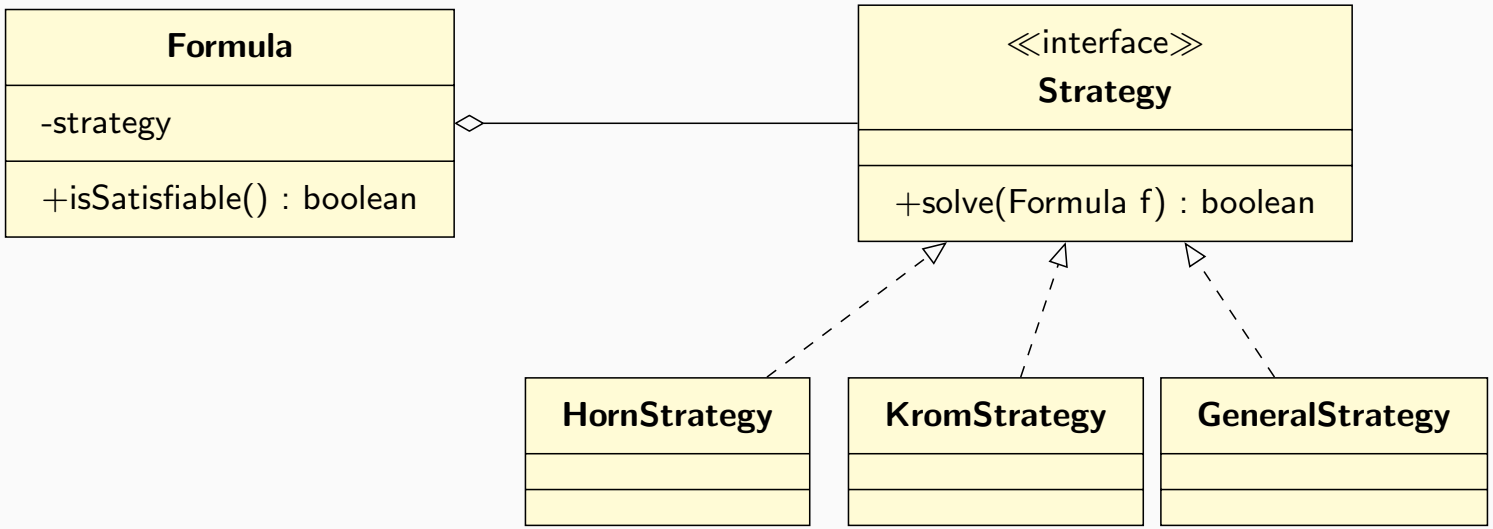
Exemple : la satisfiabilité des formules logiques (3/4)

```
public interface Strategy{
  public boolean solve(Formula f);
}

public class HornStrategy implements Strategy {
  public boolean solve(Formula f){
    // Algo pour les formules de Horn
  }
}
```

- Même chose pour KromStrategy et GeneralStrategy

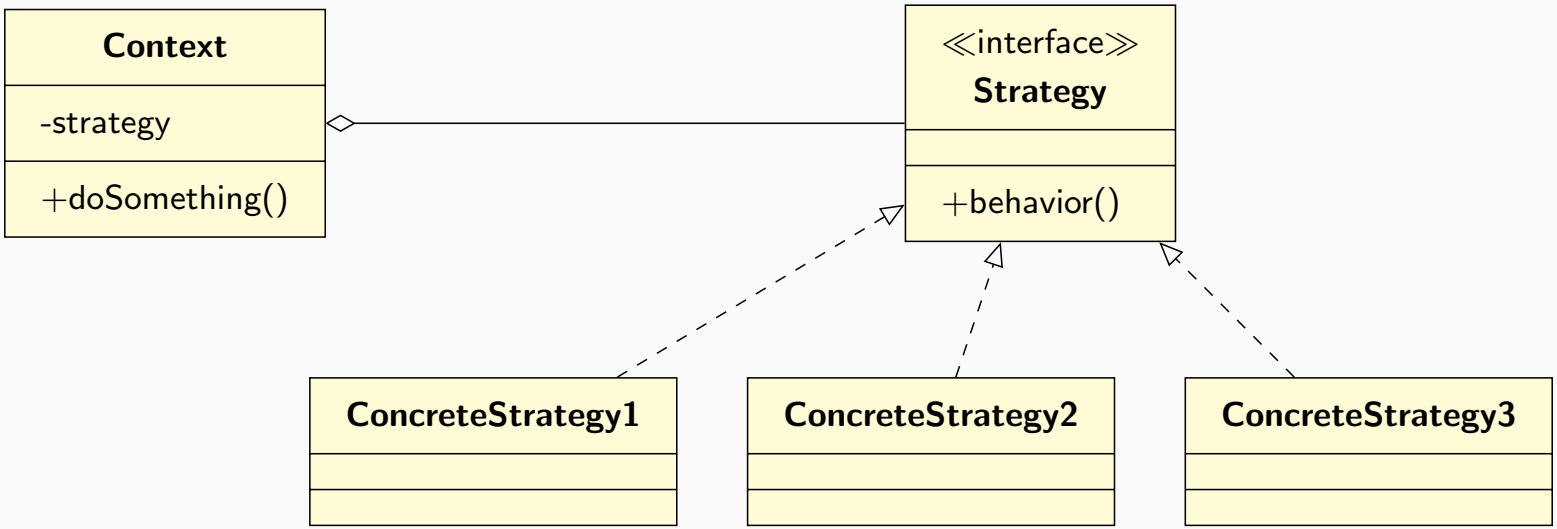
Exemple : la satisfiabilité des formules logiques (4/4)



Un autre exemple de Strategy

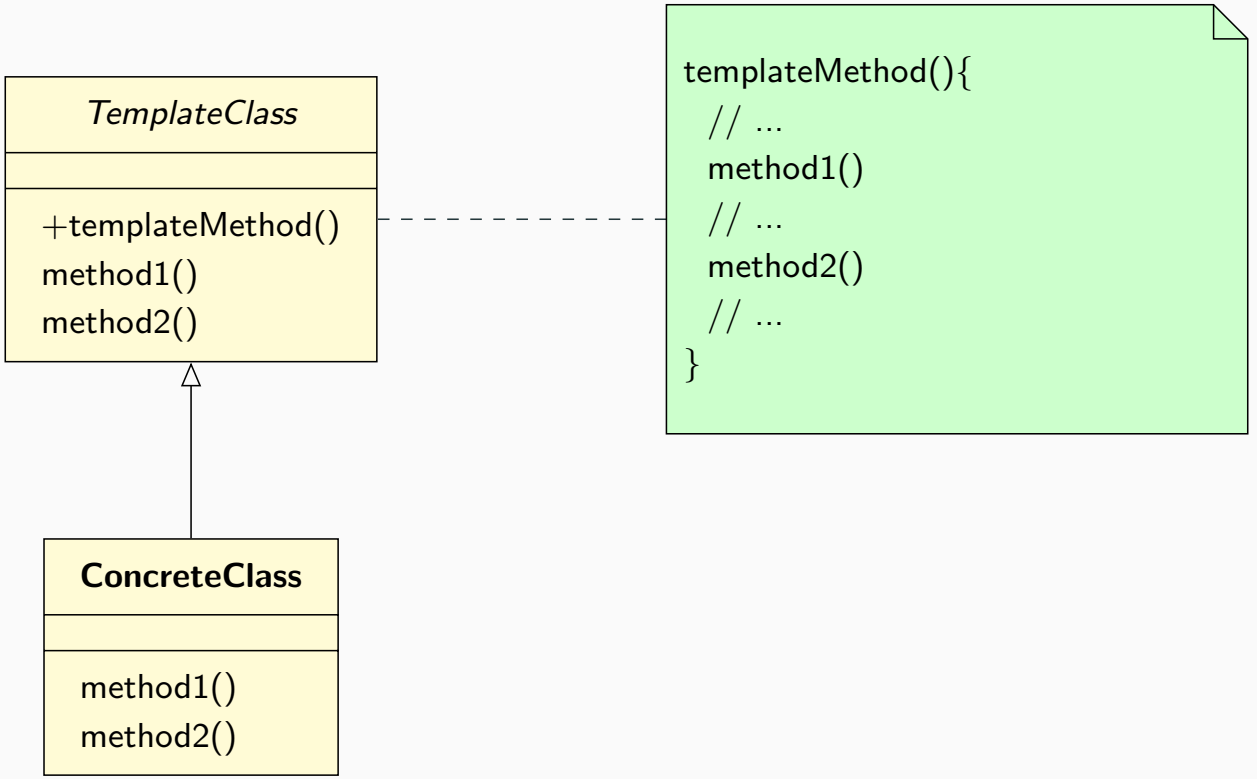
- Dans un jeu vidéo, un personnage non joueur (PNJ) peut avoir différents comportements vis à vis du joueur selon le contexte
- Ces différentes stratégies peuvent être mises à jour selon le contexte :
  - un comportement de base
  - un nouveau comportement une fois que le joueur a accompli certaines actions (*e.g.* succès d'une quête)
  - un comportement agressif si le joueur l'a attaqué
  - ...

Diagramme général pour le patron Strategy



Template method

- Utilité : définit la structure d'un algorithme tout en laissant certaines étapes à la charge de sous-classes



Template method et parcours de graphe (1/3)

```
public class Node {
  private int value ;
  private Node left ;
  private Node right ;

  public Node(int v){
    value = v ;
    left = null ;
    right = null ;
  }

  // Getters et setters
}
```

Template method et parcours de graphe (2/3)

```
public abstract class AbstractDFS {
  public void dfs(Node n){
    if(n != null){
      dfs(n.getLeft());
      dfs(n.getRight());
      doSomething(n);
    }
  }

  public abstract void doSomething(Node n);
}
```

Template method et parcours de graphe (3/3)

```
public class PrinttDFS {
  public void doSomething(Node n){
    System.out.println(n.getValue());
  }
}
```

- Il suffit de définit d'autres classes filles de AbstractDFS pour donner un comportement différent à l'algorithme DFS

Les autres patrons de comportement (1/2)

Le GoF définit plusieurs patrons de comportement

- Chain of responsibility : sépare l'émetteur d'une requête de ses récepteurs
- Command : Encapsule une requête sous forme d'objet
- Interpreter : définit une représentation pour la grammaire d'un langage, et permet d'interpréter des phrases dans ce langage
- Iterator : fournit un moyen d'accéder aux éléments d'un objet composé, sans rendre explicite sa représentation interne
- Mediator : définit un objet qui encapsule la façon dont un ensemble d'objets interragit

Les autres patrons de comportement (2/2)

Le GoF définit plusieurs patrons de comportement

- Memento : capture l'état interne d'un objet, permet de le restaurer
- Observer : déjà vu
- State : Permet de modifier le comportement d'un objet en fonction de son état interne
- Strategy : déjà vu
- Template method : déjà vu
- Visitor : représente une opération qui doit être effectuée sur tous les éléments d'une structure complexe

En résumé

- Design pattern = des gens intelligents ont déjà pensé à des solutions pour des problèmes récurrents, alors
  - apprenons à reconnaître ces problèmes
  - apprenons à identifier la bonne solution
  - apprenons à la mettre en oeuvre
- le GoF n'a fait que débuter le processus, d'autres design patterns ont été proposés par la suite