UNIVERSITÉ DE RENNES 1

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

Méthodes de Développement Industriel (MDI)

Mathieu Acher

http://www.mathieuacher.com

Associate Professor

University of Rennes 1

UMR IRISA

Inria informatics mathematics

Objectifs de MDI

- Méthodes de développement industriel (MDI)
  - En fait: génie logiciel / software engineering
  - Comment développer des systèmes logiciels de plus en plus complexe?
- #1 Prendre conscience de la complexité des systèmes logiciels actuels et à venir
  - Les enjeux et l'impact sur le métier
- #2 Modélisation
  - UML, SysML
- **#3** Design patterns**, refactoring, test**
  - **OO avancé**
- **#4 Méthodes**

Agenda

- Contrôle de 2h sur les design patterns le 26 avril (dernier cours)
  - Savoir identifier un design pattern dans un code existant
  - Justifier ou critiquer l'utilisation d'un design pattern
  - Choisir un ou des design patterns pour un problème donné
  - Mettre en oeuvre un design pattern
  - Combiner des design patterns
- Maîtrise de UML nécessaire
  - Identifier les "rôles" dans un design pattern
  - Modéliser une implémentation d'un design pattern
    - Diagramme de classes

Agenda (1)

- Présentation des “design patterns” en cours
  - Pédagogie inversée
- Groupes de 2 ou 3
- 10’ de présentation sur un sujet
- 10’ discussions/questions
- Objectifs: “révision” + note

Agenda (2)

- Projet en Binôme
- A partir d'un code existant (jeu d'échecs)
  - Savoir identifier les design patterns (éventuellement justifier ou critiquer leur utilisation)
  - Choisir un ou des design patterns pour un problème donné
  - Mettre en oeuvre un design pattern; Combiner des design patterns
- Refactoriser le code
- Etendre le code
- Tester le code

Méthode / Notation

- Pour identifier les patrons de conception dans un diagramme de classes, vous utiliserez la convention suivante : le nom du patron utilisé est défini dans une ellipse ; cette dernière est reliée, à l'aide de flèches, aux différentes classes concernées par le patron de conception.
  - Nombreux exemples dans le cours

Méthode / Notation (2)

- Par exemple, la figure 1 illustre comment mettre en avant le patron de conception Fabrique dans un diagramme de classes. Les flèches, reliant le patron de conception aux classes concernées, portent le nom de l'élément de référence dans le patron (i.e. la classe Forme est le Produit et FabriqueForme la Fabrique).

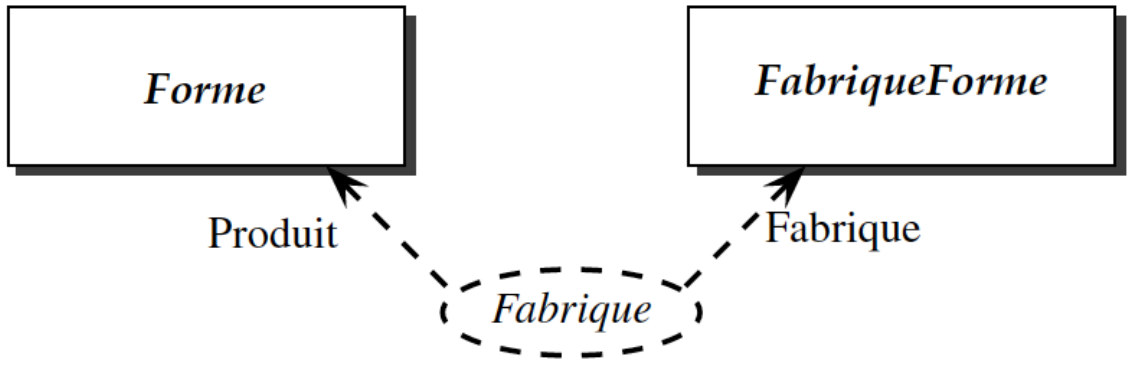


FIGURE 1 – Exemple de représentation d'un patron de conception dans un diagramme de classes UML

Flyweight
(poids mouche)

Poids-Mouche / Flyweight *(structurel)*

Problème:

Instanciation d'un (très très) grand nombre de petits objets

\Trop gourmand au niveau de la mémoire

Exemples :

Du texte contenant un grand nombre de caractères

Un jeu massivement multi-joueurs (les objets graphiques)

L'ADN

Patron de Conception
Poids-Mouche / Flyweight *(structurel)*

But :

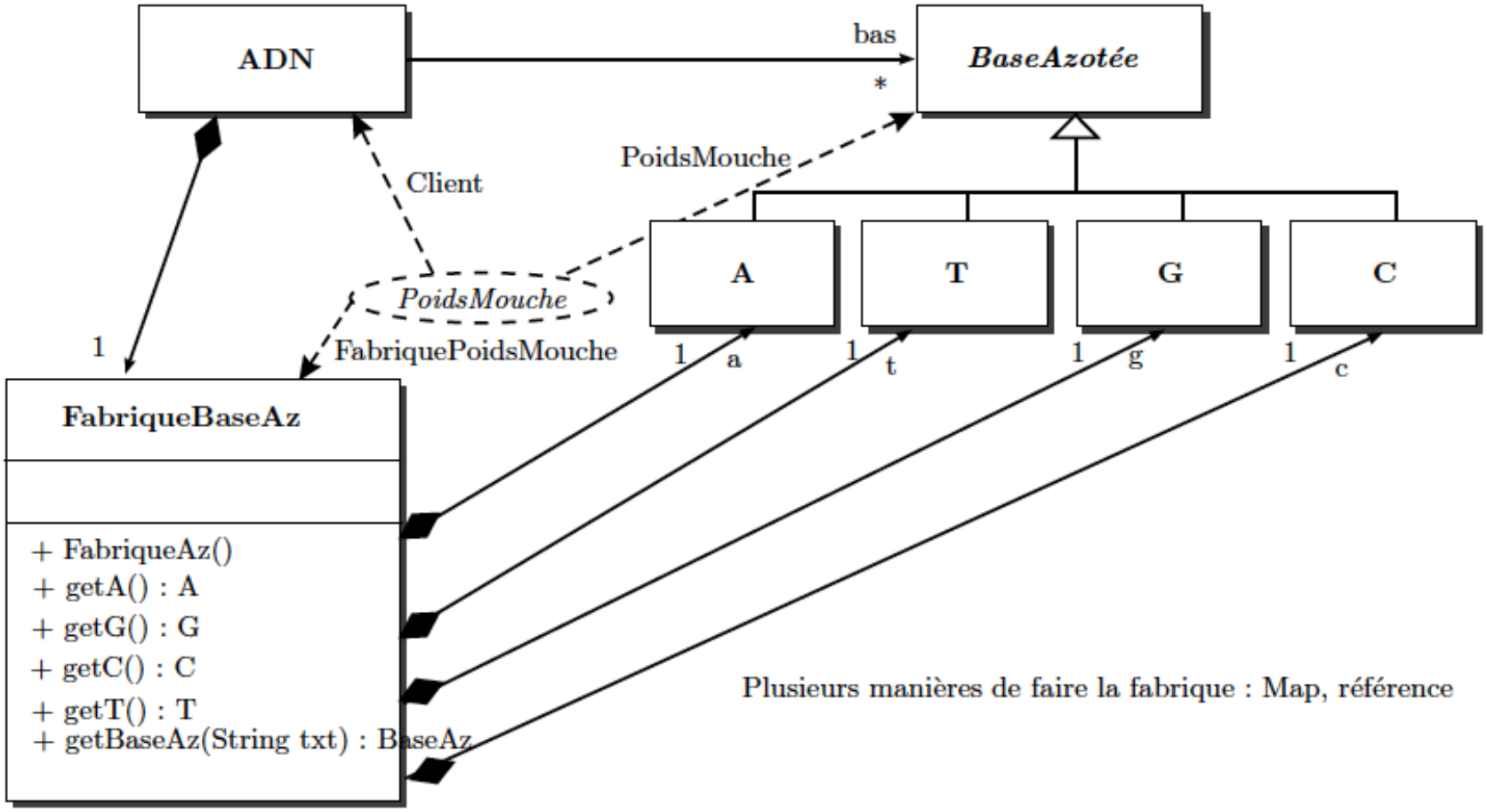
Partager efficacement un grand nombre de petits objets

Fonctionnement :

Certaines données sont extraites de l'objet
pour en minimiser le nombre d'instanciation

Exemple : l'ADN

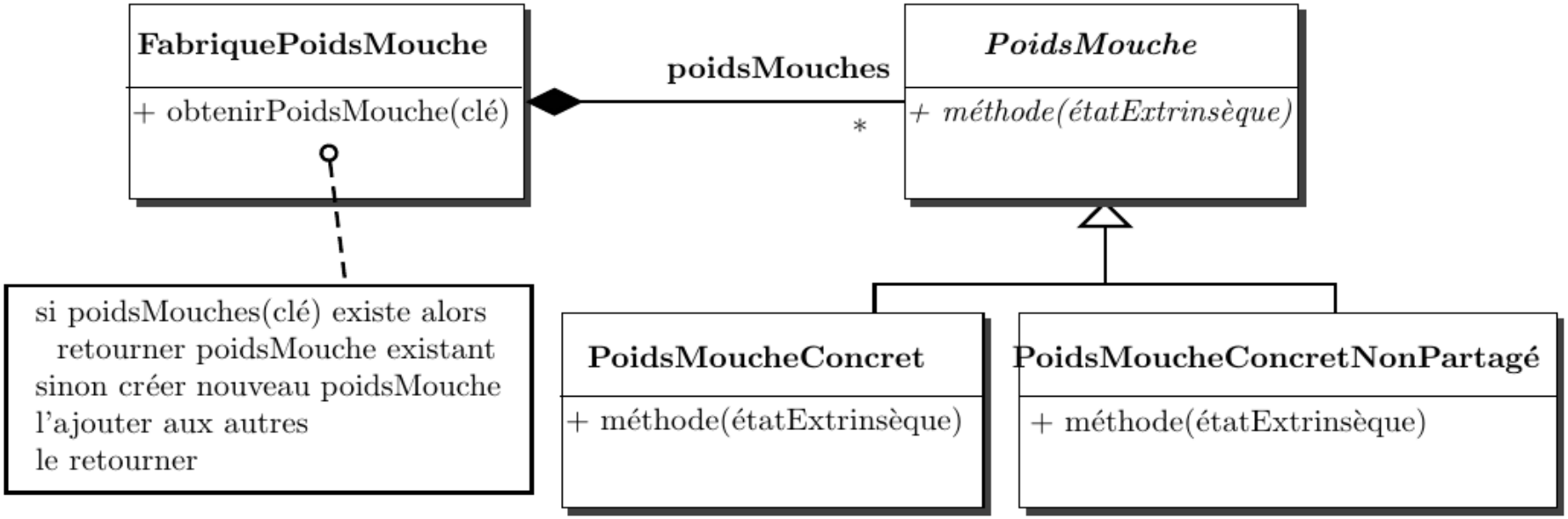
Seuls 4 bases azotées créées (G, A, T, C)
Leur position est stockée en dehors (dans l'ADN)
=> Sinon explosion du nombre d'instances de bases azotées



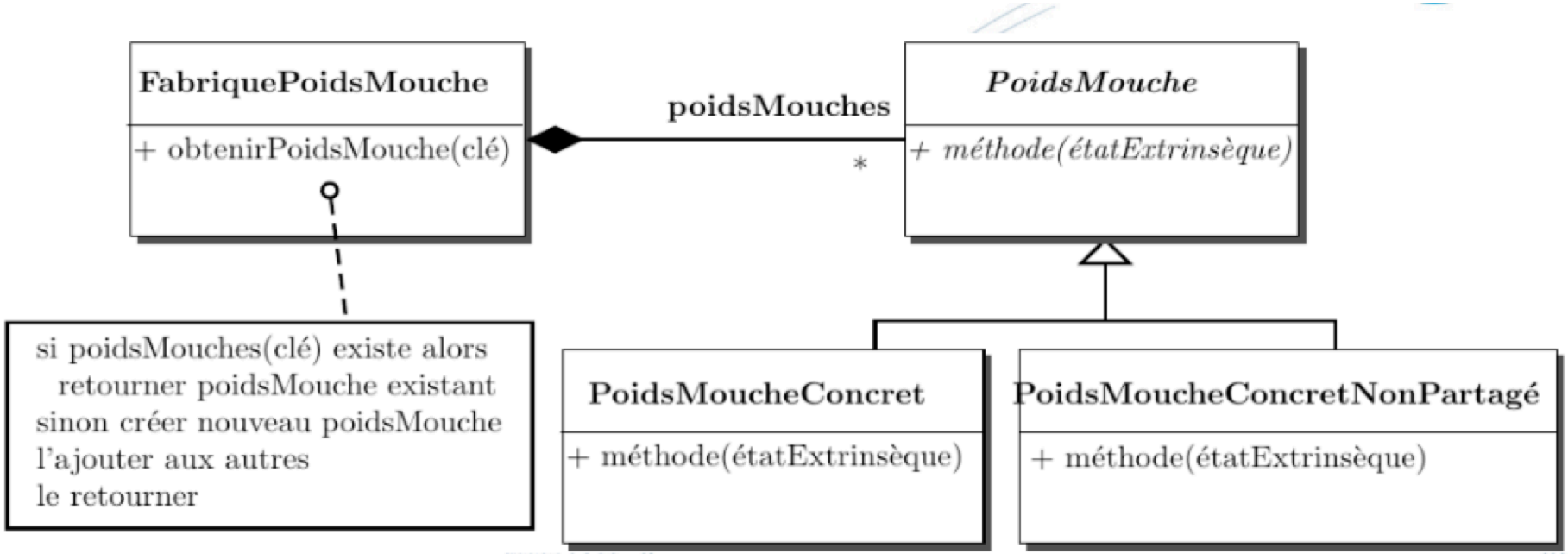
Plusieurs manières de faire la fabrique : Map, référence

```
public class FabriqueBaseAz {
        private A a;
        private C c;
        private G g;
        private T t;

    public FabriqueBaseAz() {
                super();// Ou alors lazy instanciation
                a = new A(); c = new C();
        g = new G(); t = new T();
        }

        public A getA() {
                return a;
        }
        public C getC() {
                return c;
        }
        public G getG() {
                return g;
        }
        public T getT() {
                return t;
        }
}
```

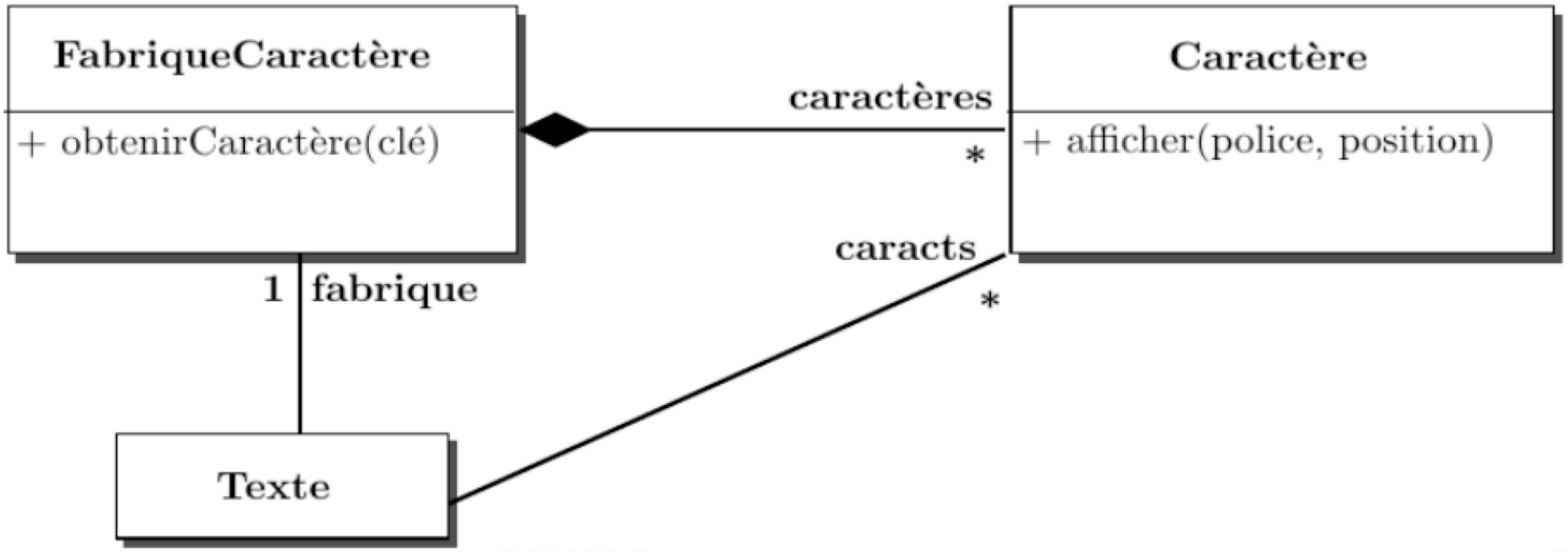
Poids-Mouche / Flyweight *(structurel)*



FabriquePoidsMouche : créer des objets uniquement si nécessaire (sinon retourne l'objet déjà existant)
L'utilisation du patron dans le code client
passe par la fabrique pour obtenir des instances



- **PoidsMoucheConcret** ne contient pas certaines données
  - ➔ Elles lui seront passées en paramètres (*étatExtrinsèque*)
- **PoidsMoucheConcretNonPartagé**: cas spécial de poids-mouche qui n'externalise pas ses données



- Exemple : un texte se compose de caractères
  - Externalisation de la position/police des caractères
  - 1 seul caractère pour 1 valeur (a, b, c, *etc.*)

```java
public final class FabriqueCaractere {
  public static FabriqueCaractere INSTANCE = new FabriqueCaractere();

  private Map<Integer, Caractere> mapCaracteres;

  private FabriqueCaractere() {
    mapCaracteres = new IdentityHashMap<Integer, Caractere>();
  }


  public Caractere ObtenirCaractere(char valeur) {
    Caractere car = mapCaracteres.get(valeur);

    if(car==null) {
      car = new Caractere(valeur);
      mapCaracteres.put((int)valeur, car);
    }
    return car;
  }
}
```

```java
public class Caractere {
  protected char valeur;
  //..

  public Caractere(char valeur) {
    this.valeur = valeur;
  }

  public void afficher(Graphics2D g, Point position) {
    //...
  }
}
```

```java
public class Texte {
  protected List<Caractere> caracts;

  public Texte() {
    caracts = new ArrayList<Caractere>();
  }

  public void afficher(Graphics2D g) {
    //...
  }

  public void ajouterCaractere(char valeur) {
    caracts.add(FabriqueCaractere.INSTANCE.ObtenirCaractere(valeur));
```

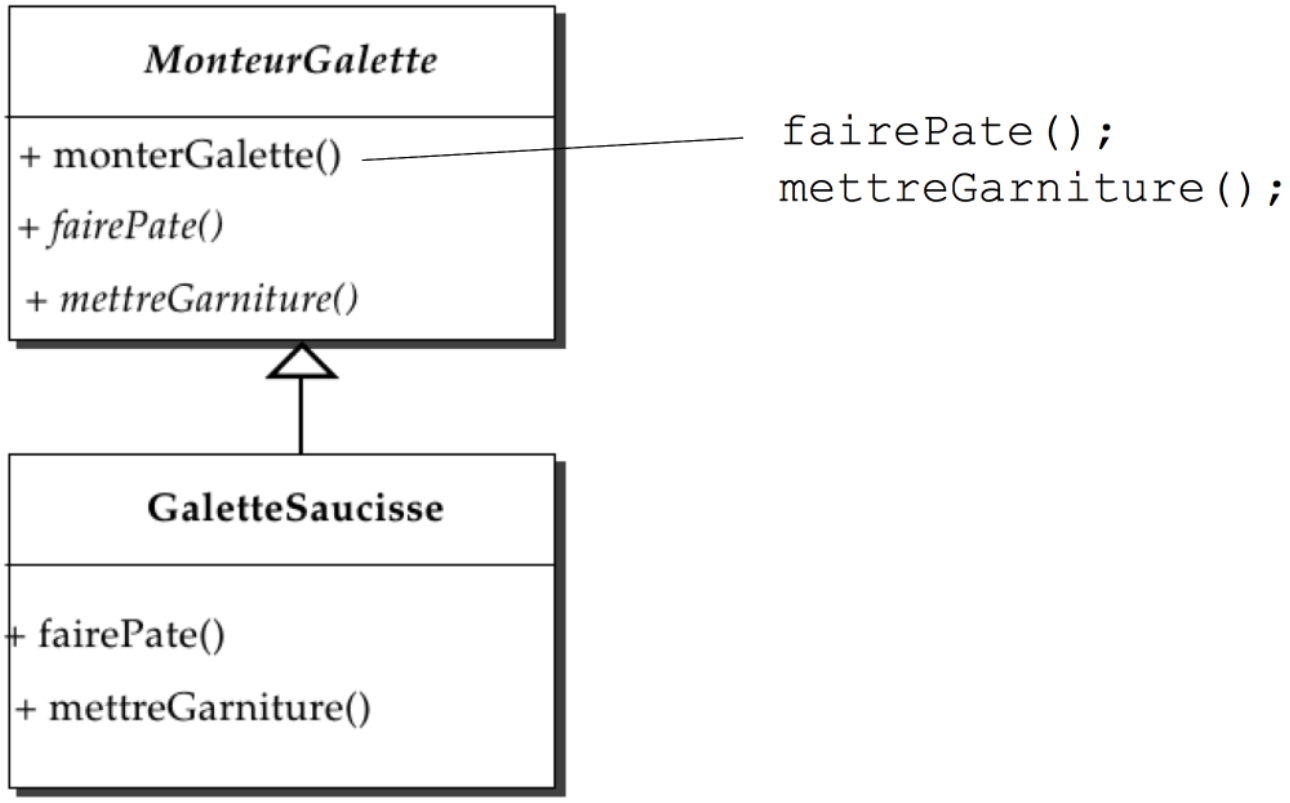
Patron de Conception
Monteur / Builder *(création)*

- Problème :
  - La construction de certains objets peut être complexe
  - → Mettre cette construction dans la classe concernée l'alourdirait

- Exemples :
  - Le montage d'un ordinateur
    - → Plusieurs configurations possibles
  - Création de grilles de Sudoku
    - → Plusieurs niveaux de difficulté possibles

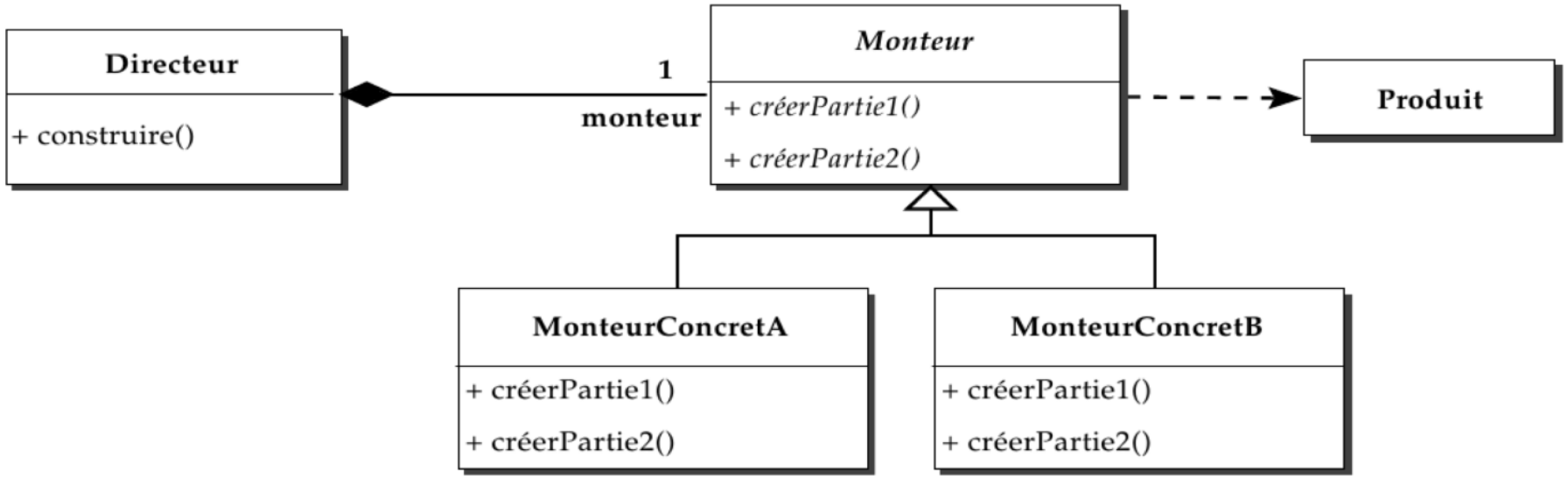
Patron de Conception
Patron de méthode /
Template method *(comportement)*

- Exemple : la création de galettes



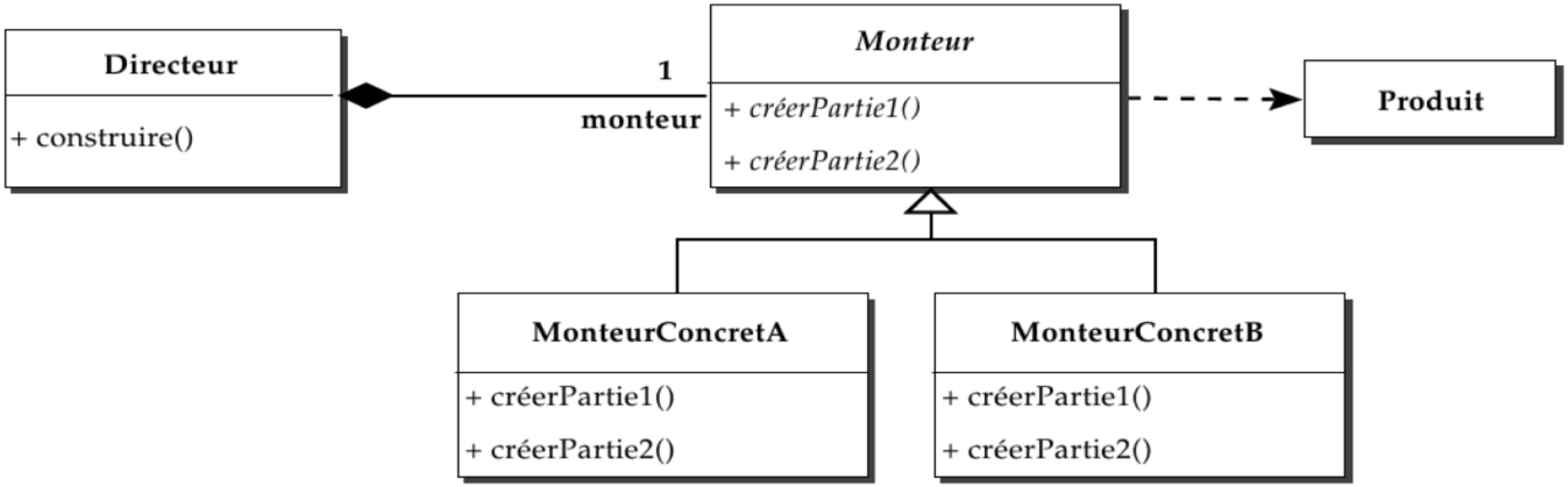
Patron de Conception
Monteur / Builder *(création)*

- But :
  - Séparer la création d'objets complexes de leur représentation
  - Le même processus de création peut servir à créer différents objets complexes



Utilise très souvent le patron *Patron de méthode*

Patron de Conception
Monteur / Builder *(création)*



- **Directeur** : définit le processus de création
- **Monteur** : interface pour l'ajout de parties dans le **Produit**
- **MonteurConcret** : différentes implémentations de création

Patron de Conception
Monteur / Builder *(création)*

Exemple d'un monteur dans JavaFX

Tiré de : http://blog.netopyr.com/2012/01/24/advantages-of-javafx-builders/

(JavaFX est la librairie graphique de Java, en remplacement de Java Swing)

Code Java pour créer et paramétrer des widgets de manière classique (sans *Monteur*) :

```
Text text1 = new Text(50, 50, "Hello World!");
text1.setFill(Color.WHITE);
text1.setFont(MY_DEFAULT_FONT);

Text text2 = new Text(50, 100, "Goodbye World!");
text2.setFill(Color.WHITE);
text2.setFont(MY_DEFAULT_FONT);

Text text3 = new Text(50, 150, "JavaFX is fun!");
text3.setFill(Color.WHITE);
text3.setFont(MY_DEFAULT_FONT);
```

- Cela fonctionne mais verbeux
- C'est un inconvénient de la programmation impérative

Patron de Conception
Monteur / Builder *(création)*

Exemple d'un monteur dans JavaFX

Code Java pour créer et paramétrer des widgets avec un **monteur** fournit par JavaFX :

```
Text text1 = TextBuilder.create().text("Hello World!").x(50).y(50)
                .fill(Color.WHITE).font(MY_DEFAULT_FONT).build();

Text text2 = TextBuilder.create().text("Goodbye World!").x(50).y(100)
                .fill(Color.WHITE).font(MY_DEFAULT_FONT).build();

Text text3 = TextBuilder.create().text("JavaFX is fun!").x(50).y(150)
                .fill(Color.WHITE).font(MY_DEFAULT_FONT).build();
```

- *create()* créer un monteur qui est ensuite paramétré
- *build()* construit ensuite l'instance de *Text*
- Cette manière de faire s'inspire de la **programmation fonctionnelle** :
  - enchainement des appels de fonctions sur un même objet
  - plus clair (en général), limite les effets de bord, facilite la parallélisation

Patron de Conception
Monteur / Builder *(création)*

Exemple d'un monteur dans JavaFX

Javadoc du monteur *TextBuilder* :

http://docs.oracle.com/javafx/2/api/javafx/scene/text/TextBuilder.html

Text build() : Make an instance of Text based on the properties set on this builder.

Static TextBuilder<?> create() : Creates a new instance of TextBuilder.

TextBuilder<?> font(Font x) : Set the value of the font property for the instance constructed by this builder.

```
public Text build() {
    Text x = new Text();
    applyTo(x); // Configure l'instance
    return x;
}
```

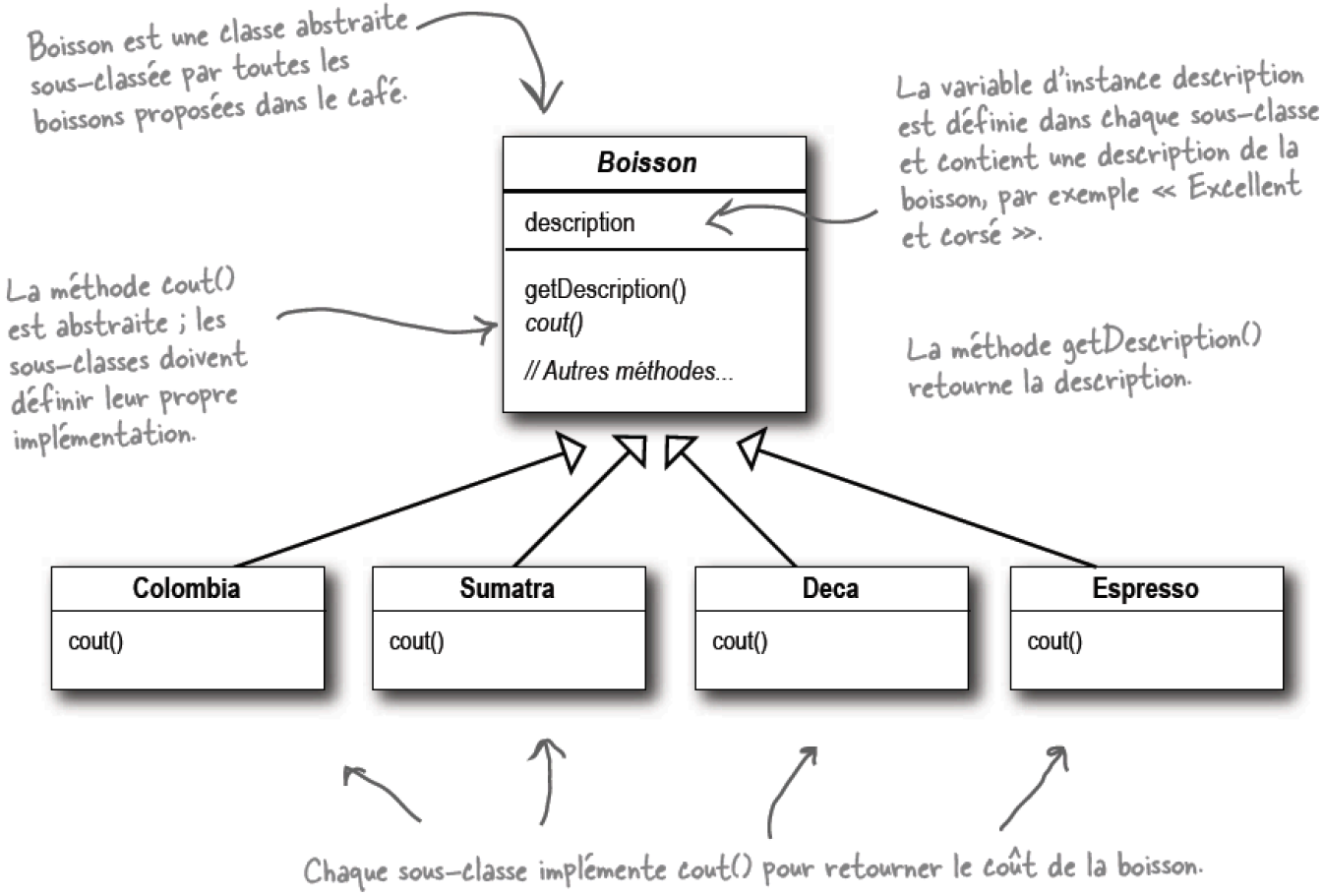
```
public static TextBuilder create() {
    return new TextBuilder();
}
```

```
public TextBuilder font(Font x) {
    font = x;
    return this;
}
```

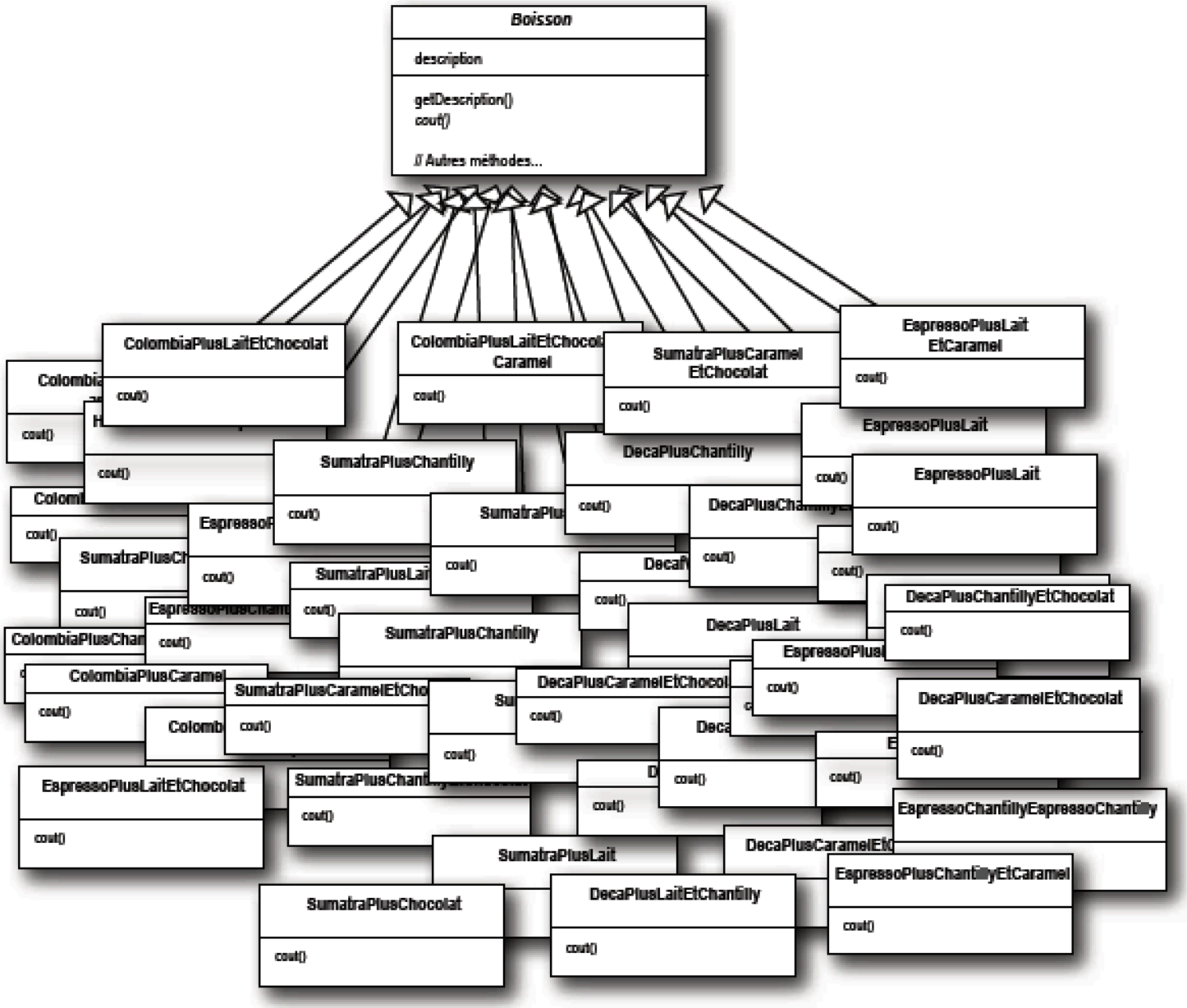
Les monteurs de JavaFX sont désormais «deprecated» (à ne plus utiliser) pour diverses raisons obscures : http://mail.openjdk.java.net/pipermail/openjfx-dev/2013-March/006725.html

Un café?

Decorator


Boisson est une classe abstraite sous-classée par toutes les boissons proposées dans le café.
La variable d'instance description est définie dans chaque sous-classe et contient une description de la boisson, par exemple « Excellent et corsé ».
La méthode cout() est abstraite ; les sous-classes doivent définir leur propre implémentation.
La méthode getDescription() retourne la description.
Boisson
description
getDescription()
cout()
// Autres méthodes...
Colombia
cout()
Sumatra
cout()
Deca
cout()
Espresso
cout()
Chaque sous-classe implémente cout() pour retourner le coût de la boisson.

#1 Héritage

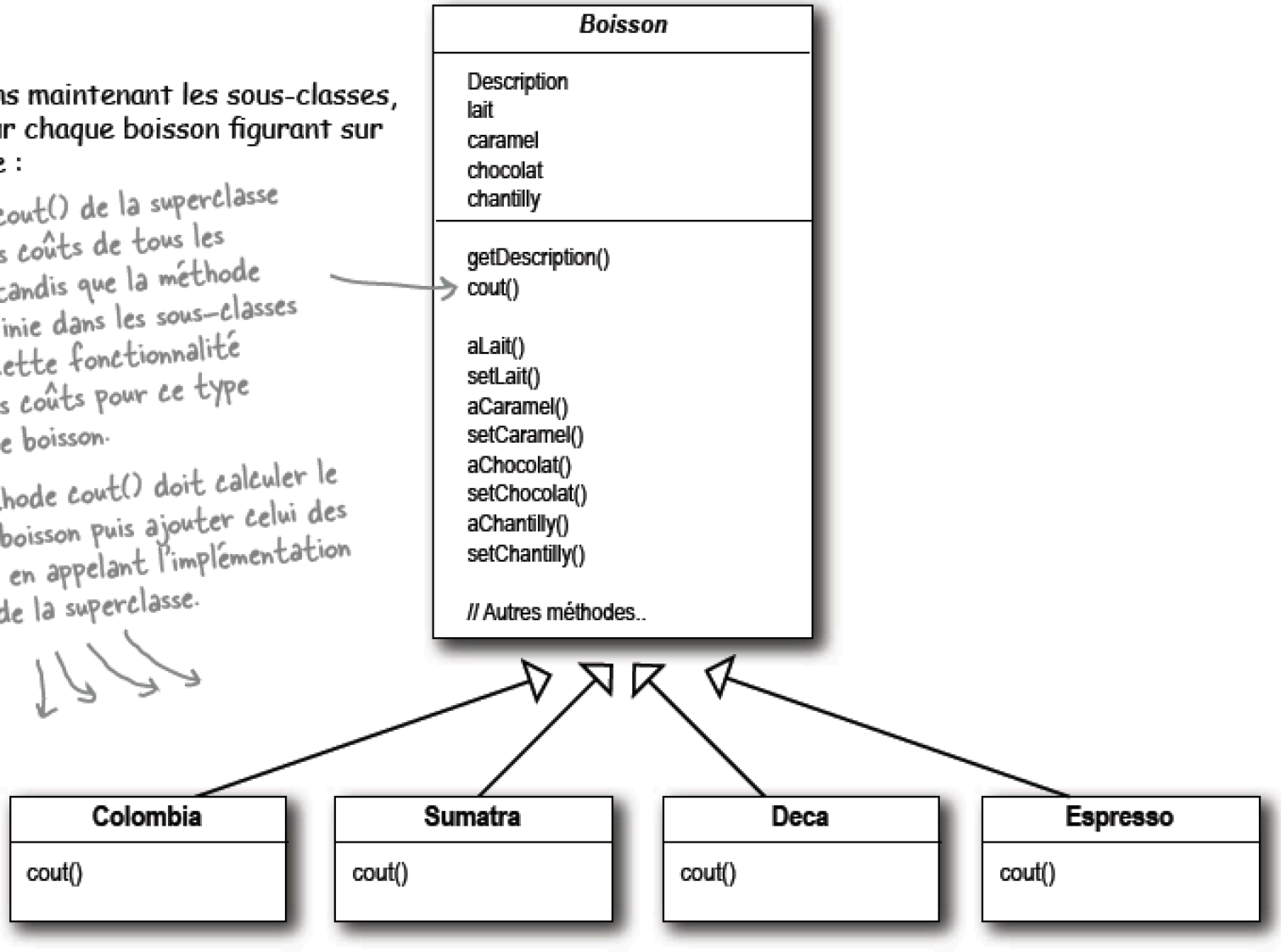

Boisson
description
getDescription()
cout()
// Autres méthodes...
ColombiaPlusLaitEtChocolat
cout()
SumatraPlusCaramel EtChocolat
cout()
EspressoPlusLait EtCaramel
cout()
EspressoPlusLait
cout()
SumatraPlusChantilly
cout()
DecaPlusChantilly
cout()
EspressoPlusLait
cout()
DecaPlusChantillyEtChocolat
cout()
DecaPlusLait
cout()
DecaPlusCaramelEtChocolat
cout()
EspressoPlusLaitEtChocolat
cout()
SumatraPlusLait
cout()
SumatraPlusChocolat
cout()
DecaPlusLaitEtChantilly
cout()
EspressoChantillyEspressoChantilly
EspressoPlusChantillyEtCaramel
cout()

#2 Une autre solution

Ajoutons maintenant les sous-classes, une pour chaque boisson figurant sur la carte :

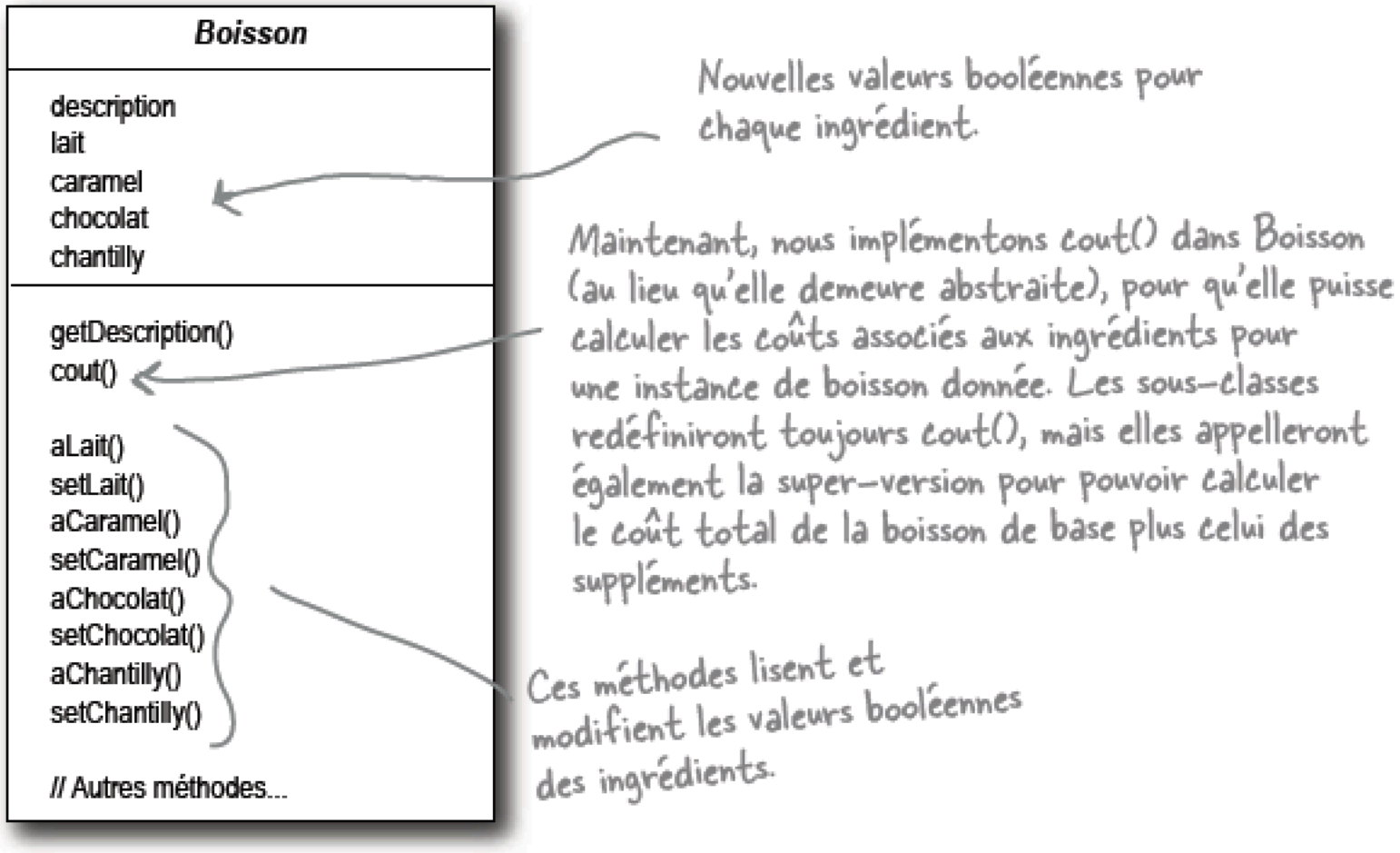
La méthode cout() de la superclasse va calculer les coûts de tous les ingrédients, tandis que la méthode cout() redéfinie dans les sous-classes va étendre cette fonctionnalité et inclure les coûts pour ce type spécifique de boisson.

Chaque méthode cout() doit calculer le coût de la boisson puis ajouter celui des ingrédients en appelant l'implémentation de cout() de la superclasse.

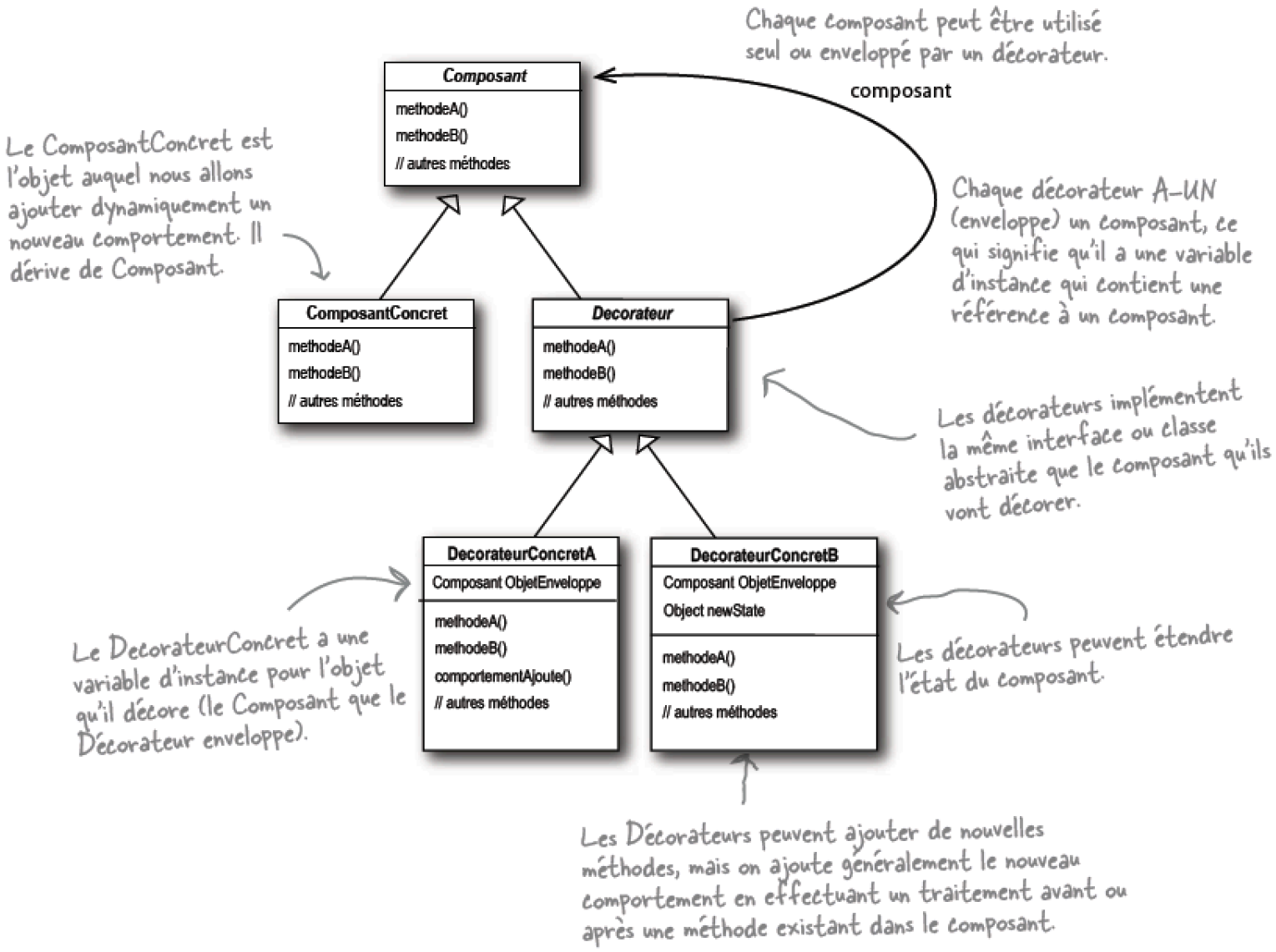


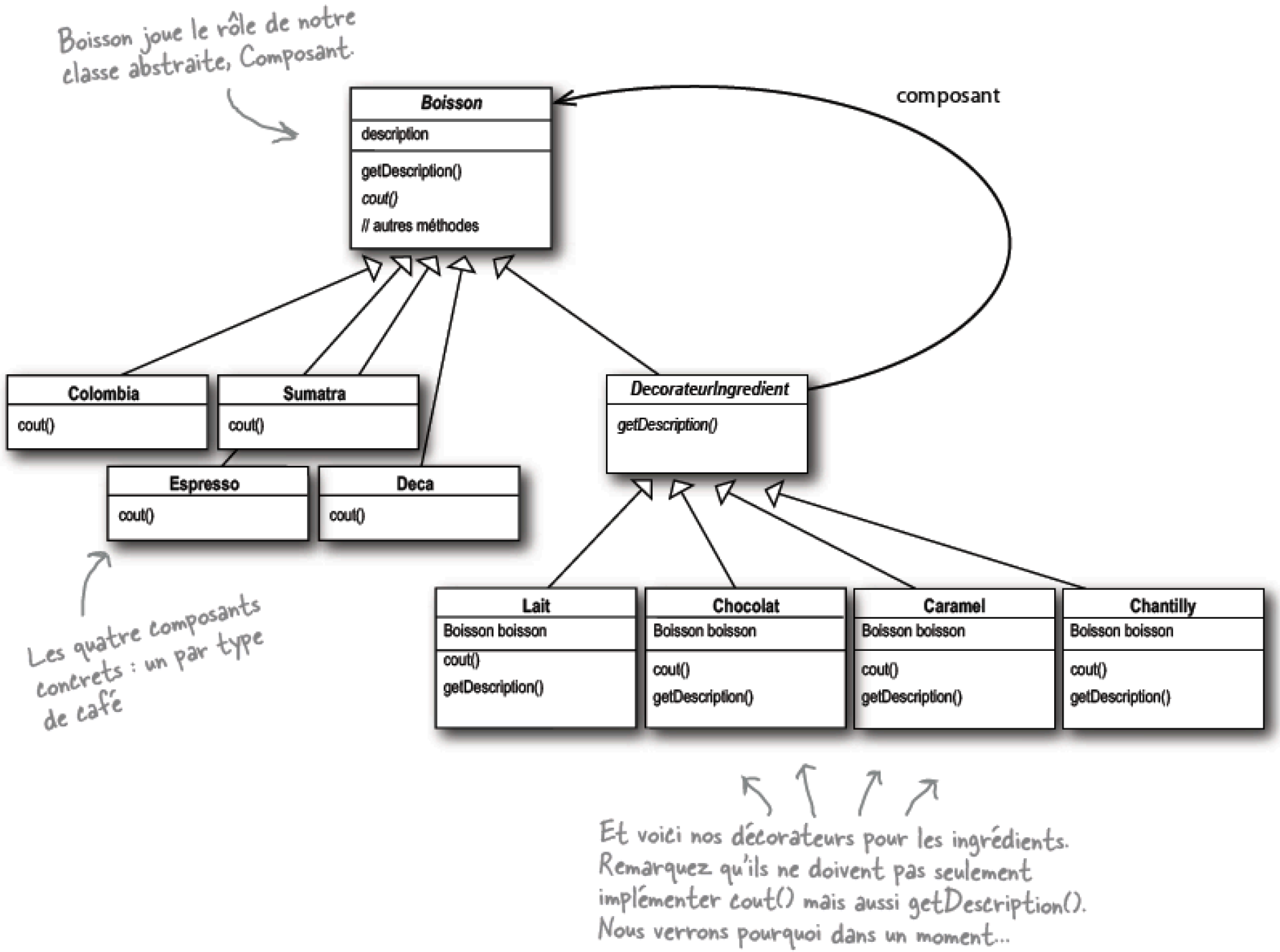
Augmentation du prix des ingrédients oblige à **modifier le code existant.**
Quid des nouveaux ingrédients?
Quid des nouvelles boissons?

Les classes doivent être ouvertes à l'extension, mais fermées à la modification.



Le pattern **Décorateur** attache dynamiquement des responsabilités/fonctionnalités supplémentaires à un objet. Il fournit une alternative souple à l'héritage, pour étendre les fonctionnalités.




Boisson joue le rôle de notre classe abstraite, Composant.
composant
Boisson
description
getDescription()
cout()
// autres méthodes
Colombia
cout()
Sumatra
cout()
Espresso
cout()
Deca
cout()
DecorateurIngredient
getDescription()
Les quatre composants concrets : un par type de café
Lait
Boisson boisson
cout()
getDescription()
Chocolat
Boisson boisson
cout()
getDescription()
Caramel
Boisson boisson
cout()
getDescription()
Chantilly
Boisson boisson
cout()
getDescription()
Et voici nos décorateurs pour les ingrédients. Remarquez qu'ils ne doivent pas seulement implémenter cout() mais aussi getDescription(). Nous verrons pourquoi dans un moment...

```
public abstract class Boisson {
       String description = "Boisson inconnue";

    public String getDescription() {
       return description;
    }

    public abstract double cout();
}
```

Boisson est une classe abstraite qui possède deux méthodes : getDescription() et cout().

getDescription a déjà été implémentée pour nous, mais nous devons implémenter cout() dans les sous-classes.

```
public class Colombia extends Boisson {
    public Colombia() {
        description = "Pur Colombia";
    }

    public double cout() {
        return .89;
    }
}
```

```
public abstract class DecorateurIngredient extends Boisson {
    public abstract String getDescription();
}
```

D'abord, comme elle doit être interchangeable avec une Boisson, nous étendons la classe Boisson.

Nous allons aussi faire en sorte que les ingrédients (décorateurs) réimplémentent tous la méthode getDescription(). Nous allons aussi voir cela dans une seconde...

Chocolat est un décorateur : nous étendons DecorateurIngredient.

Souvenez-vous que DecorateurIngredient étend Boisson.

```
public class Chocolat extends DecorateurIngredient {
   Boisson boisson;

   public Chocolat(Boisson boisson) {
      this.boisson = boisson;
   }

   public String getDescription() {
      return boisson.getDescription() + ", Chocolat";
   }

   public double cout() {
      return .20 + boisson.cout();
   }
}
```

Nous allons instancier Chocolat avec une référence à une Boisson en utilisant :

(1) Une variable d'instance pour contenir la boisson que nous enveloppons.

(2) Un moyen pour affecter à cette variable d'instance l'objet que nous enveloppons. Ici, nous allons transmettre la boisson que nous enveloppons au constructeur du décorateur.

La description ne doit pas comprendre seulement la boisson – disons « Sumatra » – mais aussi chaque ingrédient qui décore la boisson, par exemple, « Sumatra, Chocolat ». Nous allons donc déléguer à l'objet que nous décorons pour l'obtenir, puis ajouter « Chocolat » à la fin de cette description.

Nous devons maintenant calculer le coût de notre boisson avec Chocolat. Nous déléguons d'abord l'appel à l'objet que nous décorons pour qu'il calcule son coût. Puis nous ajoutons le coût de Chocolat au résultat.

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Boisson boisson = new Espresso();
        System.out.println(boisson.getDescription()
            + " €" + boisson.cout());

        Boisson boisson2 = new Sumatra();
        boisson2 = new Chocolat(boisson2);
        boisson2 = new Chocolat(boisson2);
        boisson2 = new Chantilly(boisson2);
        System.out.println(boisson2.getDescription()
            + " €" + boisson2.cout());

        Boisson boisson3 = new Colombia();
        boisson3 = new Caramel(boisson3);
        boisson3 = new Chocolat(boisson3);
        boisson3 = new Chantilly(boisson3);
        System.out.println(boisson3.getDescription()
            + " €" + boisson3.cout());
    }
}
```

Commander un espresso, pas d'ingrédients et afficher sa description et son coût.

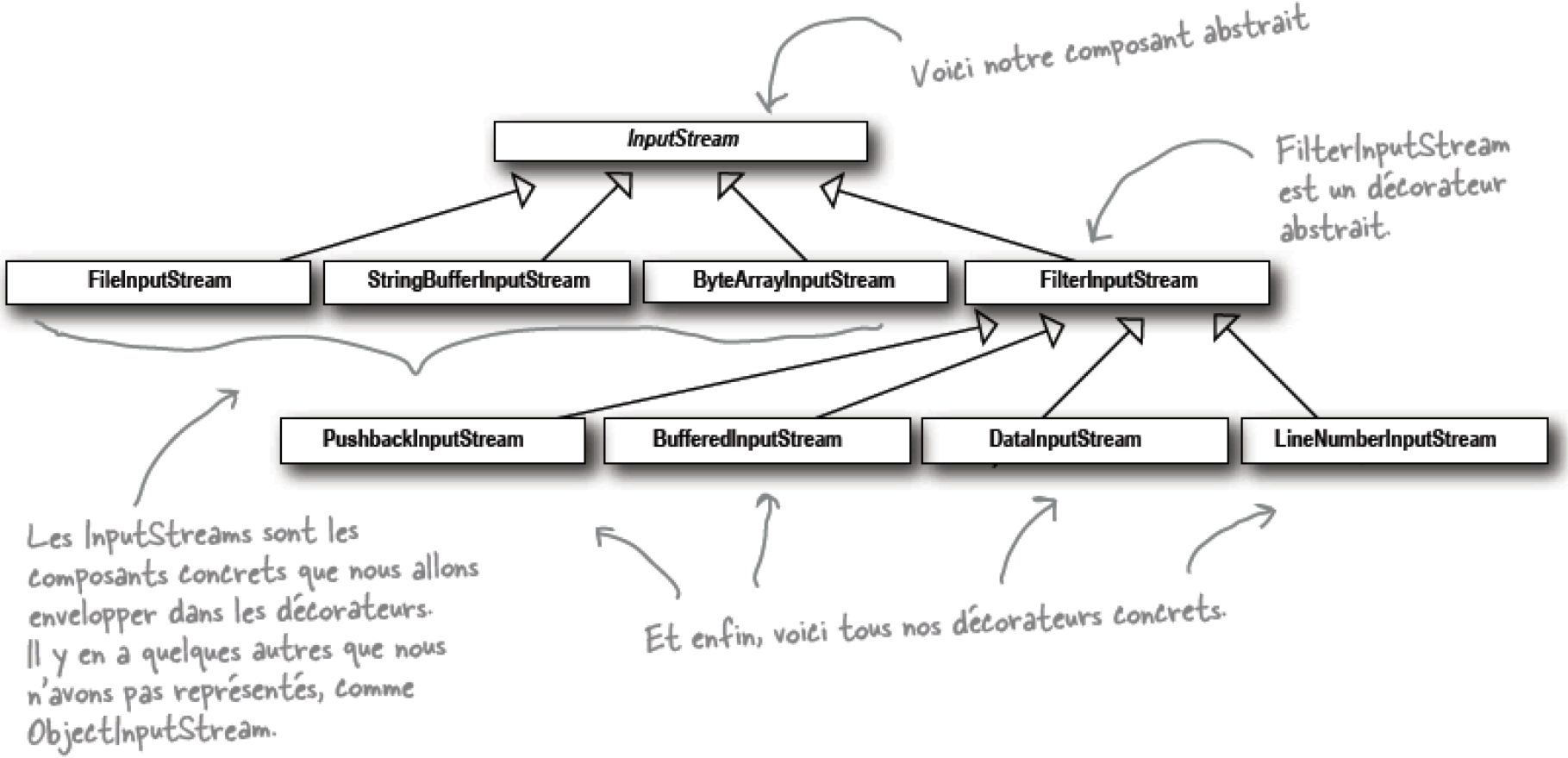
Créer un objet Sumatra.

L'envelopper dans un Chocolat.

L'envelopper dans un second Chocolat.

L'envelopper de Chantilly.

Enfin nous servir un Colombia avec Caramel, Chocolat et Chantilly.


Voici notre composant abstrait
InputStream
FilterInputStream est un décorateur abstrait.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
Les InputStreams sont les composants concrets que nous allons envelopper dans les décorateurs. Il y en a quelques autres que nous n'avons pas représentés, comme ObjectInputStream.
Et enfin, voici tous nos décorateurs concrets.

Adapter versus Decorator ?

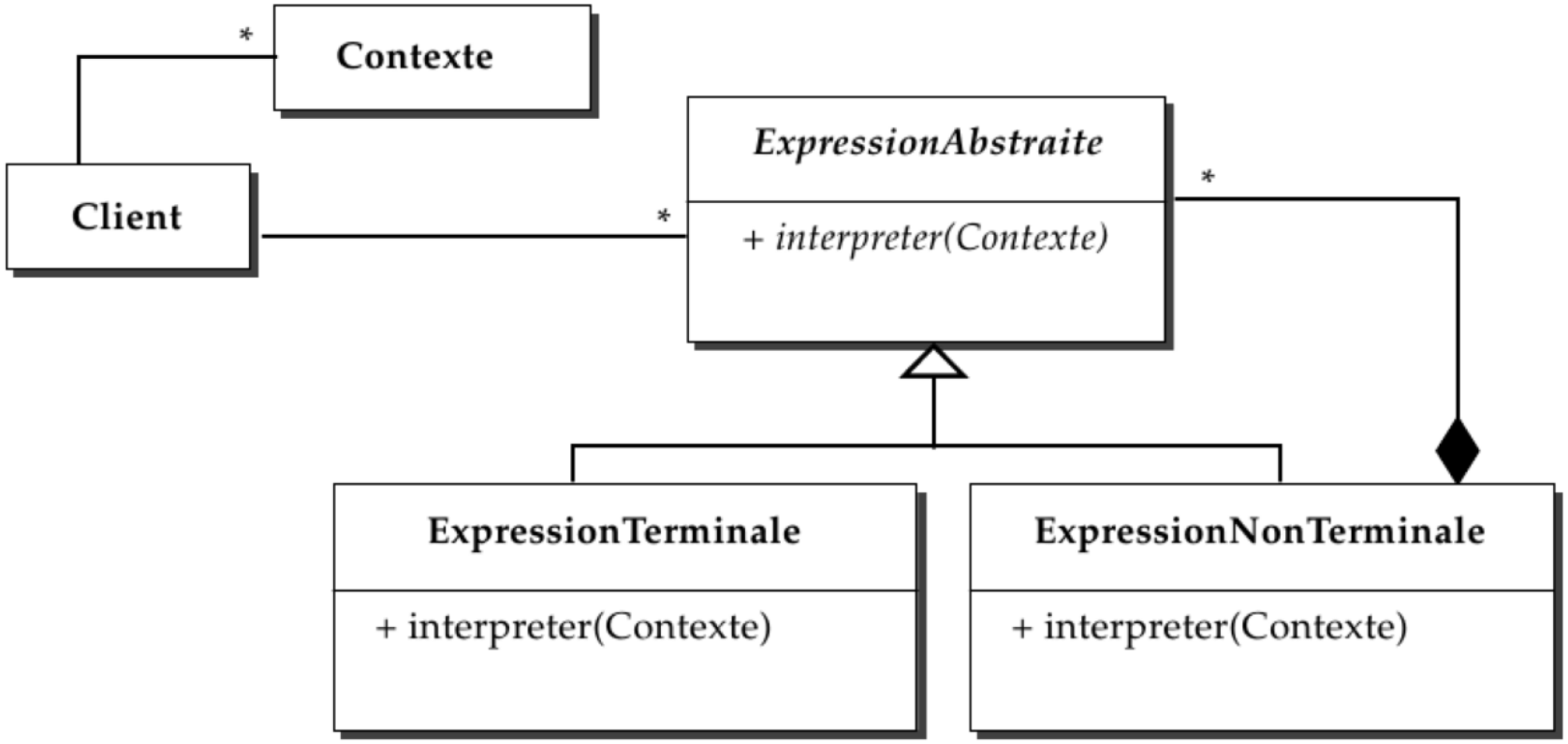
Décorateur = Ajout de comportements aux opérations existantes. Interface non modifiée.

Adaptateur = Ajout d'interfaces à un objet

Interpreter

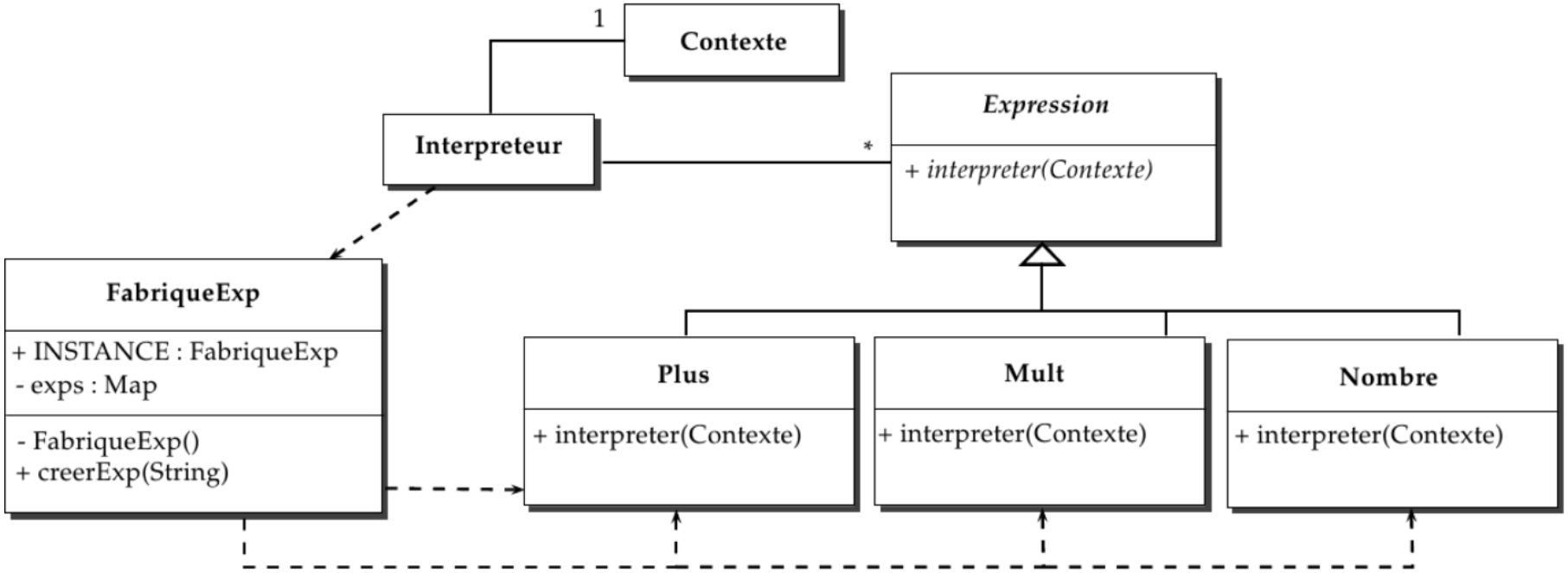
Patron de Conception
Interpréteur / Interpreter *(comportement)*

- But de l'Interpréteur :
  - Étant donné un langage, définir une représentation pour sa grammaire ainsi qu'un interpréteur pour interpréter des séquences du langage



Patron de Conception
Interpréteur / Interpreter *(comportement)*

- Exemple : arithmétique (notation polonaise inversée)



Patron de Conception
Interpréteur / Interpreter *(comportement)*

- Exemple : arithmétique (notation polonaise inversée)

```
class Contexte extends Stack<Double> {

  public Contexte() {
    super();
  }

  public double getFinalValue() {
    if(size()==1)
      return peek();
    return Double.NaN;
  }
}
```

```
interface Expression {
  void interprete(Contexte ctxt);
}
```

```
class Mult implements Expression {
  public void interprete(Contexte ctxt) {
    ctxt.push(ctxt.pop()*ctxt.pop());
  }
}
```

```
class Nombre implements Expression {
  protected double value;

  public Nombre(double val) {
    super();
    value = val;
  }

  public void interprete(Contexte ctxt) {
    ctxt.push(value);
  }
}
```

```
class Plus implements Expression {
  public void interprete(Contexte ctxt) {
    ctxt.push(ctxt.pop()+ctxt.pop());
  }
}
```

Patron de Conception
Interpréteur / Interpreter *(comportement)*

- Exemple : arithmétique (notation polonaise inversée)

```
final class FabriqueExp {
  public static final FabriqueExp INSTANCE = new FabriqueExp();

  private Map<String, Expression> exps;

  private FabriqueExp() {
    super();
    exps = new HashMap<String, Expression>();
  }

  public Expression creerExp(String token) {
    if(token==null) return null;

    Expression exp = exps.get(token);

    if(exp==null) {
      if("+".equals(token))
        exp = new Plus();
      else if("*".equals(token))
        exp = new Mult();
      else
        try {
          exp = new Nombre(Double.valueOf(token));
        }catch(NumberFormatException ex) { return null; }
      exps.put(token, exp);
    }

    return exp;
  }
}
```

```
public class Interpreteur {
  public static void main(String[] args) {
    String txt = "42 4 2 * +";
    Contexte ctxt = new Contexte();

    for(String token : txt.split(" ")) {
      Expression exp = FabriqueExp.
                       INSTANCE.creerExp(token);
        if(exp!=null)
          exp.interprete(ctxt);
    }
    System.out.println(ctxt.getFinalValue());
  }
}
```

Résultat = 50

Poids-mouche + fabrique

A Case of Visitor versus Interpreter Pattern

Mark Hills[1,2], Paul Klint[1,2], Tijs van der Storm[1], and Jurgen Vinju[1,2]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

2.1 Creating and Processing Abstract Syntax Trees

Rascal has many AST classes (about 140 abstract classes and 400 concrete classes). To facilitate language evolution the code for these classes, along with the Rascal parser, is generated from the Rascal grammar. The AST code generator also creates a Visitor interface (`IASTVisitor`), containing methods for all the node types in the hierarchy, and a default visitor that returns null for every node type (`NullASTVisitor`). This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes. Naturally, each AST node implements the `accept(IASTVisitor<T> visitor)` method by calling the appropriate visit method. For example, `Statement.If` contains:

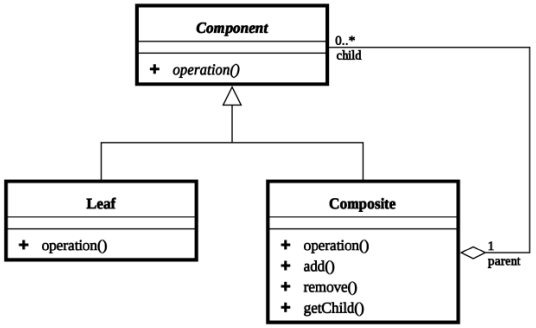


Fig. 2. The Composite Pattern[3]

```
public <T> accept(IASTVisitor<T> v) {
    return v.visitStatementIf(this);
}
```

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern. We wanted to avoid having to manually edit generated code. Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code. When the Rascal grammar changes, the AST hierarchy is regenerated. Many implementations of `IASTVisitor` will contain Java compiler errors and warnings because the signature of visit methods will have changed. This is very helpful for locating the code that needs to be changed due to a language change. Most of the visitor classes actually extend `NullASTVisitor`

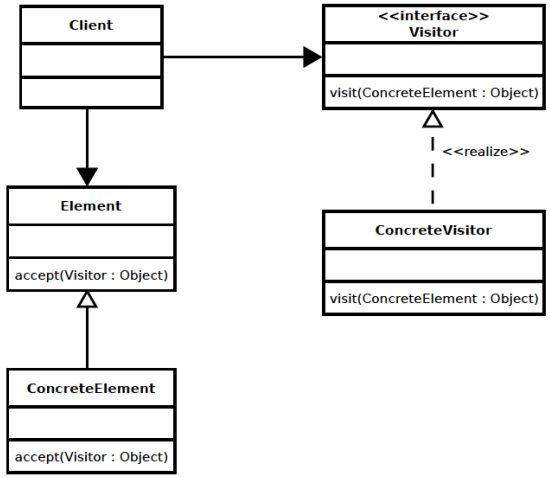


Fig. 3. The Visitor Pattern[4]

2.2 A Comparison with the Interpreter Pattern

Considering that our design already employs the Composite pattern, the difference in design complexity between the Visitor and Interpreter patterns is striking (Figure 4). The Composite pattern contains all the elements for the Interpreter pattern (abstract classes that are instantiated by concrete ones)—only an `interpret` method needs to be added to all relevant classes. So rather than having to add new concepts, such as a `Visitor` interface, the `accept` method and `NullASTVisitor`, the Interpreter pattern builds on the existing infrastructure of Composite and reuses it. Also, by adding more `interpret` methods (varying either the name or the static type) it is possible to reuse the Interpreter design pattern again and again without having to add additional classes. However, as a consequence, understanding each algorithm as a whole is now complicated by the fact that the methods implementing it are scattered over different AST classes. Additionally, there is the risk that methods contributing to different algorithms get tangled because a single AST class may have to manage the combined state required for all implemented algorithms. The experiments discussed in Section 4 help make this tradeoff between separation of concerns and complexity more concrete.

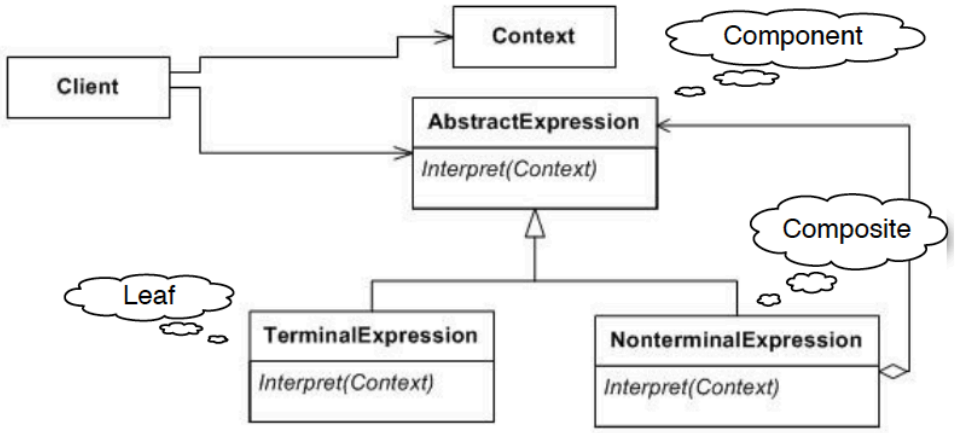


Fig. 4. The Interpreter Pattern with references to Composite (Figure 2).[7]

Et dans le futur ? *(from E. Gamma)*

a new categorization

- **Core**
  - Composite
  - Strategy
  - State
  - Command
  - Iterator
  - Proxy
  - Template Method
  - Facade
  - ***Null Object***

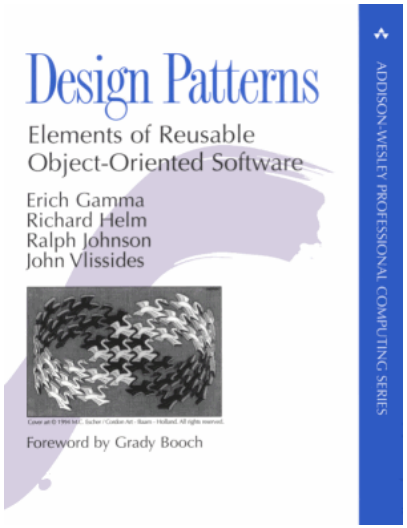
- **Creational**
  - Factory method
  - Prototype
  - Builder
  - ***Dependency Injection***
- **Peripheral**
  - Abstract Factory (peripheral)
  - Memento
  - Chain of responsibility
  - Bridge
  - Visitor
  - ***Type Object***
  - Decorator
  - Mediator
  - Singleton
  - ***Extension Objects***
- **Other** (Compound)
  - Interpreter
  - Flyweight

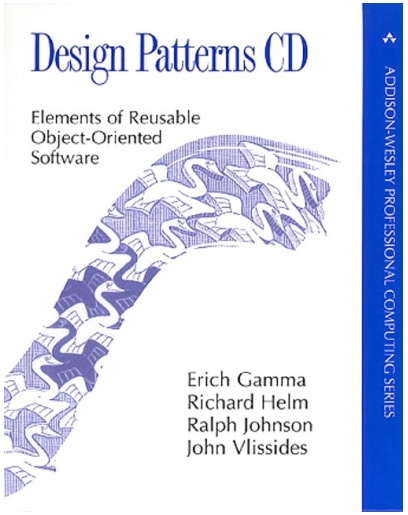
the patterns the students should learn

lean on demand

- Composite, State, Strategy, Command, Observer
- Template Method, Singleton, Facade, Abstract Factory, Visitor, Memento, Adapter
- Decorator
- Builder, Flyweight, Interpreter
- Iterator

References


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


Design Patterns CD
Elements of Reusable
Object-Oriented
Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


A Brain-Friendly Guide
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

References

http://refcardz.dzone.com/refcardz/design-patterns

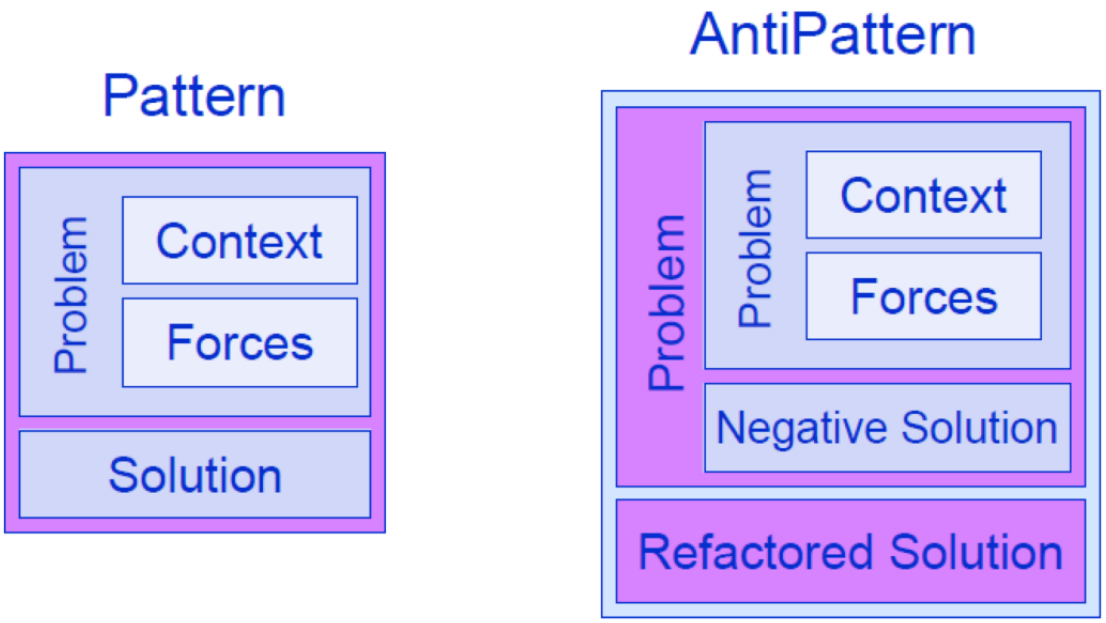
En génie logiciel

un patron ≠ de

≠ d'un anti-patron

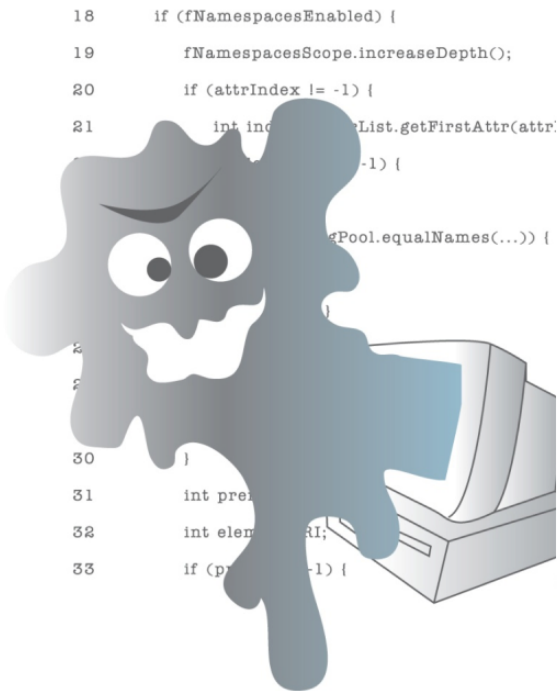
Défauts de conception

- Un **anti-patron** est un type spécial de patron de conception caractérisé par une solution refactorisée



Défauts de conception

- 2 exemples d'anti-patrons
  - Blob *(God Class)*



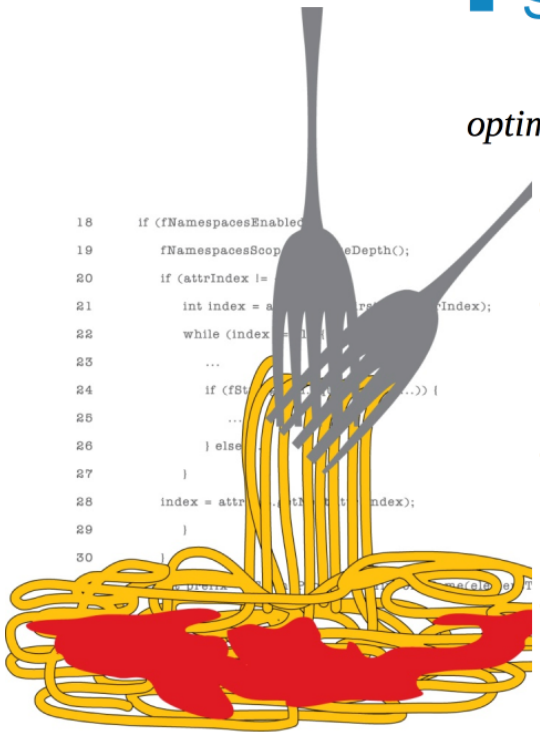
" Procedural-style design leads to one object with a lion's share of the responsibilities while most other objects only hold data or execute simple processes "

- Conception procédurale en programmation OO
- Large classe contrôleur
- Beaucoup d'attributs et méthodes avec une faible cohésion*
- Dépend de classes de données

** À quel point les méthodes sont étroitement liées aux attributs et aux méthodes de la classe.*

Défauts de conception

- 2 exemples d'anti-patrons



- Spaghetti Code

" Ad hoc software structure makes it difficult to extend and optimize code. "

  - Conception procédurale en programmation OO
  - Manque de structure : pas d'héritage, pas de réutilisation, pas de polymorphisme
  - Noms des classes suggèrent une programmation procédurale
  - Longues méthodes sans paramètres avec une faible cohésion
  - Utilisation excessive de variables globales

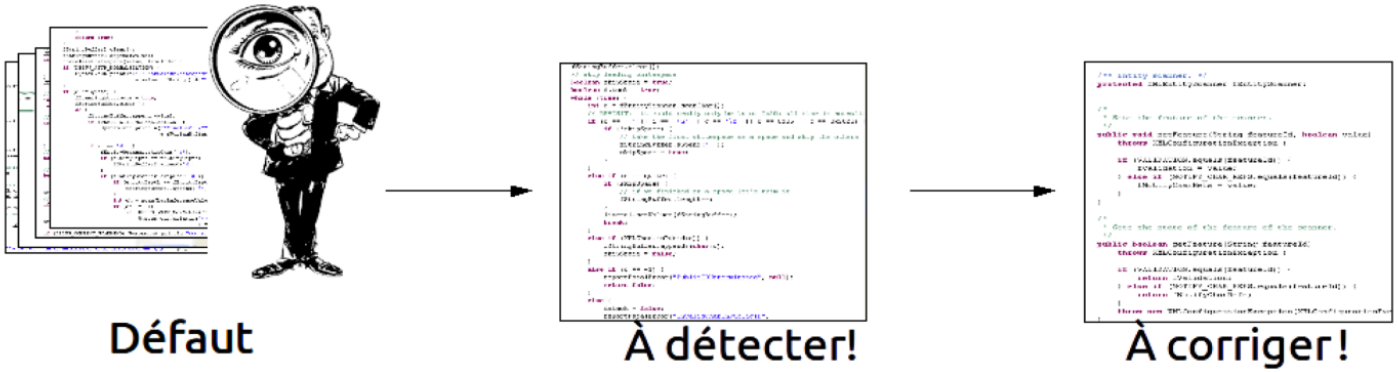
```
public final class Frame extends JFrame implements ActionListener, ItemListener, ChangeListener, WindowStateListener{
private static final long serialVersionUID = 1L;
public final static String VERSION   = "2.0.8";//$NON-NLS-1$
public final static String VERSION_STABILITY = ""; //$NON-NLS-1$
public static final boolean WITH_UPDATE = true;
public static final Insets INSET_BUTTON = new Insets(1,1,1,1);
public static final String DEFAULT_PATH = System.getProperty("user.home");//$NON-NLS-1$
protected ShortcutsFrame shortcutsFrame;
protected PSTProgressBarManager progressbarPST = null;
protected SVGProgressBarManager progressbarSVG = null;
protected transient MenusListener menusListener;
protected transient RecentFilesListener recentFilesListener;
private String lateXIncludes = null;
private String pathDistribLatex = null;
private String pathTexEditor = null;
private InsertPSTricksCodeFrame insertCodeFrame;
protected CodePanel codePanel;
protected DrawPanel drawPanel;
protected LToolbar toolbar;
protected JProgressBar progressBar;
protected JButton stopButton;
protected LMenuBar menuBar;
private static final String ID_BUILD = "20100314";//$NON-NLS-1$
protected transient UndoRedoManager undoManager;
private ParametersLineFrame paramLineFrame;
private ParametersAxeFrame paramAxesFrame;
private ParametersCircleSquareFrame paramCircleFrame;
private ParametersEllipseRectangleFrame paramEllipseFrame;
private ParametersBezierCurveFrame paramBezierCurveFrame;
private ParametersAkinPointsFrame paramAkinPointsFrame;
private ParametersDotFrame paramDotFrame;
```

Tout le contraire d'un code monolithique (une seule classe)

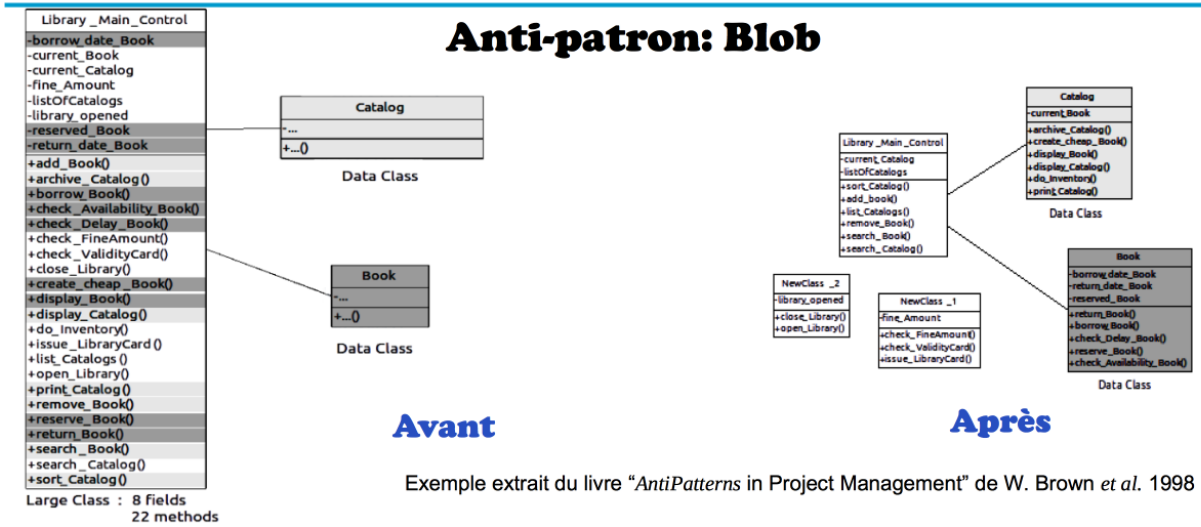
Version graphique?
Sauvegarder/Charger une partie?
Comment s'assurer que la règle du pat a bien été prise en compte?
Nouveau moteur de AI?

- ChessDay1
  - main(String[]) : void
  - board : PieceType[][]
  - ChessDay1()
  - canMove(int, int, int, int) : boolean
  - displayBoard() : void
  - get50MoveRulePlyCount() : int
  - getCapturedPieces() : List<PieceType>
  - getCapturedPieces(boolean) : List<PieceType>
  - getCurrentMoveNumber() : int
  - getHistory() : String
  - getMaterialCount(boolean) : int
  - getPossibleMoves() : List<int[][]>
  - getPossibleSquares(int, int) : List<int[][]>
  - getThreats(int, int) : List<int[][]>
  - getUnCapturedPieces(boolean) : List<PieceType>
  - initBoard() : void
  - isBlackCastleableKingside() : boolean
  - isBlackCastleableQueenside() : boolean
  - isBlockable() : boolean
  - isCheck() : boolean
  - isCheckmate() : boolean
  - isDoubleCheck() : boolean
  - isEnPassantFile(int) : boolean
  - isStalemate() : boolean
  - isWhiteCastleableKingside() : boolean
  - isWhiteCastleableQueenside() : boolean
  - isWhiteToPlay() : boolean
  - recordMove(int, int, int, int) : boolean
  - redo() : boolean
  - reset() : void
  - setBlackCastleableKingside(boolean) : void
  - setBlackCastleableQueenside(boolean) : void
  - setWhiteCastleableKingside(boolean) : void
  - setWhiteCastleableQueenside(boolean) : void
  - undo() : boolean

Défauts de conception



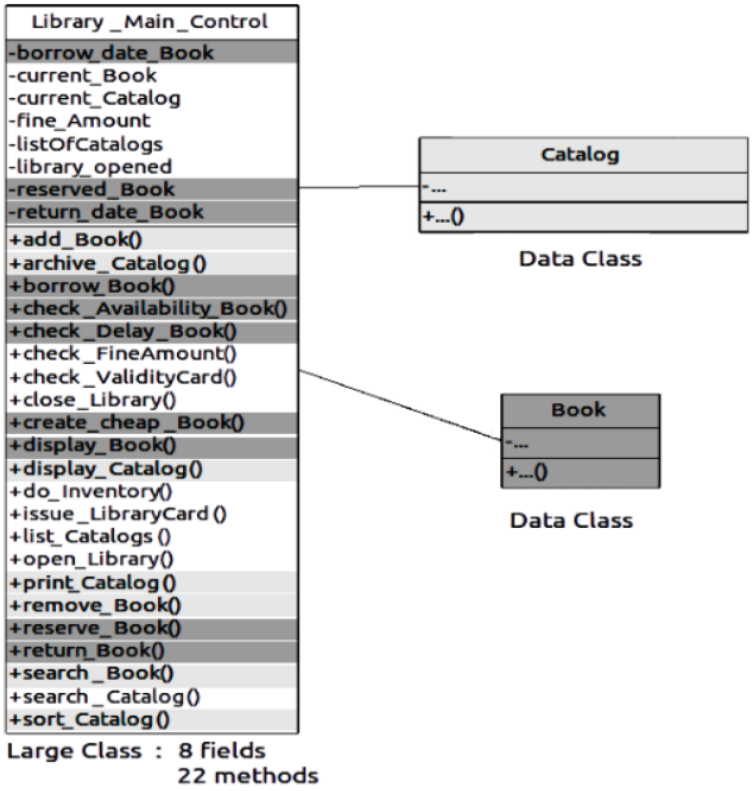
Anti-patron: Blob



Exemple extrait du livre "*AntiPatterns* in Project Management" de W. Brown *et al.* 1998

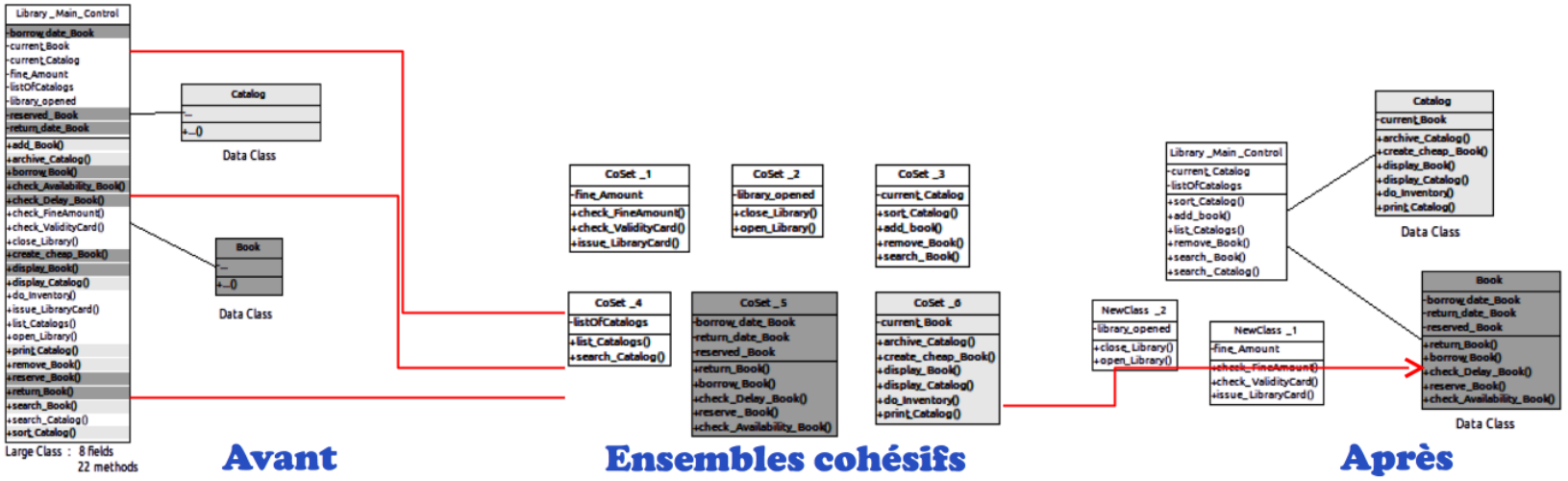
Détecter un anti-patron

- Blob
  - Identifier les larges classes
  - Identifier les classes de données



Corriger un anti-patron

- Blob
  - Identifier ou catégoriser les attributs et opérations liées
  - Rechercher des classes candidates pour les accueillir
  - Appliquer des techniques de conception objet (héritage, délégation, patrons, *etc.*)



Mauvaises odeurs

- Longue méthode
- Large classe
- Longue liste de paramètres
- Primitive obsession
- Groupe de données (Data clumps)
- Instructions Switch
- Champ temporaire
- Héritage refusé
- Classes alternatives avec des interfaces différentes
- Hiérarchies parallèles d'héritage
- Classe paresseuse
- Classe de données
- Code dupliqué
- Généralité spéculative
- Chaine de messages
- Middle man
- Feature envy
- Divergent change
- Shotgun surgery
- Classe de librairie incomplète
- Commentaires

- Code dupliqué
  - Structure de code dupliqué à différents endroits
- Longue méthode
  - À décomposer pour viser la clarté et facilité de maintenance
- Large classe
  - Classes qui essaient d'en faire trop. Présence de code dupliqué
- Longue liste de paramètres
  - Passer à la méthode juste ce dont elle a besoin
- Commentaires

- Divergent Change
  - Si une classe est modifiée de différentes manières pour différentes raisons, ça vaut la peine de diviser la classe de sorte que chaque partie soit associée à un type de changement particulier.
- Shotgun Surgery
  - Si un type de changement nécessite plusieurs petits changement de code dans différentes classes, tous ces bouts de code qui sont affectés devraient être mis ensemble dans une classe.
- Feature Envy
  - Une méthode d'une classe est plus intéressée par les attributs d'une autre classes que celles de sa propre classe. Peut-être que placer la méthode dans cette autre classe serait plus appropriée.

```
public class A {
 public void fooA() {
 }
```

```
public class B {
 A a = new A();
 public void foobarB() {
```

- Primitive Obsession
  - Parfois, plus intéressant de déplacer un type de données primitives vers une classe légère pour le rendre explicite et identifier les opérations à réaliser (ex : créer une classe date plutôt qu'utiliser un couple d'entiers).
- Instructions Switch
  - Tendent à créer de la duplication. Plusieurs instructions switch éparpillées à différents endroits. Utiliser des classes et du polymorphisme.
- Hiérarchies parallèles d'héritage
  - Deux hiérarchies parallèles existent et un changement dans une classe de la hiérarchie nécessite des changements dans l'autre hiérarchie.

```
public class B extends A {
}
```

```
public enum AEnum {
 B,
```

Trouver le défaut

```
class OwnershipTest...
   private void createUserInGroup() {
      GroupManager groupManager = new GroupManager();

      Group group = groupManager.create(TEST_GROUP, false,
                    GroupProfile.UNLIMITED_LICENSES, "",
                       GroupProfile.ONE_YEAR, null);

      user = userManager.create(USER_NAME, group, USER_NAME,
             "joshua", USER_NAME,LANGUAGE, false, false,
             new Date(), "blah", new Date());
}
```

Longue liste de paramètres

Trouver le défaut

```
public class Phone {
  public String getAreaCode() {
     return 1 ;
  }
  public String getPrefix() {
     return 21 ;
  }
  public String getNumber() {
     return 1234 ;
  }
}

public class Customer {
  private Phone phone;

  public String getPhoneNumber() {
    return "(" + phone.getAreaCode() + ") "
        + phone.getPrefix() + "-" + phone.getNumber();
    }
}
```

Trouver le défaut

```
public class Phone {
  public String getAreaCode() {
     return 1 ;
  }
  public String getPrefix() {
     return 21 ;
  }
  public String getNumber() {
     return 1234 ;
  }
}
```

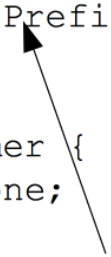
Feature Envy

Customer va rechercher dans les données de Phone
getPhoneNumber devrait être La class Phone.

```
public class Customer {
  private Phone phone;

  public String getPhoneNumber() {
    return "(" + phone.getAreaCode() + ") "
        + phone.getPrefix() + "-" + phone.getNumber();
    }
}
```

Trouver le défaut

```
public class Phone {
  public String getAreaCode() {
     return 1 ;
  }
  public String getPrefix() {
     return 21 ;
  }
  public String getNumber() {
     return 1234 ;
  }

  public String toString() {
    return "(" + phone.getAreaCode() + ") "
        + phone.getPrefix() + "-" + phone.getNumber();
}

public class Customer {
  private Phone phone;

  public String getPhoneNumber() {
    return phone ;
  }
}
```



Correction

Customer compte sur Phone pour faire le formatage

```
public abstract class AbstractCollection implements collection
   public void addAll(AbstractCollection c) {
      if(c instanceof Set) {
           Set s = (Set)c;
           for(int i=0; i<s.size();i++)
                if(!contains(s.get(i)))
                     add(s.get(i));
      }
      else if(c instanceof List) {
             List l = (List)c;
             for(int i=0;i<l.size();i++)
                    if(!contains(l.get(i)))
                           add(l.get(i));
   }
```

Trouver le défaut

Instruction Switch

```
public abstract class AbstractCollection implements collection
   public void addAll(AbstractCollection c) {
      if(c instanceof Set) {
           Set s = (Set)c;
           for(int i=0; i<s.size();i++)
                if(!contains(s.get(i)))
                     add(s.get(i));
      }
      else if(c instanceof List){
             List l = (List)c;
             for(int i=0;i<l.size();i++)
                    if(!contains(l.get(i)))
                           add(l.get(i));
   }
```

Classes alternatives avec interfaces différentes

Code dupliqué

Trouver le défaut

```
public abstract class AbstractCollection  {
    public void add(Object element) {
    }
}
```

```
public class Map extends AbstractCollection {
    // Do nothing because user must input key and value
    public void add(Object element) {
    }
}
```

Outils et Méthodes

Software Engineering