Les patrons GoF comportementaux

Observateur – introduction

- Plusieurs objets (**observateurs**) ont besoin de savoir quand un autre objet (**observé**) ***change*** pour réaliser des actions spécifiques
  - Actions potentiellement basées sur l'exploitation des **propriétés** de l'objet observé

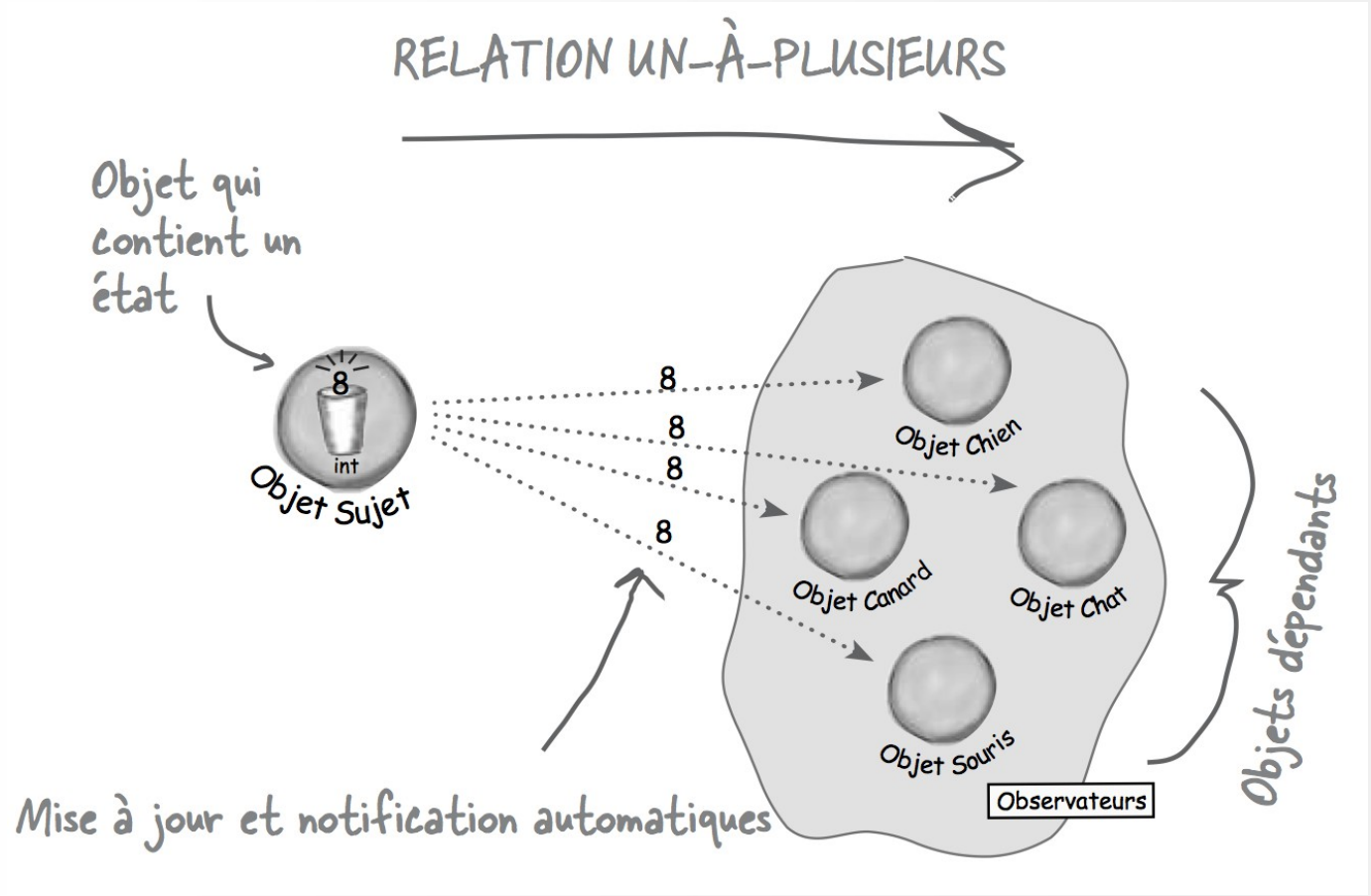
- Comment s'y prendre ? (en évitant la scrutation/ attente active !)
  - Inspirez vous des solutions bas niveau utilisées dans un système d'exploitation...

TDm dédié à ce patron !

Observateur - définition

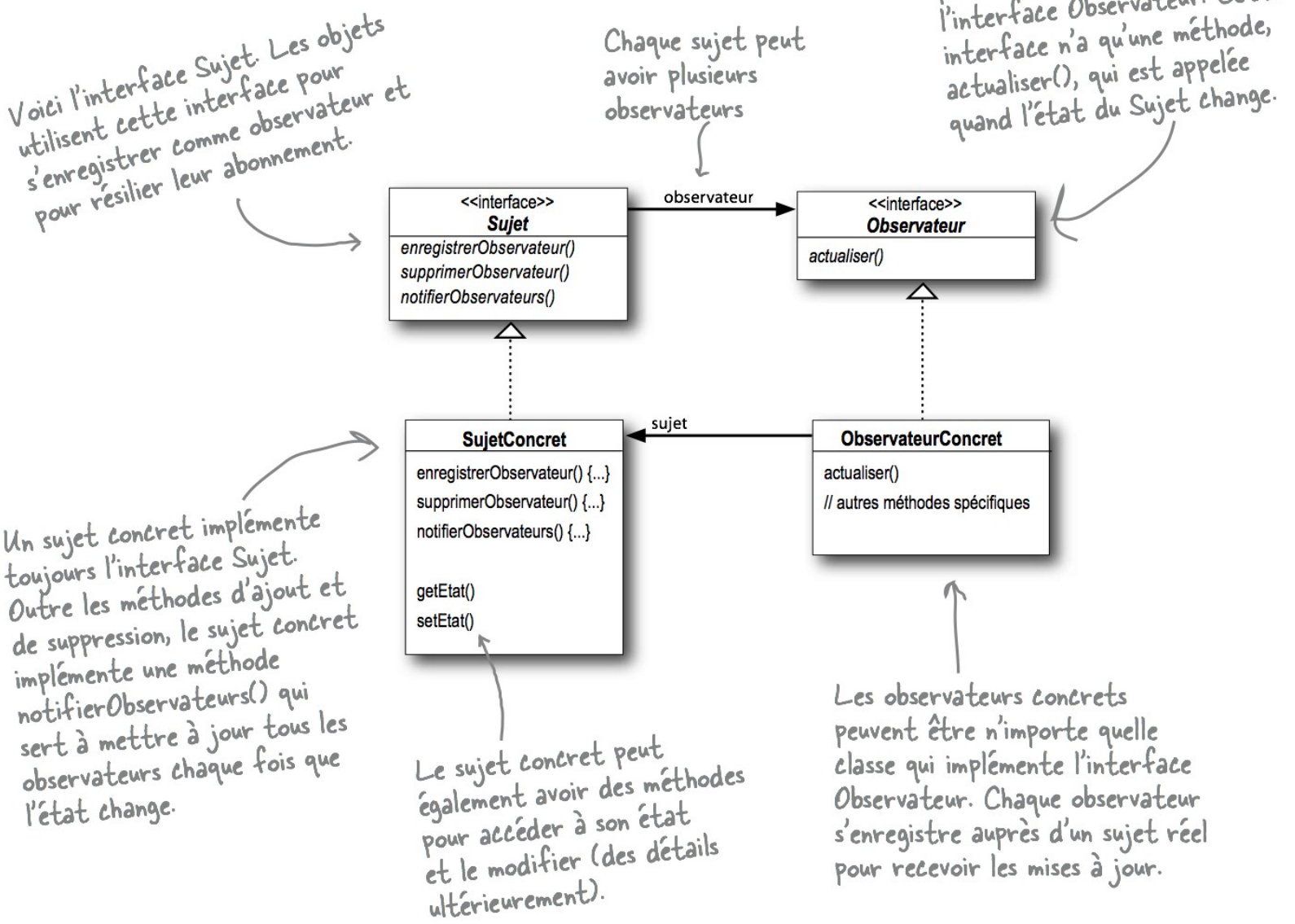
- Définition
  - Il propose une solution **1-à-plusieurs** de **notification** entre 1 objet **observé** et plusieurs **observateurs**
  - Quand l'état du **sujet** d'observation **change**, tous les observateurs sont **informés**
  - La notification peut éventuellement aussi transmettre directement aux abonnés des **valeurs**
    - Si non, ce sont les observateurs qui devront interrogés le sujet d'observation après avoir été notifiés
- Utilité
  - Il permet un **découplage** entre l'observé et les observateurs
    - L'observé ne connaît pas le ***type réel*** des observateurs
  - … mais pas l'inverse : les observateurs voient le sujet **concret**
- Mécanismes
  - Observateur = SOUSCRIPTION + DIFFUSION
  - Même technique que **l'abonnement** à un magazine !

Observateur – illustration de l'approche



Observateur – modélisation UML

le diagramme de classes



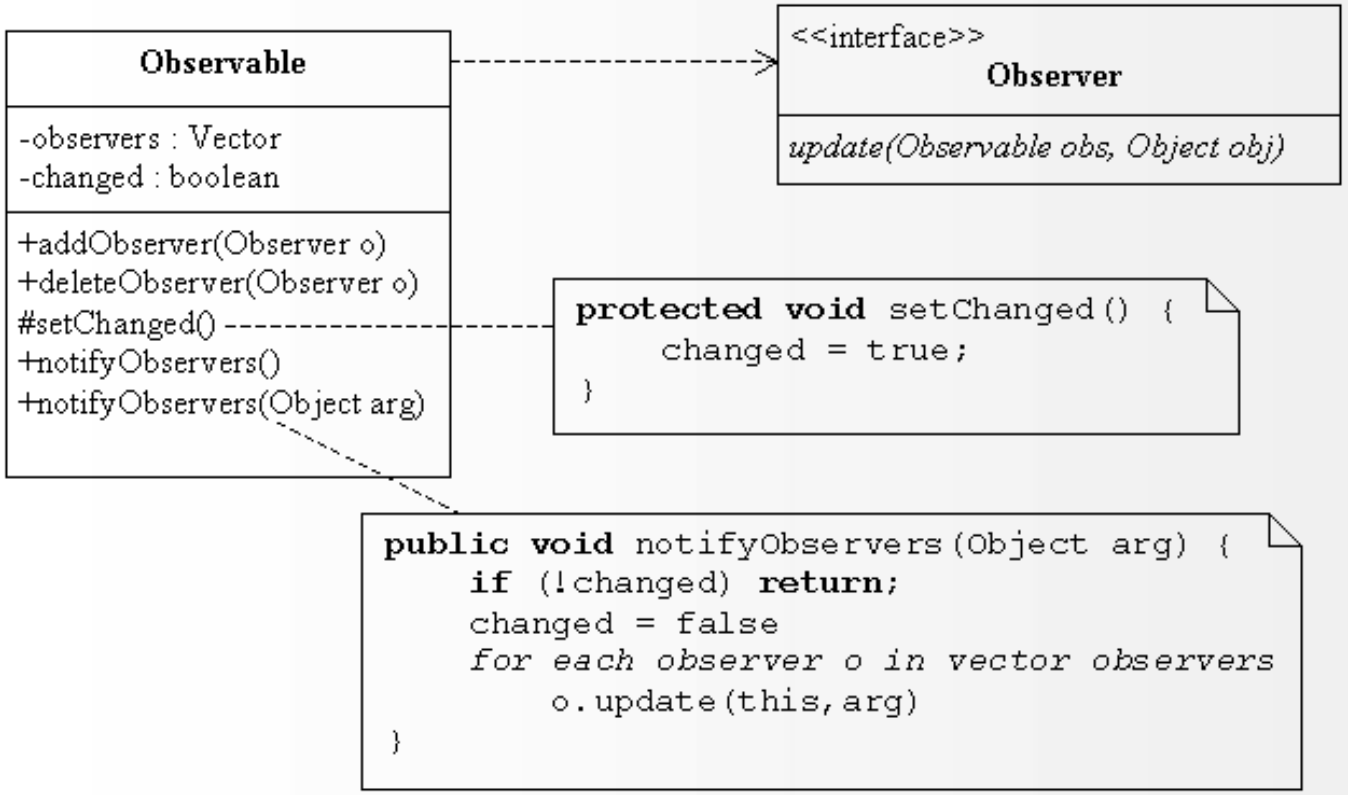
Observateur et Java

- Le patron est appliqué pour toutes les implémentation de l'interface *java.util.EventListener*
  - *pratiquement* tout dans la librairie Swing utilise ce patron !

- Le SDK de Java propose une solution générique pour que les développeurs appliquent ce patron :
  - **java.util.Observer / java.util.Observable**

Observateur et Java

- Modélisation UML de la solution standard de Java



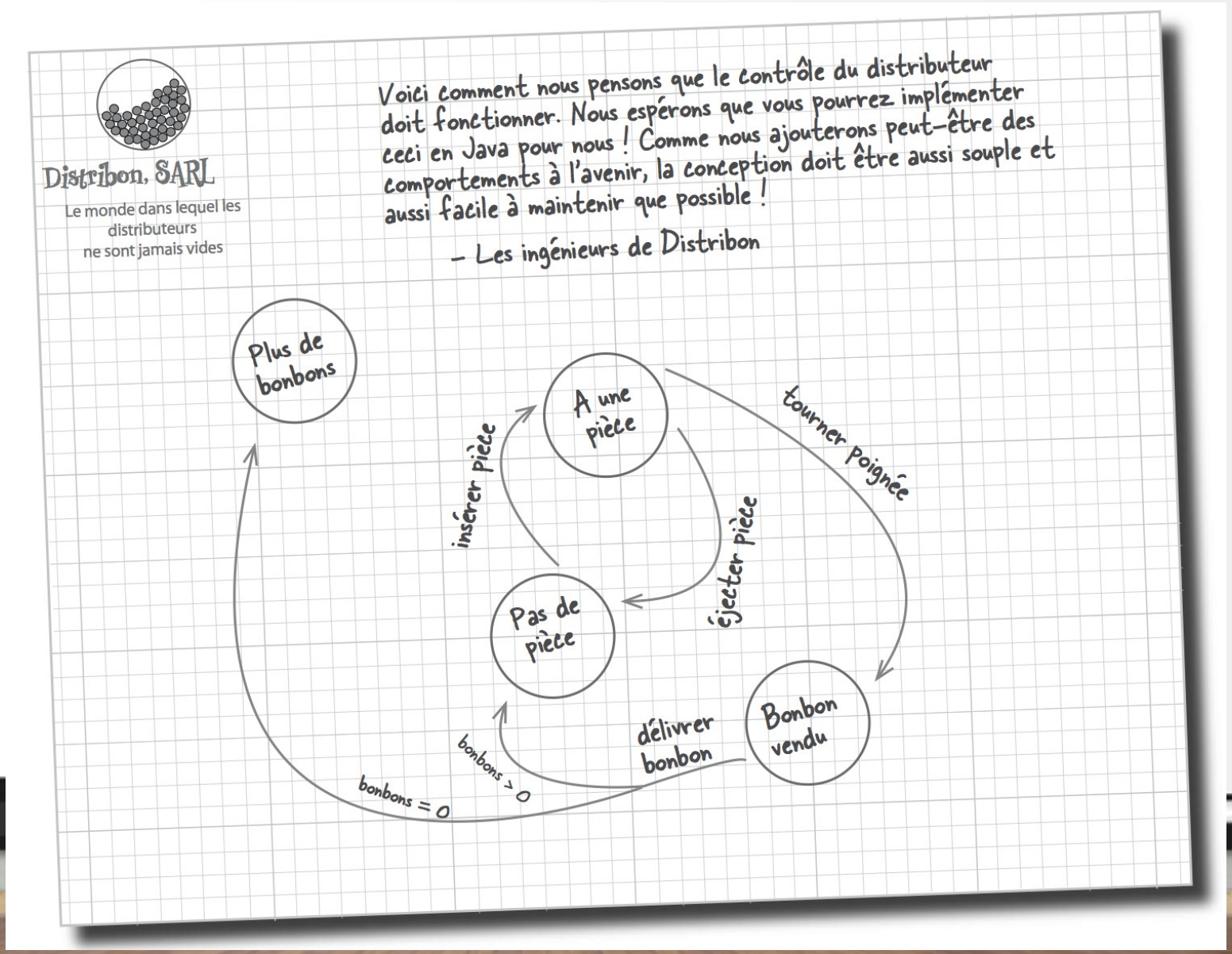
Quel est le principal défaut ?

Observateur – exemple de codage

- Premier codage en Java

```
class Signal extends Observable {

  void setData(byte[] lbData){
    setChanged();        // Positionne son indicateur de changement
    notifyObservers(); // (1) notification
  }
}
class JPanelSignal extends JPanel implements Observer {

  void init(Signal lSigAObserver) {
    lSigAObserver.addObserver(this); // (2) ajout d'observateur
  }

  void update(Observable observable, Object objectConcerne) {
    repaint();           // (3) traitement de l'observation
  }
}
```

Patron Etat – introduction : distributeur de bonbons



Comment coder en Java ?

Appelons « Plus de bonbons »
« Epuisé » pour abréger.

```
final static int EPUISE = 0;
final static int SANS_PIECE = 1;
final static int A_PIECE = 2;
final static int VENDU = 3;

int etat = EPUISE;
```

Voici chaque état représenté par un entier unique....

...et voilà la variable d'instance qui contient l'état courant. Nous la positionnons à « Epuise » puisque l'appareil est vide quand on le sort de sa boîte et qu'on le met en service.

insérer pièce

tourner poignée

éjecter pièce

délivrer

Ces actions représentent l'interface du distributeur : ce que vous pouvez faire avec.

« délivrer » est plutôt une action interne que la machine invoque sur elle-même.

Par rapport au diagramme, l'appel de l'une quelconque de ces actions peut déclencher une transition.

Patron Etat – codage d'une des transitions

```
public void insererPiece() {

  if (etat == A_PIECE) {

      System.out.println("Vous ne pouvez plus insérer de pièces");

  } else if (etat == EPUISE) {

      System.out.println("Vous ne pouvez pas insérer de pièce, nous sommes en rupture de stock");

  } else if (etat == VENDU) {

      System.out.println("Veuillez patienter, le bonbon va tomber");

  } else if (etat == SANS_PIECE) {

      etat = A_PIECE;
      System.out.println("Vous avez inséré une pièce");
  }
}
```

Chaque état possible est testé à l'aide d'une instruction conditionnelle...

...et présente le comportement approprié pour chaque état possible...

...mais peut également déclencher une transition vers d'autres états, comme le montre le diagramme..

Patron état : première version

- Bien que le code fonctionne, que se passe t'il en cas de nouveaux besoins / changements ?
  - ❏ A. Ce code n'adhère pas au principe Ouvert-Fermé
  - ❏ B. Ce code ferait la fierté d'un programmeur FORTRAN
  - ❏ C. Cette conception n'est pas orientée objet
  - ❏ D. Les transitions ne sont pas explicites. Elles sont enfouies au milieu d'un tas d'instructions conditionnelles
  - ❏ E. Nous n'avons pas encapsulé ce qui varie
  - ❏ F. Les ajouts ultérieurs sont susceptibles de provoquer des bogues dans le code

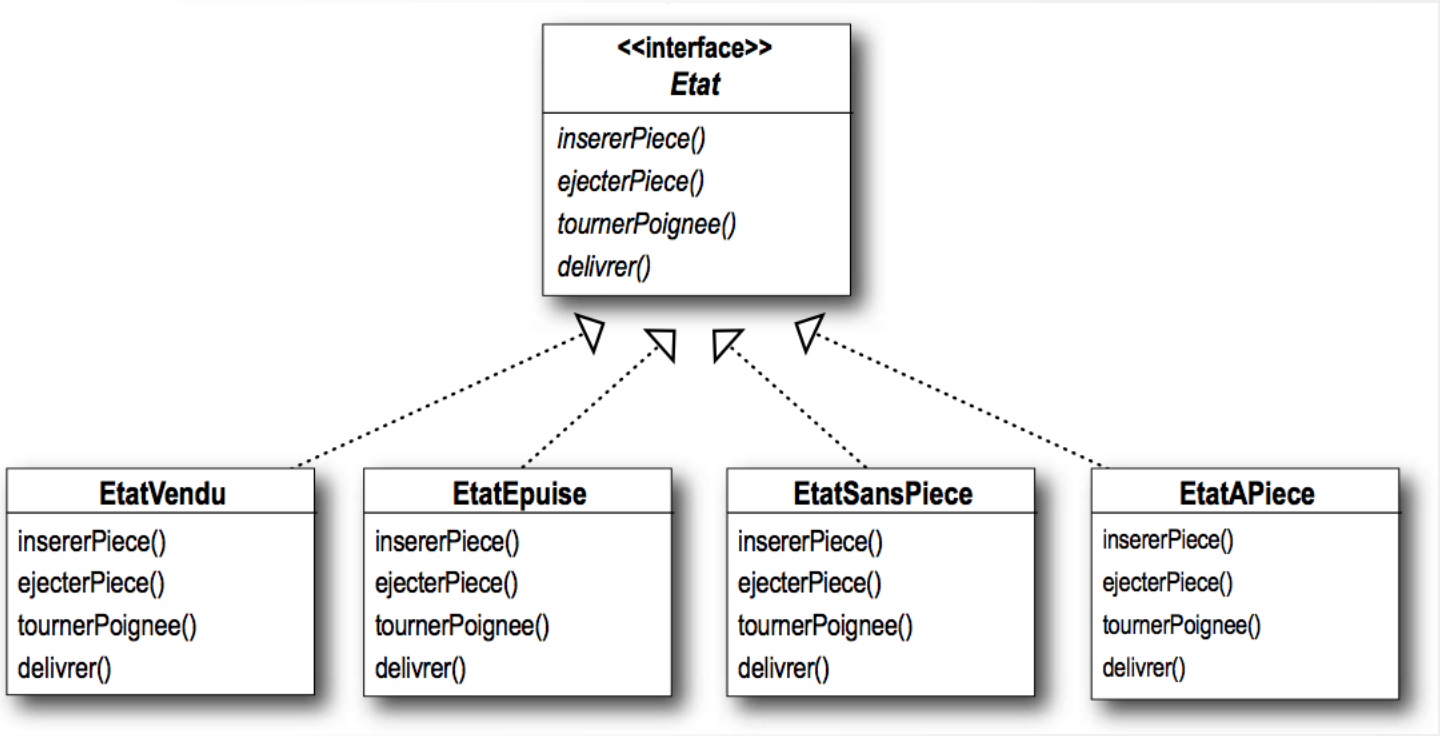
Patron état : nouvelle conception

- Tout d'abord, nous allons définir une **interface Etat** qui contiendra une **méthode** pour chaque **action** liée au Distributeur.

- Ensuite, nous allons **implémenter une classe** pour chaque **état** de la machine.
  - Ces classes seront responsables du comportement du distributeur quand il se trouvera dans l'état correspondant

- Enfin, nous allons nous débarrasser de toutes nos instructions conditionnelles et les **remplacer par une délégation** à la classe état qui travaillera à notre place

Patron état : solution au problème



Patron état : exemple implémentation EtatSansPiece

Nous devons d'abord implémenter l'interface Etat

On transmet une référence au Distributeur dans le constructeur et on la place dans une variable d'instance.

```
public class EtatSansPiece implements Etat {
  Distributeur distributeur;

  public EtatSansPiece(Distributeur distributeur) {
    this.distributeur = distributeur;
  }

  public void insererPiece() {
    System.out.println("Vous avez inséré une pièce");
    distributeur.setEtat(distributeur.getEtatAPiece());
  }

  public void ejecterPiece() {
    System.out.println("Vous n'avez pas inséré de pièce");
  }

  public void tournerPoignee() {
    System.out.println("Vous avez tourné, mais il n'y a pas de pièce");
  }

  public void delivrer() {
    System.out.println("Il faut payer d'abord");
  }
}
```

Si quelqu'un insère une pièce, nous affichons un message indiquant qu'elle a été acceptée et nous passons dans l'état EtatAPiece.

Vous allez voir dans une seconde comment ceci fonctionne...

Vous ne pouvez pas reprendre votre pièce : vous ne nous l'avez jamais donnée !

Et si vous ne payez pas, vous ne pouvez pas avoir de bonbon.

Impossible de donner gratuitement des bonbons..

Patron état : le distributeur

```
public class Distributeur {

  Etat etatEpuise;
  Etat etatSansPiece;
  Etat etatAPiece;
  Etat etatVendu;

  Etat etat = etatEpuise;
  int nombre = 0;

  public Distributeur(int nombreBonbons) {
    etatEpuise = new EtatEpuise(this);
    etatSansPiece = new EtatSansPiece(this);
    etatAPiece = new EtatAPiece(this);
    etatVendu = new EtatVendu(this);
    this.nombre = nombreBonbons;
    if (nombreBonbons > 0) {
        etat = etatSansPiece;
    }
  }

  public void insererPiece() {
    etat.insererPiece();
  }

  public void ejecterPiece() {
    etat.ejecterPiece();
  }

  public void tournerPoignee() {
    etat.tournerPoignee();
    etat.delivrer();
  }

  void setEtat(Etat etat) {
    this.etat = etat;
  }

  void liberer() {
    System.out.println("Un bonbon va sortir...");
    if (nombre != 0) {
        nombre = nombre - 1;
    }
  }
  // Autres méthodes, dont une méthode get pour chaque état...
}
```

Voici de nouveau les États...

...et la variable d'instance etat.

La variable d'instance nombre contient le nombre de bonbons – initialement, le distributeur est vide.

Notre constructeur accepte le nombre de bonbons initial et le stocke dans une variable d'instance.

Il crée également les instances des états, une pour chacun.

S'il y a plus de 0 bonbons, nous positionnons l'état à EtatSansPiece.

Au tour des actions. Maintenant, elles sont TRÈS FACILES à implémenter. Il suffit de déléguer à l'état courant.

Notez que nous n'avons pas besoin de méthode d'action pour delivrer() dans la classe Distributeur parce que ce n'est qu'une action interne : l'utilisateur ne peut pas demander directement à la machine de délivrer. Mais nous appelons bien delivrer() sur l'objet état depuis la méthode tournerPoignee().

Cette méthode permet à d'autres objets (comme nos objets état) de déclencher une transition de la machine vers un autre état.

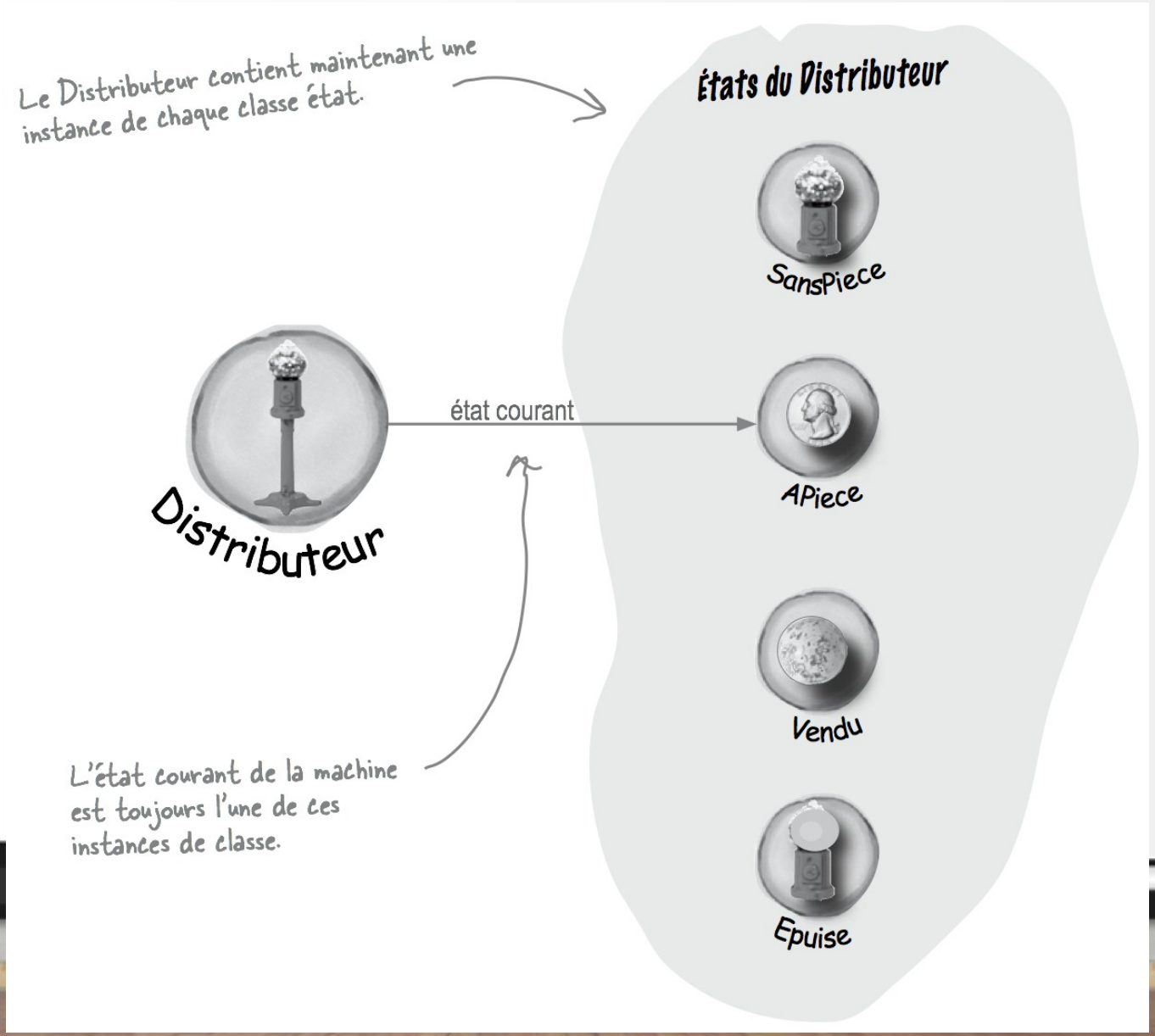
La machine prend en charge une méthode auxiliaire, liberer(), qui libère le bonbon et décrémente variable d'instance nombre.

Par exemple des méthodes comme getEtatSansPiece() pour obtenir chaque objet état et getNombre() pour obtenir le nombre de bonbons.

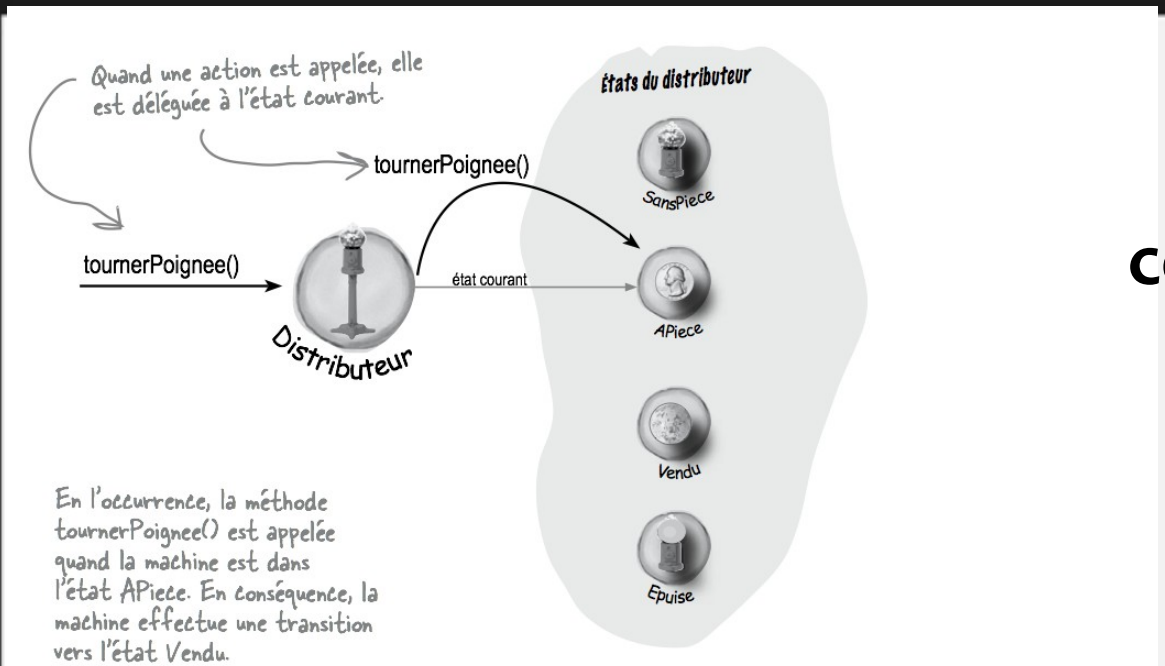
Patron état – exercice (maison / examen?)

- Sauriez-vous maintenant implémenter les autres états ?

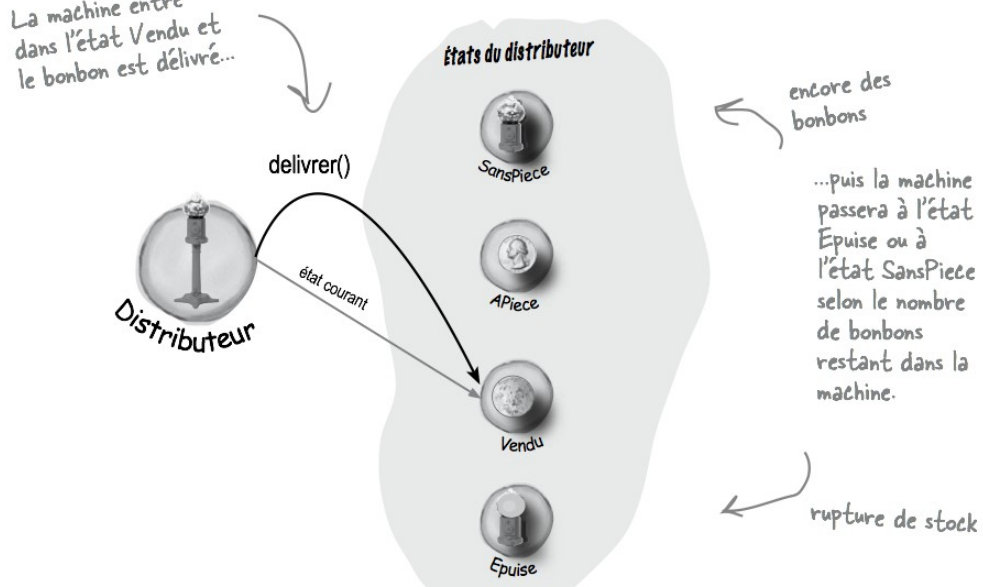
Patron état : conceptualisation de la solution



Patron état : conceptualisation de la solution



TRANSITION VERS L'ÉTAT VENDU



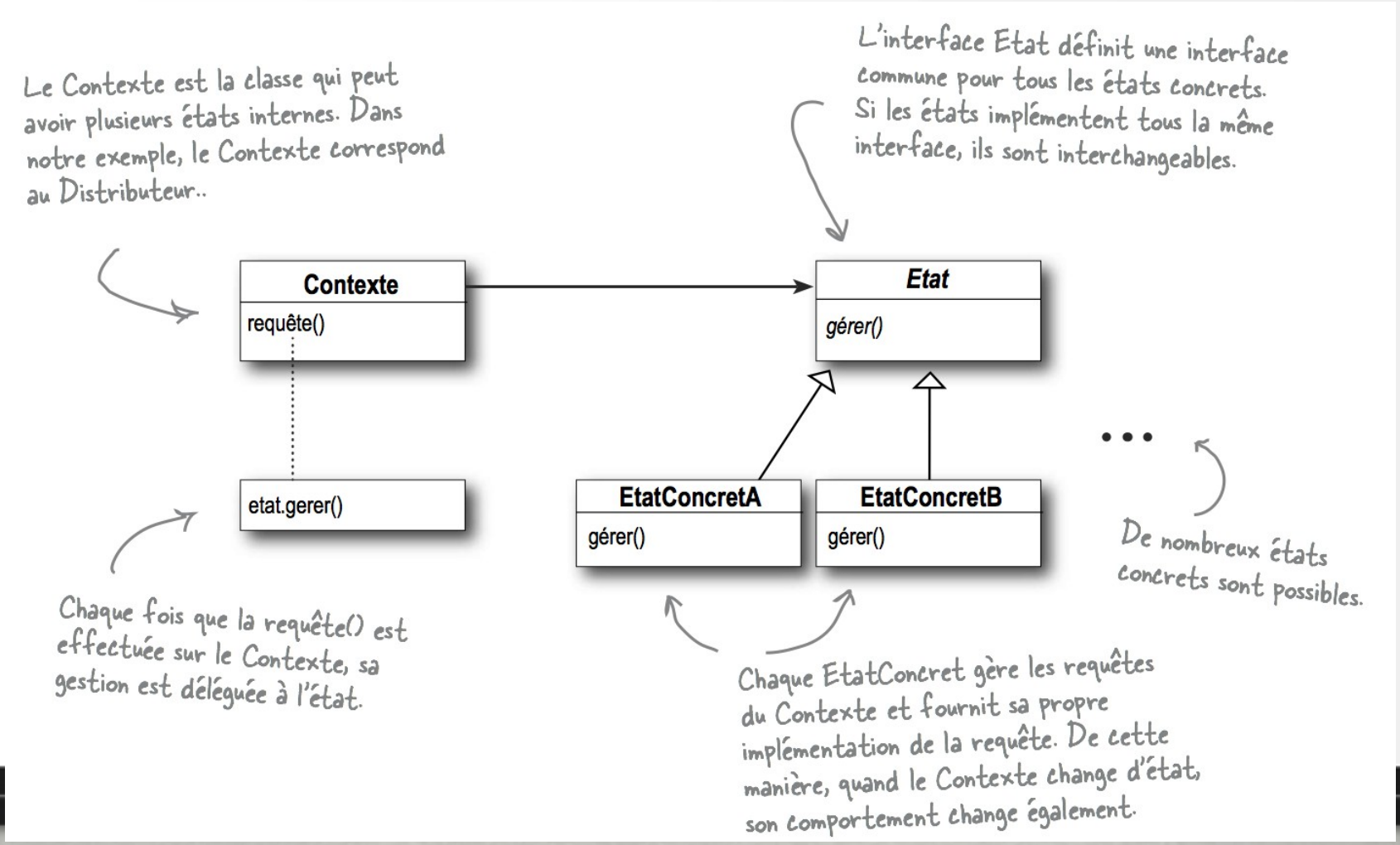
Patron Etat

- Définition
  - permet à un objet de modifier son comportement quand son état interne change
  - Tout se passera comme si l'objet changeait de classe

- Mise en oeuvre
  - **Encapsulation** des états
  - **Composition** vers l'état courant et **délégation**

Etat

- Modélisation UML

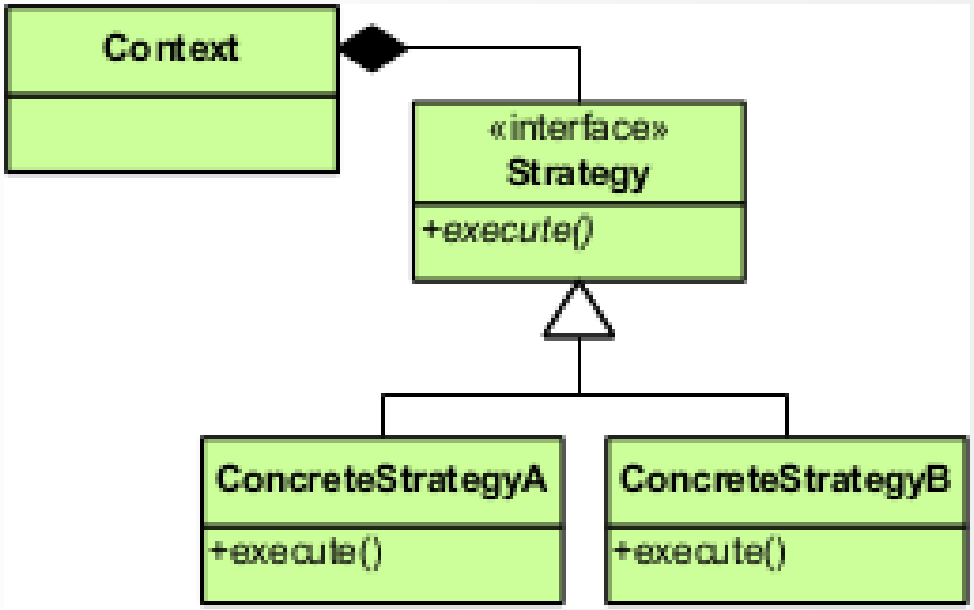


Patron Stratégie

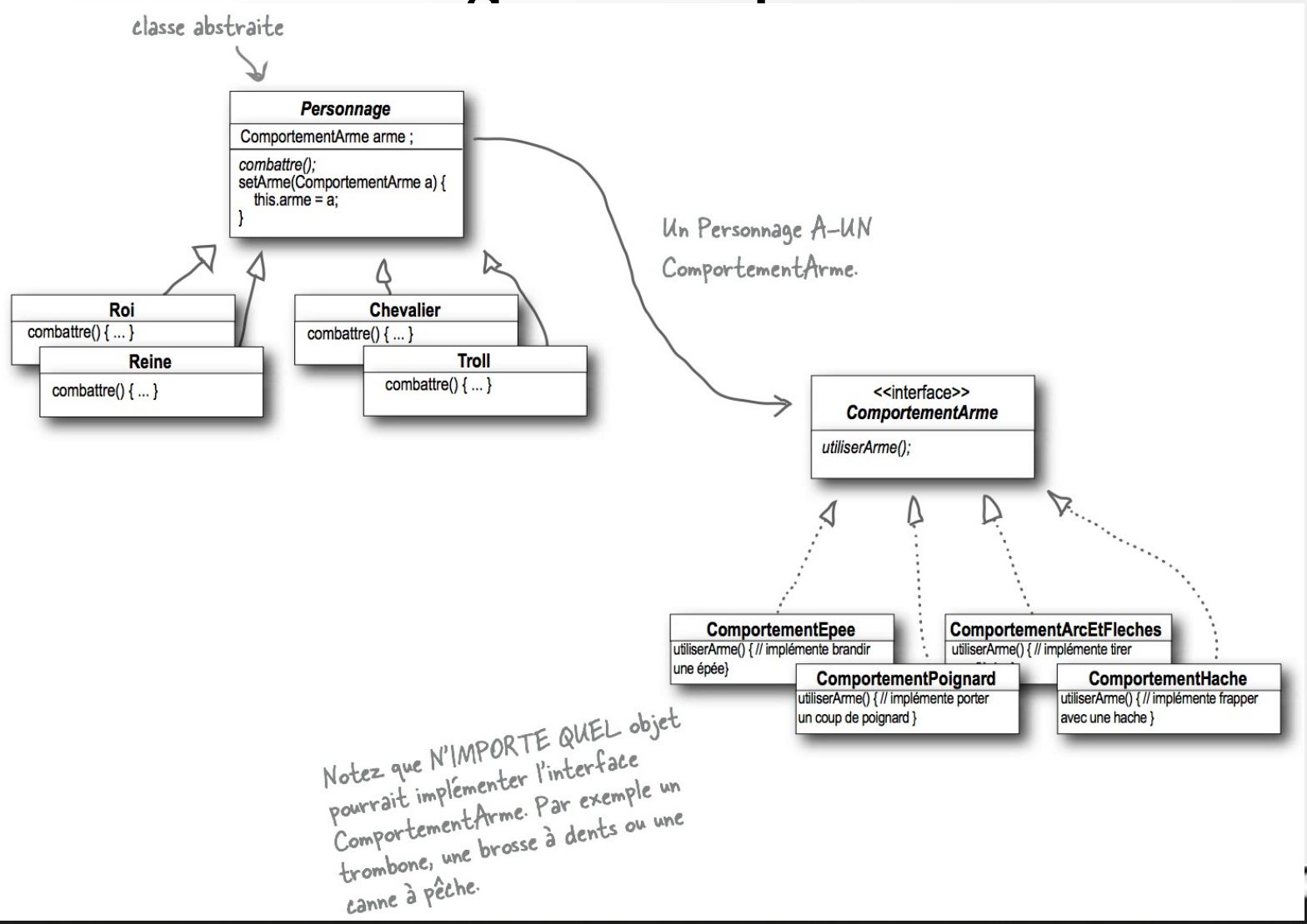
- Problème/Définition
  - comment permettre à un algorithme (ou un comportement) de varier indépendamment des clients qui l'utilisent ?
- Solution
  - définir une **famille d'algorithmes**
  - **encapsuler** chacun d'eux et les rendre **interchangeables**
- Patron voisin
  - **Stratégie** et **état** visent des objectifs différents mais s'appuie sur la **même solution** !
- Implantation connue dans le *core* de Java
  - `java.util.Comparator#compare()`, exécutée par la plupart des implémentations de `Collections#sort()`

Patron Stratégie

- Modélisation UML



Patron Stratégie – exemple d'utilisation



Patron de méthode

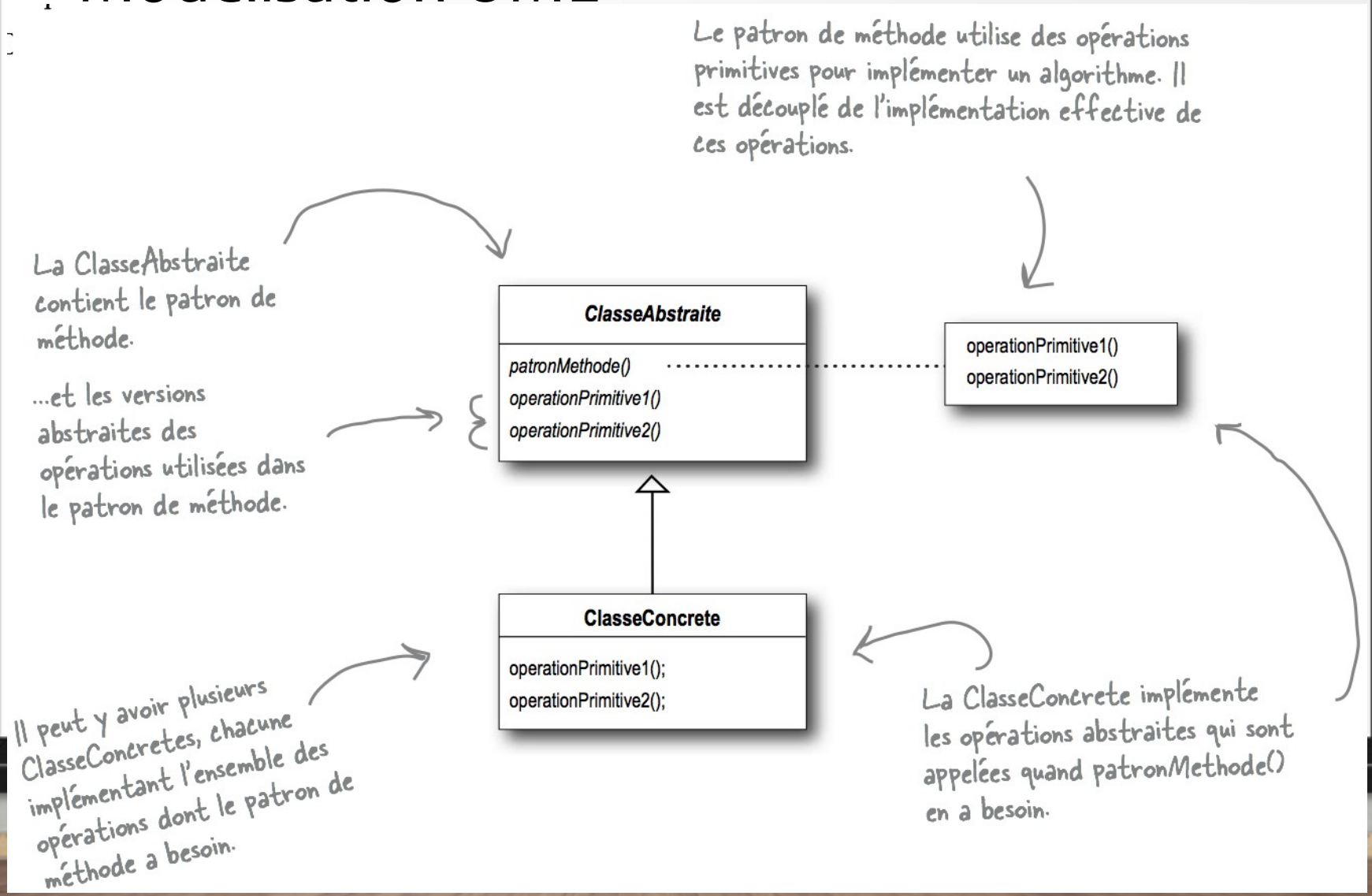
- Problème/contexte
  - Factoriser les éléments d'algorithme commun (code redondant) au comportement de 2 objets différents
- Solution
  - **définir le squelette de l'algorithme dans une méthode d'une classe abstraite, en déléguant certaines étapes aux sous-classes**
  - **permettre aux sous-classes de redéfinir certaines étapes de l'algorithme sans modifier la structure de celui-ci**
- Implantations connues dans le *core* de Java
  - Dans les méthodes non-abstraites de *java.io.InputStream*, *java.io.OutputStream*, *java.io.Reader* et *java.io.Writer*
  - Et aussi *java.util.AbstractList*, *java.util.AbstractSet* et *java.util.AbstractMap*

Patron de méthode

- Détails sur la technique
  - c'est la **méthode de la classe parent** qui appelle des opérations n'existant que dans les sous-classes !
    - C'est une pratique courante dans les classes abstraites, alors que d'habitude dans une hiérarchie de classes concrètes c'est le contraire : ce sont plutôt les méthodes des sous-classes qui appellent les méthodes de la super-classe comme morceau de leur propre comportement
  - L'implémentation d'un patron de méthode est parfois appelée **méthode socle** parce qu'elle ancre solidement un comportement qui s'applique alors à toute la hiérarchie de classes par héritage
    - Pour s'assurer que ce comportement ne sera pas redéfini arbitrairement dans les sous-classes, on déclare la méthode socle **final en Java**

Patron de méthode

- Modélisation UML



Patron de méthode - Exemple de codage en Java

- La classe abstraite commune à divers jeux

```
abstract class JeuDeSociété{

   protected int nombreDeJoueurs;

   abstract void initialiserLeJeu();

   abstract void faireJouer(int joueur);

   abstract boolean partieTerminée();

   abstract void proclamerLeVainqueur();

   /* Une méthode socle : */
   final void jouerUnePartie(int nombreDeJoueurs){
     this.nombreDeJoueurs = nombreDeJoueurs;
     initialiserLeJeu();
     int j = 0;
     while( ! partieTerminée() ){
       faireJouer( j );
       j = (j + 1) % nombreDeJoueurs;
     }
     proclamerLeVainqueur();
   }
 }
```

Patron de méthode - Exemple de codage en Java

- On dérive pour créer des jeux spécifiques

```
class Monopoly extends JeuDeSociété{

   /* Implémentation concrète des méthodes nécessaires */

   void initialiserLeJeu(){
     // ...
   }

   void faireJouer(int joueur){
     // ...
   }

   boolean partieTerminée(){
     // ...
   }

   void proclamerLeVainqueur(){
     // ...
   }

   /* Déclaration des composants spécifiques au jeu du Monopoly */
    // ...

 }
```

Patron de méthode – autre exemple

```
public abstract class BoissonCafeinee {
    final void suivreRecette() {
        faireBouillirEau();
        preparer();
        verserDansTasse();
        ajouterSupplements();
    }
    abstract void preparer();
    abstract void ajouterSupplements();

    void faireBouillirEau() {
        System.out.println("Portage de l'eau à ébullition");
    }
    void verserDansTasse(){
        System.out.println("Remplissage de la tasse");
    }
}
```

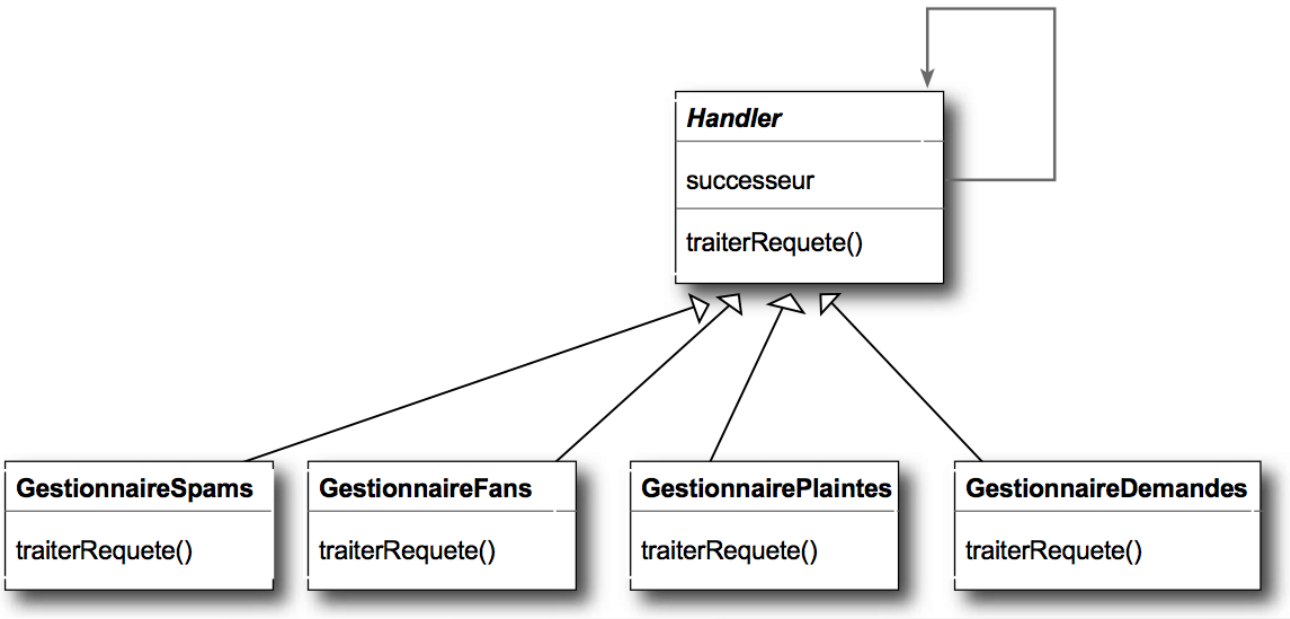
Patron de méthode – autre exemple

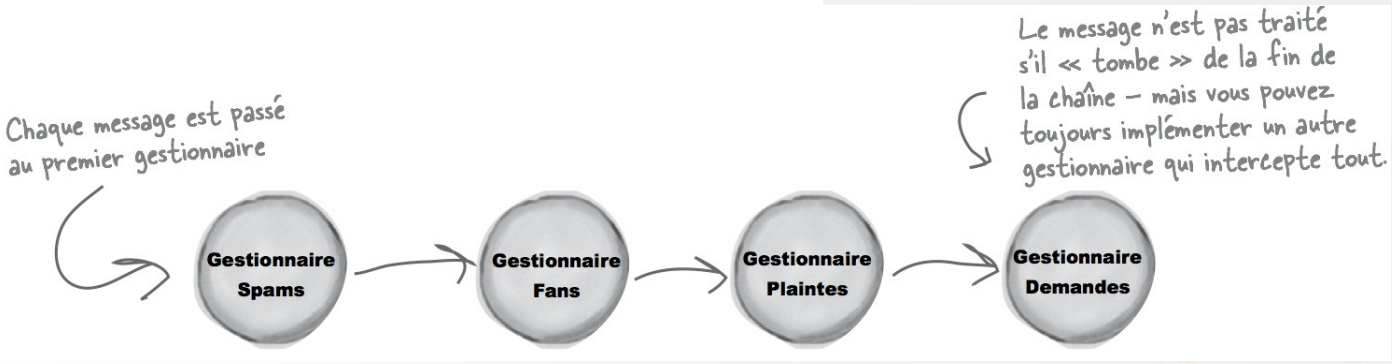
```
public class The extends BoissonCafeinee {
    public void preparer() {
        System.out.println("Infusion du thé");
    }
    public void ajouterSupplements() {
        System.out.println("Ajout du citron");
    }
}

public class Coffee extends BoissonCafeinee {
    public void preparer() {
       System.out.println("Passage du café");
    }
    public void ajouterSupplements() {
       System.out.println("Ajout du lait et du sucre");
    }
}
```

Patron Chaine de responsabilité

- Problème/contexte
  - permettre à un nombre quelconque de classes d'essayer de répondre à une requête sans connaître les possibilités des autres classes sur cette requête
  - Réduire le couplage entre ces objets
- Solution
  - **créer une chaîne d'objets qui examinent une requête**
  - **chaque objet considère la requête à son tour et la traite ou la transmet à l'objet suivant dans la chaîne**
- Implantations connues dans le *core* de Java
  - `java.util.logging.Logger#log()`

Patron Chaine de responsabilité - exemple





Chaine de responsabilité – autre exemple en Java

- La classe abstraite

```
abstract class Logger {
   public static int ERR = 3;
   public static int NOTICE = 5;
   public static int DEBUG = 7;
   protected int mask;

   // The next element in
   // the chain of responsibility
   protected Logger next;

   public void setNext(Logger log) {
      next = log;
   }
```

```
public void message(String msg, int priority)
{
        if (priority <= mask) {
            writeMessage(msg);
        }
        if (next != null) {
            next.message(msg, priority);
        }
    }

    abstract protected void
writeMessage(String msg);
}
```

Chaine de responsabilité – autre exemple en Java

- Les 3 classes concrètes

```
class StdoutLogger extends Logger {
   public StdoutLogger(int mask) {
      this.mask = mask;
   }
    protected void writeMessage(String msg) {
      System.out.println("Writing to stdout: "
        + msg);
   }
}
 class EmailLogger extends Logger {
   public EmailLogger(int mask) {
      this.mask = mask;
   }
    protected void writeMessage(String msg) {
      System.out.println("Sending via e-mail: " + msg);
   }
}
```

```
class StderrLogger extends Logger {
    public StderrLogger(int mask) {
        this.mask = mask;
    }

    protected void writeMessage(String msg)
{
        System.err.println("Sending to stderr: "
+ msg);
    }
}
```

Chaine de responsabilité – autre exemple en Java

- La classe cliente

```
public class ChainOfResponsibilityExample {

    private static Logger createChain() {
        // Build the chain of responsibility

        Logger logger = new StdoutLogger(Logger.DEBUG);

        Logger logger1 = new EmailLogger(Logger.NOTICE);
        logger.setNext(logger1);

        Logger logger2 = new StderrLogger(Logger.ERR);
        logger1.setNext(logger2);

        return logger;
    }

    public static void main(String[] args) {
        Logger chain = createChain();

        // Handled by StdoutLogger (level = 7)
        chain.message("Entering function y.", Logger.DEBUG);

        // Handled by StdoutLogger and EmailLogger (level = 5)
        chain.message("Step1 completed.", Logger.NOTICE);

        // Handled by all three loggers (level = 3)
        chain.message("An error has occurred.", Logger.ERR);
    }
}
```

Chaine de responsabilité – autre exemple en Java

- Sorties en console

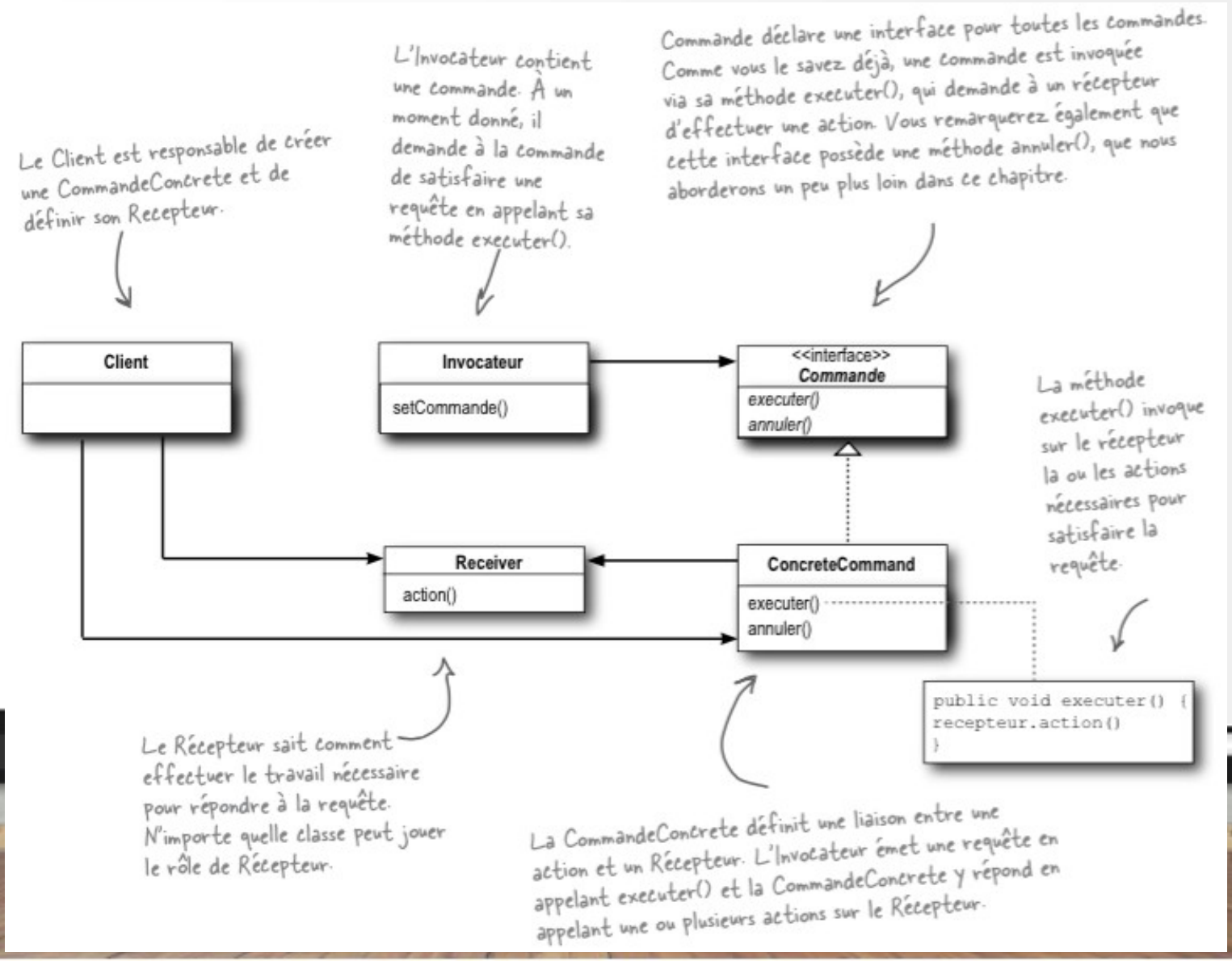
```
Writing to stdout:   Entering function y.
Writing to stdout:   Step1 completed.
Sending via e-mail:  Step1 completed.
Writing to stdout:   An error has occurred.
Sending via e-mail:  An error has occurred.
Sending to stderr:   An error has occurred.
```

Patron Commande

- Problème/contexte
  - dans les interfaces graphiques, par exemple, un élément de menu peut être connecté à différentes commandes => on veut qu'il n'ait pas besoin de connaître les détails de l'action effectuée
- Solution
  - **encapsuler la notion d'invocation**
  - **séparer complètement le code initiateur de l'action du code de l'action elle-même**
  - **un objet Commande sert alors à communiquer l'action à effectuer, ainsi que les arguments requis**
  - **l'objet commande est envoyé à une seule méthode dans une classe, qui traite les Commandes du type requis**
- Implantations connues dans le *core* de Java
  - `Toutes les implémentations de java.lang.Runnable`
  - `Toutes les implémentations de javax.swing.Action`

Patron commande

- Modélisation UML de la solution



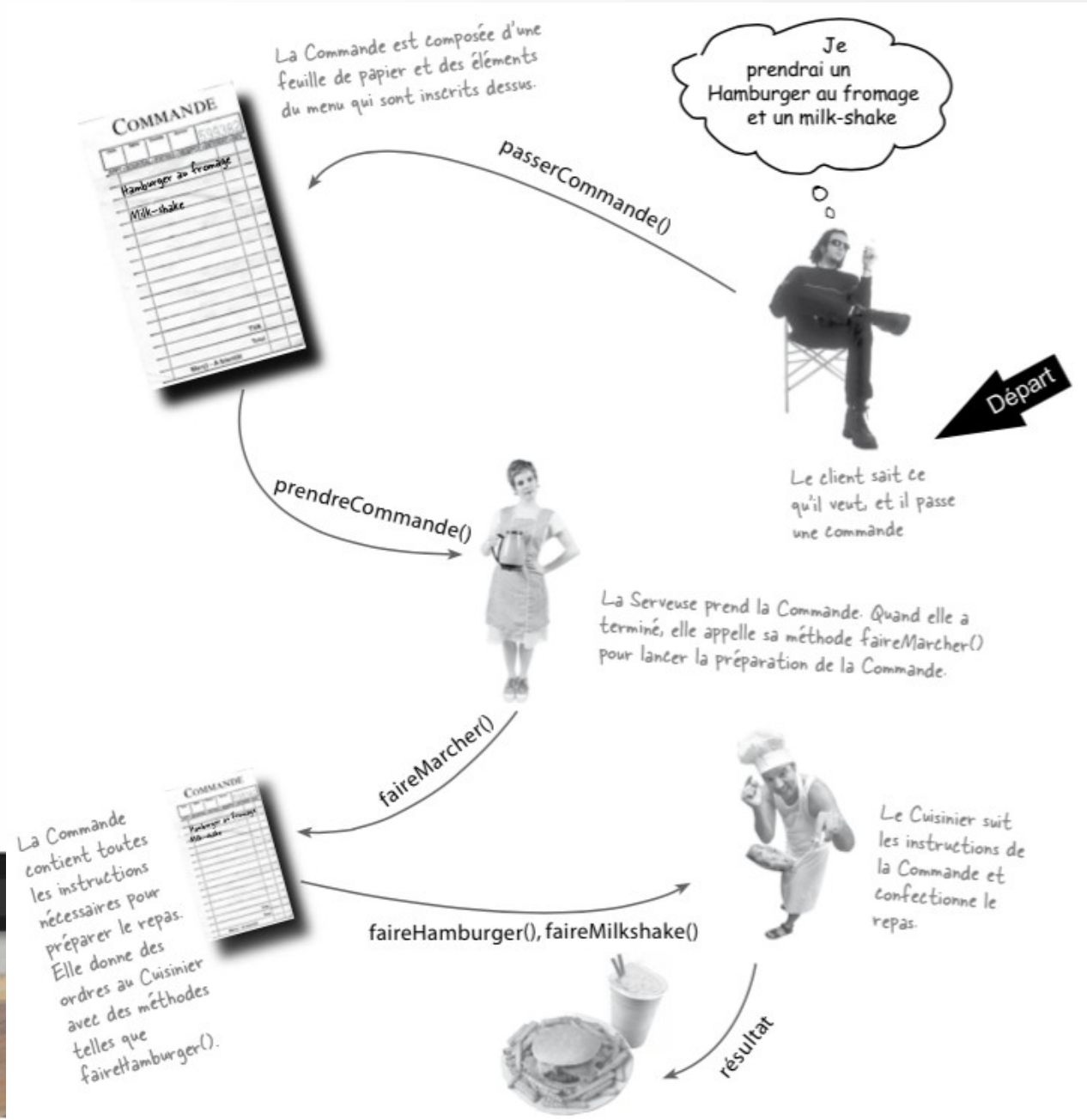
Une exemple concret de commande

❶ Vous, le Client, passez votre Commande à la Serveuse.

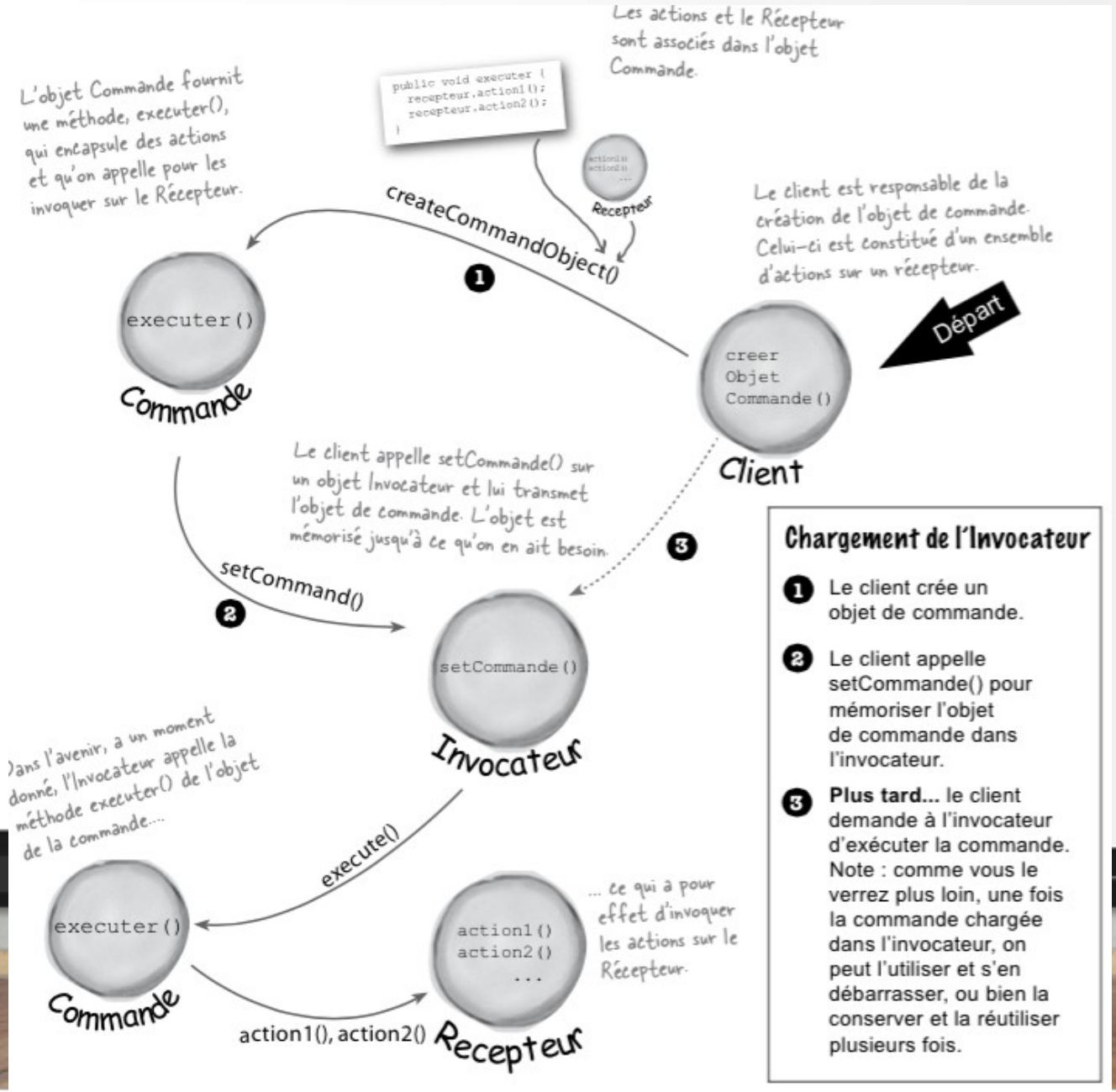
❷ La Serveuse prend la Commande, la place sur le comptoir et dit « Ça marche ! »

❸ Le Cuisinier lit la Commande et prépare votre repas.

Regardons plus en détail les interactions



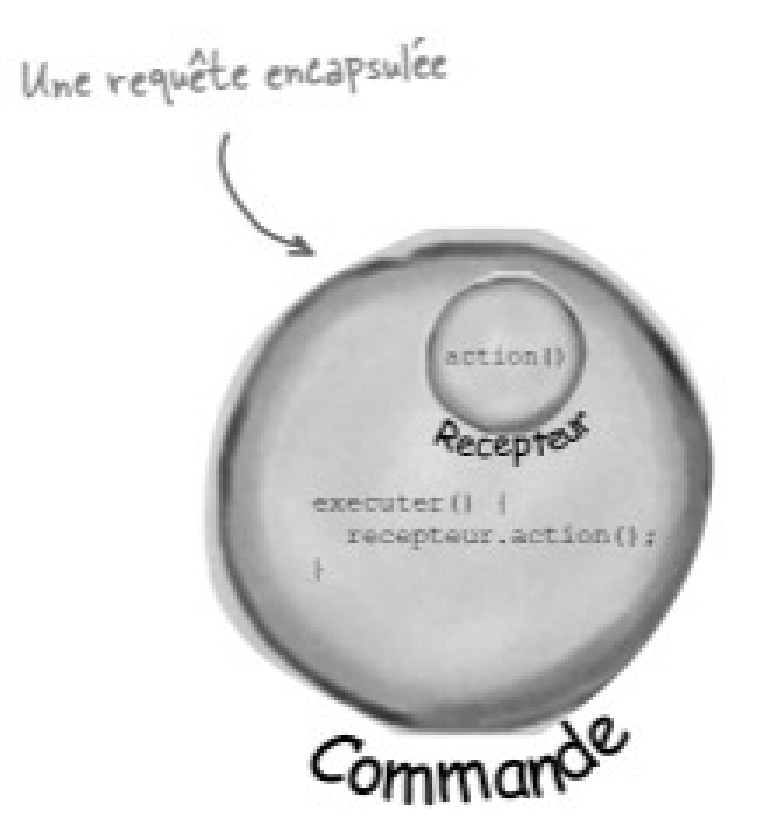
Retour au patron commande



Chargement de l'Invocateur

1. Le client crée un objet de commande.
2. Le client appelle setCommande() pour mémoriser l'objet de commande dans l'invocateur.
3. **Plus tard...** le client demande à l'invocateur d'exécuter la commande. Note : comme vous le verrez plus loin, une fois la commande chargée dans l'invocateur, on peut l'utiliser et s'en débarrasser, ou bien la conserver et la réutiliser plusieurs fois.

La commande encapsule récepteur et actions !



Petit exercice : qui fait quoi ?

Cafétéria	Command Pattern
Serveuse	Commande
Cuisinier	executer()
faireMarcher()	Client
Commande	Invocateur
Client	Récepteur
prendreCommande()	setCommande()

Exemple Java simple

- L'interface Commande

```
public interface Commande {
    public void executer();
}
```

- Une commande concrète

```
public class CommandeAllumerLampe implements Commande {
    Lampe lampe;

    public CommandeAllumerLampe(Lampe lampe) {
        this.lampe = lampe;
    }
    public void executer() {
        lampe.marche();
    }
}
```

Le receveur

Exemple Java simple

- La classe invocatrice

```
public class TelecommandeSimple {
        Commande emplacement;

        public TelecommandeSimple() {}

        public void setCommande(Commande commande) {
                emplacement = commande;
        }
        public void boutonPresse() {
                emplacement.executer();
        }
}
```

Exemple Java simple

- Le client

```
public class TestTelecommande {

	public static void main(String[] args) {
		TelecommandeSimple telecommande = new TelecommandeSimple();
		Lampe lampe = new Lampe();

		CommandeAllumerLampe lampeAllumee = new CommandeAllumerLampe(lampe);

		telecommande.setCommande(lampeAllumee);

		telecommande.boutonPresse();
	}
}
```

Patron Interpréteur

- Problème/contexte
  - Analyser/parser/interpréter une chaîne algébrique d'actions spécifiques
- Solution
  - **Construire un interpréteur pour un langage**
  - **Revient à coder une grammaire formelle d'un langage**

- Exemple de langage à interpréter

Voici un exemple du langage :

```
droite;
tantque(faitjour) voler;
cancaner;
```

Faire tourner le canard à droite.

Voler toute la journée...

...Puis cancaner.

Patron Interpréteur

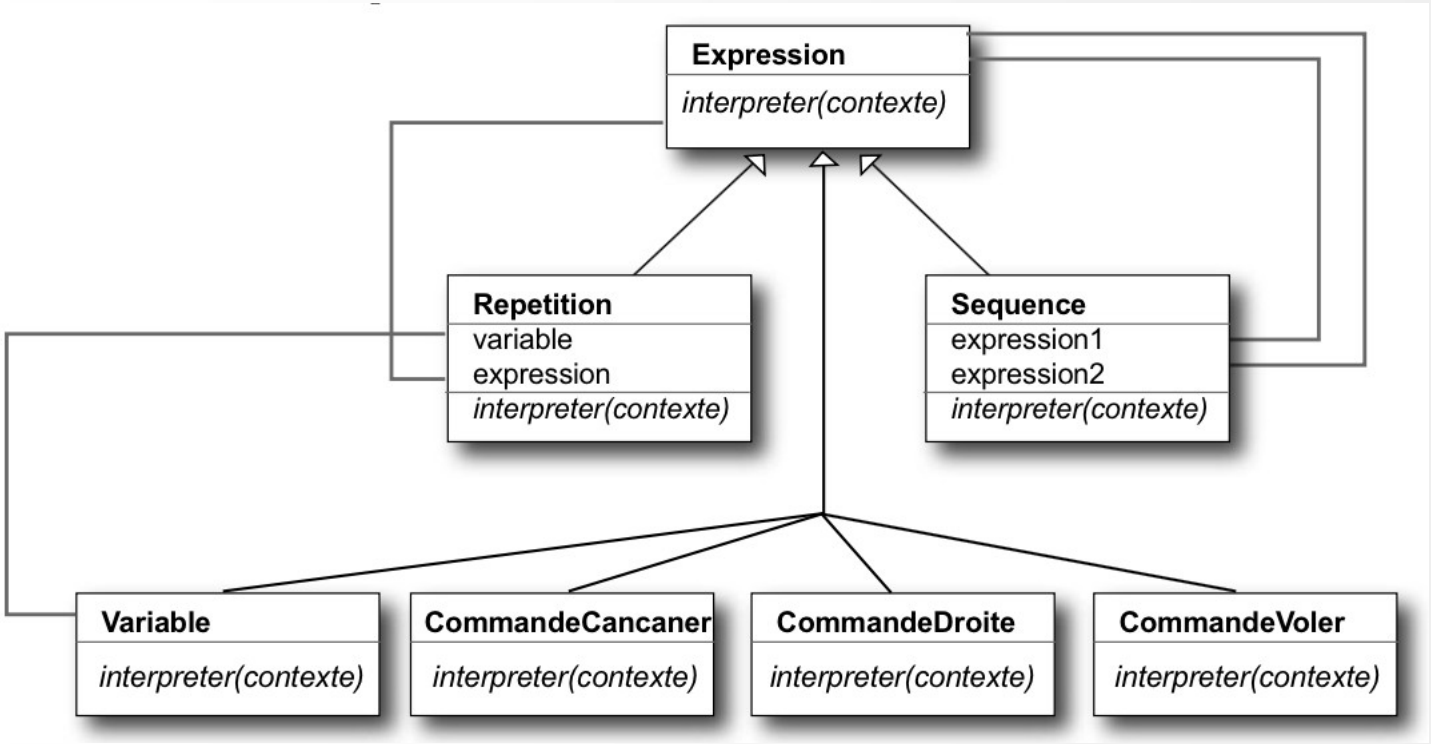
- La grammaire donnerait :

  expression ::= <commande> | <sequence> | <repetition>
  sequence ::= <expression> ‘;’ <expression>
  commande ::= droite | voler | cancaner
  repetition ::= tantque ‘(‘ <variable> ‘)’<expression>
  variable ::= [A-Z,a-z]+

- Le patron Interpréteur permet de définir
  - des classes pour représenter la grammaire
  - un *interpréteur* pour interpréter les phrase
- Utilisation d'une classe pour représenter chaque règle du langage

Patron Interpréteur

- Correspondance de la grammaire de l'exemple en appliquant le patron



Patron Interpréteur

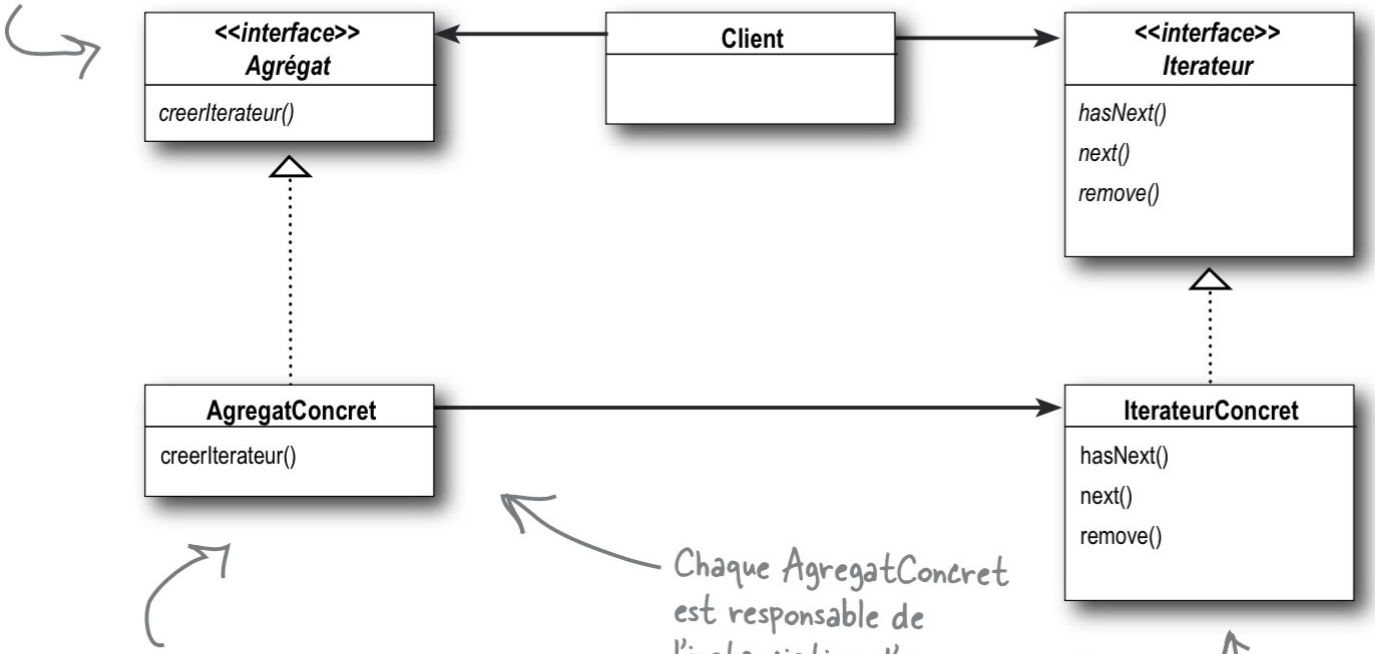
- Comment fonctionne l'interprétation
  - appel de la méthode *interpreter()* sur chaque type d'expression
  - la méthode reçoit en argument un *contexte* qui contient le flot d'entrée que nous analysons
  - la méthode apparie cette entrée et l'évalue

Patron Iterateur

- Problème/contexte
  - Comment parcourir les éléments d'un agrégat sans exposer l'implémentation sous-jacente ?
  - Comment ne pas donner cette responsabilité à l'agrégat ?
- Solution
  - **Fournir un moyen d'accès en séquence à un objet de type agrégat sans révéler sa représentation sous-jacente**
- Implantations connues dans le *core* de Java
  - `Toutes les implémentations de java.util.Iterator`
  - `Toutes les implémentations de java.util.Scanner`
  - `Toutes les implémentations de java.util.Enumeration`

Patron itérateur

Disposer d'une interface commune pour les agrégats est pratique pour votre client : elle découple la collection d'objets de son implémentation.



Cette interface est celle que tous les itérateurs doivent implémenter, et elle fournit un ensemble de méthodes qui permettent de parcourir les éléments d'une collection.

Ici, il s'agit de l'interface java.util.Iterator. Si vous ne voulez pas l'utiliser, vous pouvez toujours créer la vôtre.

L'AgregatConcret a une collection d'objets et implémente la méthode qui retourne un itérateur pour cette collection.

Chaque AgregatConcret est responsable de l'instanciation d'un IterateurConcret qui peut parcourir sa collection d'objets.

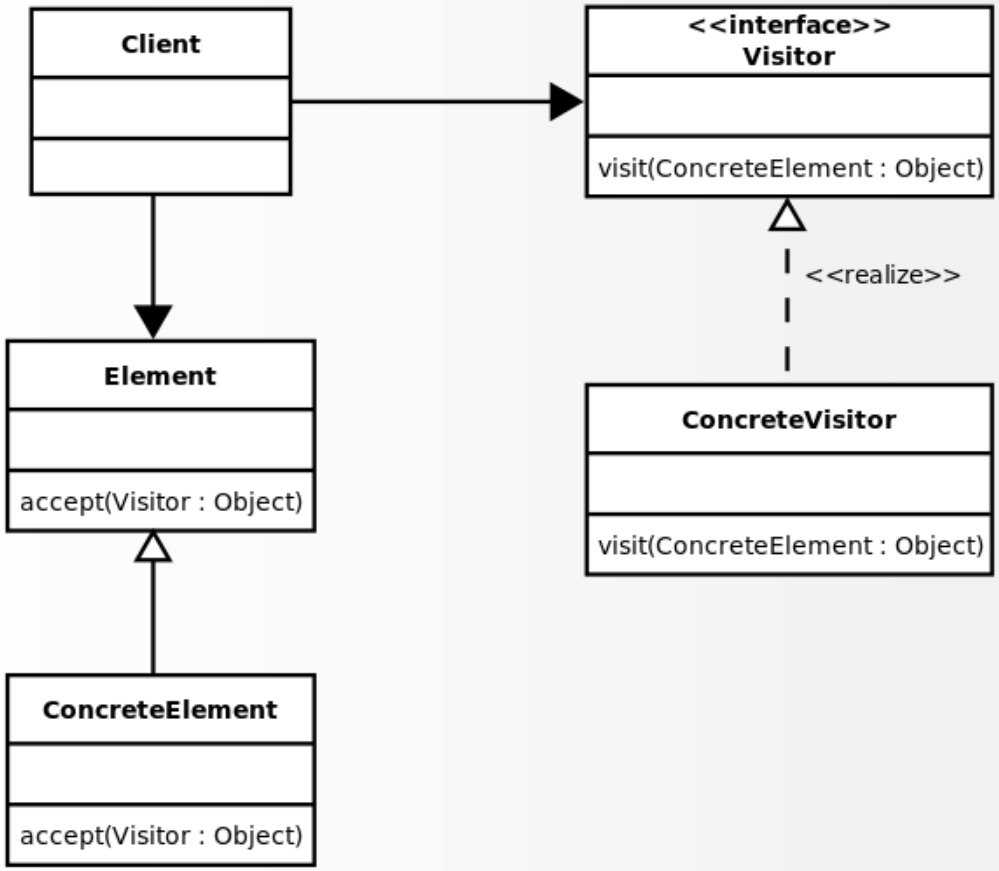
L'IterateurConcret est responsable de la gestion de la position courante de l'itération.

Patron Visiteur

- Problème/contexte
  - on a un ensemble de classes fermé (par exemple fourni par un tiers) et l'on veut effectuer un nouveau traitement sur ces classes sans avoir à les modifier
- Solution
  - **chaque classe « visitable » doit avoir une méthode publique « accepter » prenant comme argument un objet du type « visiteur »**
  - **la méthode « accepter » appellera la méthode « visite » de l'objet du type « visiteur » avec pour argument l'objet visité**
  - **ainsi, un objet visiteur pourra connaître la référence de l'objet visité et appeler ses méthodes publiques pour obtenir les données nécessaires au traitement à effectuer**
- Implantations connues dans le *core* de Java
  - `javax.lang.model.element.AnnotationValue` et `AnnotationValueVisitor`
  - `javax.lang.model.element.Element` et `ElementVisitor`
  - `javax.lang.model.type.TypeMirror` et `TypeVisitor`

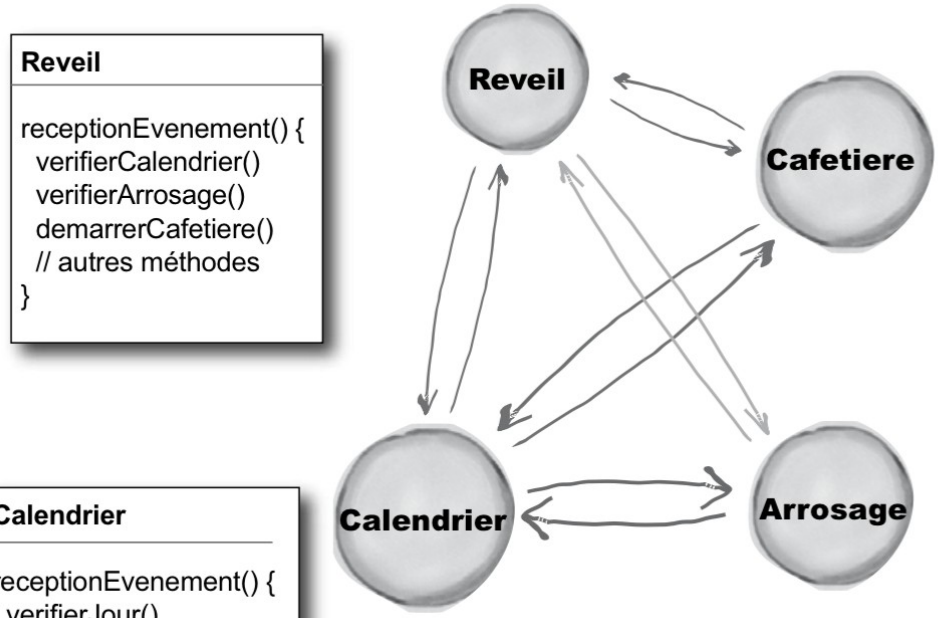
Patron Visiteur

- Modélisation UML



Patron Médiateur

- Problème : comment mémoriser facilement qui interagit avec qui et quand



Reveil

```
receptionEvenement() {
  verifierCalendrier()
  verifierArrosage()
  demarrerCafetiere()
  // autres méthodes
}
```

Cafetiere

```
receptionEvenement() {
  verifierCalendrier()
  verifierReveil()
  // autres méthodes
}
```

Calendrier

```
receptionEvenement() {
  verifierJour()
  programmerArrosager()
  programmerCafetiere()
  programmerReveil()
  // autres méthodes
}
```

Arrosage

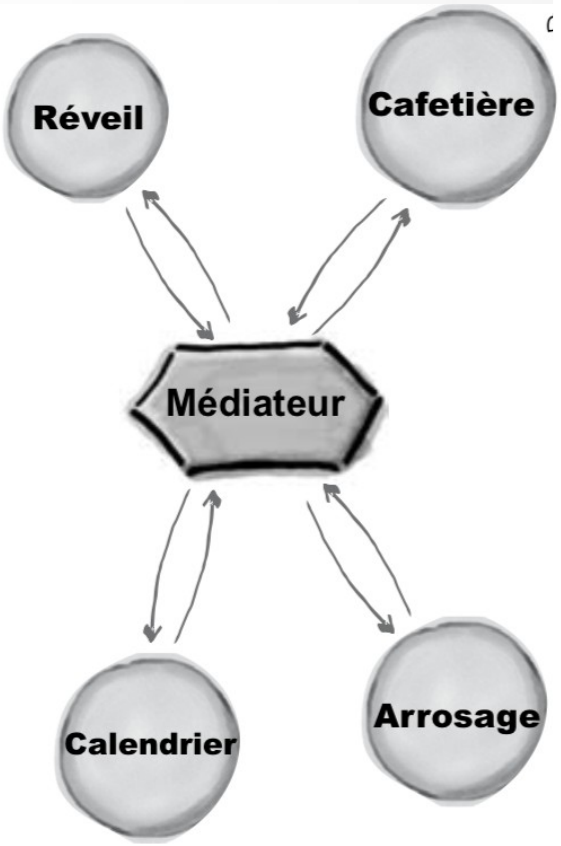
```
receptionEvenement() {
  verifierCalendrier()
  verifierTemperature()
  verifierPluie()
  // autres méthodes
}
```

Patron Mediateur

- Problème/contexte
  - les traitements et données sont répartis entre de nombreuses classe et le problème de communication entre celles-ci peut devenir complexe
  - plus les classes dépendent des méthodes des autres classes plus l'architecture devient complex (impacts sur lisibilité code et maintenabilité dans le temps)
- Solution
  - **ajouter un médiateur ayant connaissance des interfaces des autres classes => lorsqu'une classe désire interagir avec une autre, elle doit passer par le médiateur qui se chargera de transmettre l'information à la ou les classes concernées**
  - **Le médiateur fournit une interface unifiée pour un ensemble d'interfaces d'un sous-système => il réduit le couplage entre plusieurs classes**
- Implantations connues dans le *core* de Java
  - `java.util.Timer (all scheduleXXX() methods)`

Patron Médiateur

- Retour à l'exemple



Mediateur

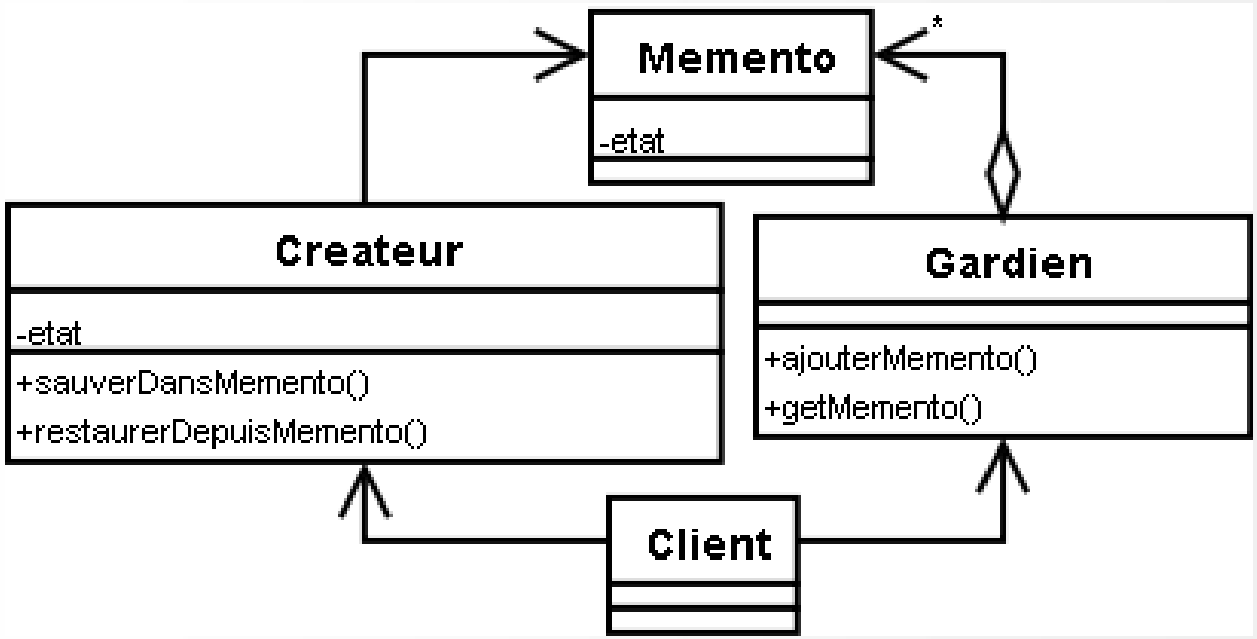
```
if(evenementRecu), {
  verifierCalendrier()
  checkShower()
  checkTemp()
}
if(weekend) {
  verifierMeteo()
  // autres méthodes
}

if(Encombrants) {
  reprogrammerReveil()
  // autres méthodes
}
```

Patron Memento

- Problème/contexte
  - Comment restaurer l'un des états précédents d'un objet, par exemple si l'utilisateur demande une « annulation » ?
- Solution
  - **le mémento est utilisé par deux objets : le créateur et le gardien**
    - **le créateur est un objet ayant un état interne (état à sauvegarder)**
    - **le gardien agira sur le créateur de manière à conserver la possibilité de revenir en arrière => demande au créateur, lors de chaque action, un objet mémento qui sauvegarde l'état de l'objet créateur avant la modification**
  - **Pour ne pas casser le principe d'encapsulation, l'objet mémento est opaque (le gardien ne peut pas le modifier)**
- Implantations connues dans le *core* de Java
  - *`java.util.Date (the setter methods do that, Date is internally represented by a long value)`*
  - *`All implementations of java.io.Serializable`*

Patron Memento

- Modélisation UML



Qui fait quoi ?

- Décorateur
- État
- Itérateur
- Façade
- Stratégie
- Proxy
- Fabrication
- Adaptateur
- Observateur
- Patron de méthode
- Composite
- Singleton
- Fabrique Abstraite
- Commande

- Enveloppe un objet et fournit une interface différente pour y accéder
- Les sous-classes décident de la façon d'implémenter les étapes d'un algorithme
- Les sous-classes décident quelles sont les classes concrètes à créer
- Garantit qu'un objet et un seul est créé
- Encapsule des comportements interchangeables et utilise la délégation pour décider lequel utiliser
- Les clients traitent les collections d'objets et les objets individuels de la même manière
- Encapsule des comportements basés sur des états et utilise la délégation pour permuter ces comportements
- Fournit un moyen de parcourir une collection d'objets sans exposer son implémentation
- Simplifie l'interface d'un ensemble de classes
- Enveloppe un objet pour fournir un nouveau comportement
- Permet à un client de créer des familles d'objets sans spécifier leurs classes concrètes
- Permet de notifier des changements d'état à des objets
- Enveloppe un objet et en contrôle l'accès
- Encapsule une requête sous forme d'objet

Plan

- Rappel bases OO
- Premiers Principes OO
- Grands principes OO
- Divers
  - Approches de développement
  - Pratiques de programmation
- Design Patterns OO
- **Restes**

Anti-patrons

- Les anti-patterns sont des designs patterns inversés: ce sont les MAUVAISES solutions les plus courantes aux problèmes
- Ces pièges dangereux doivent être reconnus et évités !
- Exemples (cf. [WIKIPEDIA])
  - Abstraction inverse
  - Action à distance
  - Ancre de bateau
  - Attente active
  - Interblocages et famine
  - Erreur de copier/coller
  - Programmation spaghetti
  - Réinventer la roue
  - Coulée de lave
  - Surcharge des interfaces
  - L'objet divin

Code smells

- code smells = mauvaises odeurs
  - mauvaises pratiques de conception logicielle qui conduisent à l'apparition de défauts
    https://fr.wikipedia.org/wiki/Code_Smell
    https://blog.codinghorror.com/code-smells/
- Code smells et refactoring associés
  - http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf

Exemples à suivre ou ne pas suivre

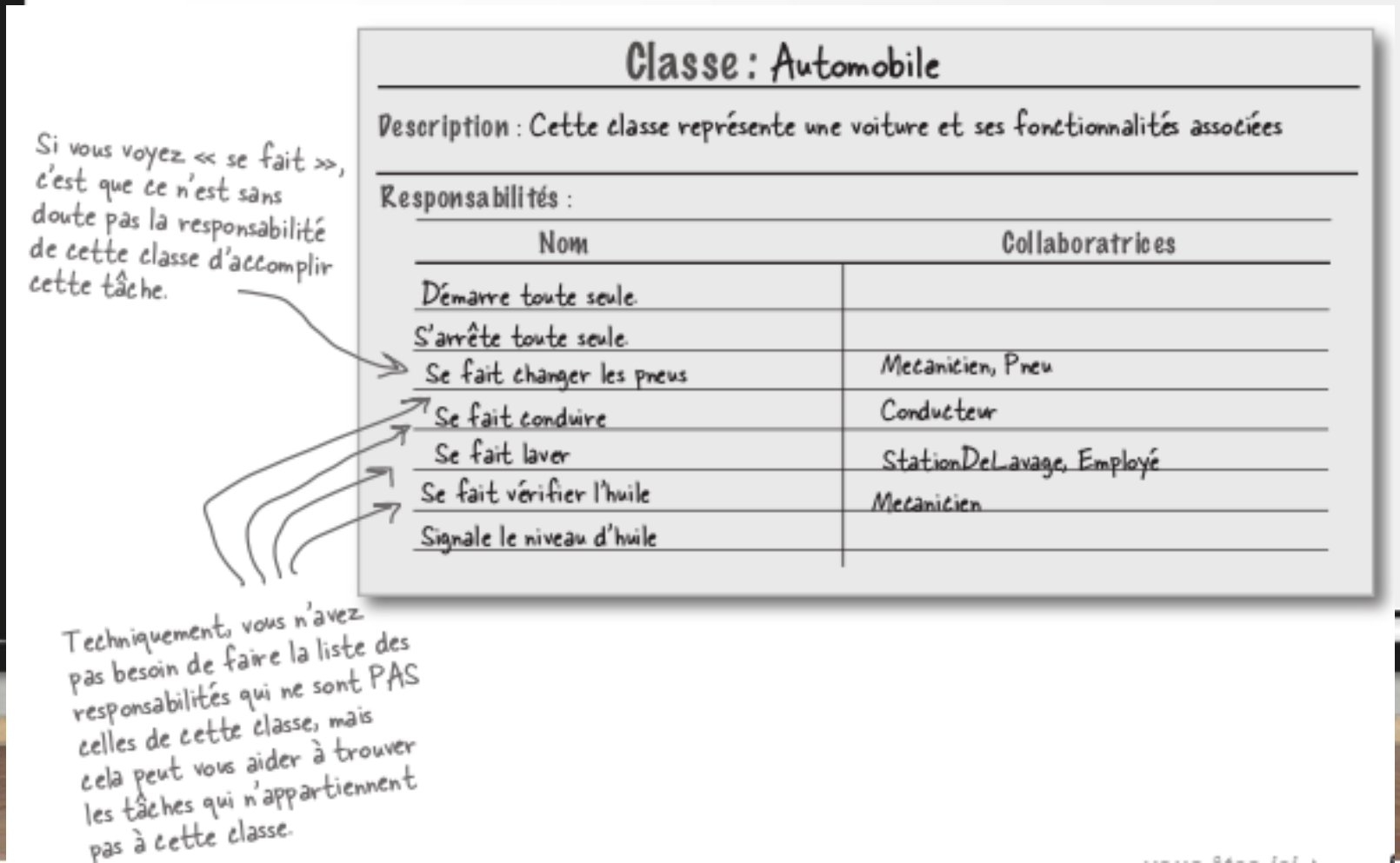
- Bonnes sources pour des petits exos d'examen !
  - http://fr.slideshare.net/mariosangiorgio/clean-code-and-code-smells
  - http://fr.slideshare.net/arturoherrero/clean-code-8036914

Autres Patrons

- Patrons parfois associés aux patrons GoF
  - Ici : http://en.wikipedia.org/wiki/Software_design_pattern
  - Exemple : Null Object
    - Créer une sous-classe dans une hiérarchie de classes pour éviter le cas particulier où des méthodes recevraient « null » au lieu d'une instance réelle ici de ces sous-classes
- Patrons GRASP
  - ***G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns*
  - De Craig Larman
  - Patrons plus « généraux » que ceux du GoF, plus proches des principes de conception : Controller, Creator, Indirection, Information Expert, High Cohesion, Low Coupling, Polymorphism, Protected Variations, Pure Fabrication

Les cartes CRC

- *Classe, Responsabilités, Collaborateurs*
- Aide à implémenter le principe de responsabilité unique



Améliorer le code

- Vérifier la couverture du code
- Vérifier le style
- Documenter le code
  - Uniquement Javadoc ! Rien de plus !
- Vérifier…

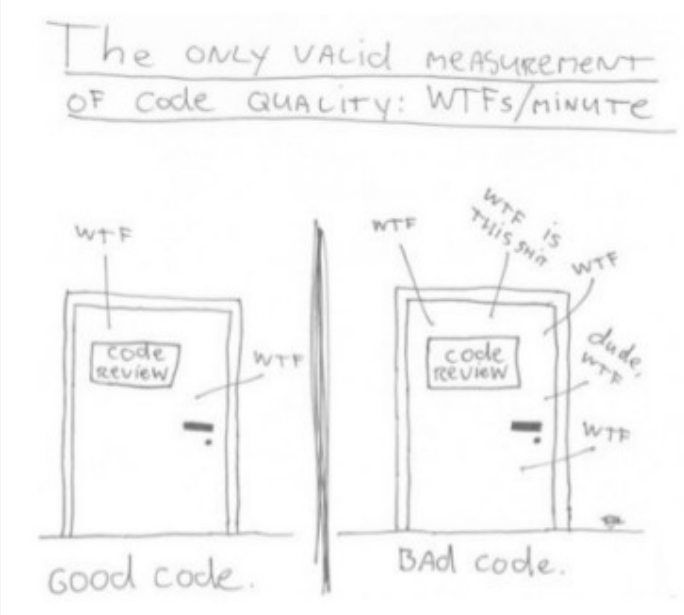
Il y a des outils pour ça ! *(ici pour Java & Eclipse)*

- Findbugs
  - Trouve les « potentiels » bugs
- PMD
  - Trouve les défauts de code, les mauvais pratiques
- Checkstyle
  - Trouve les non respects de conventions de codage
  - Attention à bien configurer les règles !
- Cobertura
  - analyse couverture du code par des tests unitaires/intégrations
- Jdepend
  - analyse des dépendances

...

… et des pratiques !

- Programmation XP
- Programmation en binôme
- TDD
- Revue de code (humaine)
- …



Questions en or

- Pourquoi le singleton n'est finalement pas conseillé ?

Bad singleton...

- C'est une sorte de variable globale…
- Il cache les dépendances du code de votre application au lieu de les exposer sous formes d'interfaces
- Rendre quelque chose global pour éviter de le passer en paramètre est mauvais signe...
- Cela viole le principe de la responsabilité unique car le singleton contrôle sa propre création et son cycle de vie
- Cela fausse et complexifie la mise en œuvre des tests unitaires (il est difficile d'isoler des classes utilisant des singletons...)