Objet et développement d'applications
Cours 3 - Pattern State

Patrons de conception pour POO

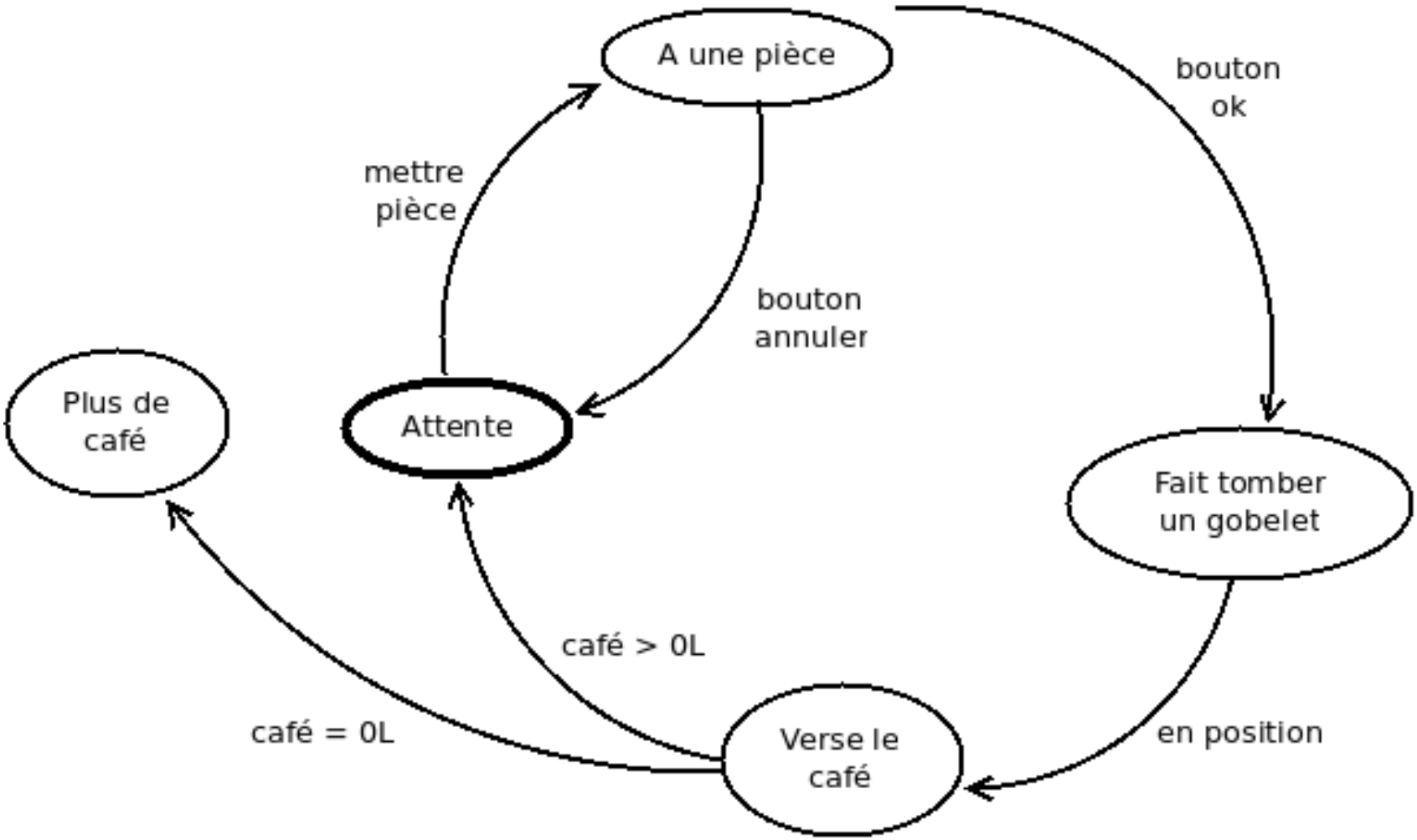
Florian Richoux

2014-2015

La machine à café

On cherche à programmer un distributeur de café.

La machine à café



Comment coder ça ?

Coder les états : 1^{ere} idée

```
final static int ATTENTE = 0;
final static int A_PIECE = 1;
final static int TOMBE = 2;
final static int VERSE = 3;
final static int VIDE = 4;

int etat = ATTENTE;
```

Comment coder ça ?

Coder les transitions : 1^{ere} idée

```
public void insererPiece()
{
  if( etat == ATTENTE )
  {
    System.out.println("Pièce␣insérée");
    etat = A_PIECE;
  }
  else if( etat == A_PIECE )
  {
    System.out.println("Déjà␣une␣pièce");
  }
  else if( etat == TOMBE || etat == VERSE )
  {
    System.out.println("Commande␣en␣cours");
  }
  else if( etat == VIDE )
  {
    System.out.println("Appareil␣vide");
  }
}
```

Comment coder ça ?

Coder le distributeur : 1^{ere} idée

```
class Distributeur
{
  final static int ATTENTE = 0;
  final static int A_PIECE = 1;
  final static int TOMBE = 2;
  final static int VERSE = 3;
  final static int VIDE = 4;

  private int etat_;
  private int doseCafe_;

  public Distributeur( int doseCafe )
  {
    doseCafe_ = doseCafe;
    if( doseCafe_ > 0 )
      etat_ = ATTENTE;
    else
      etat_ = VIDE;
  }
}
```

Et on teste

Batterie de tests

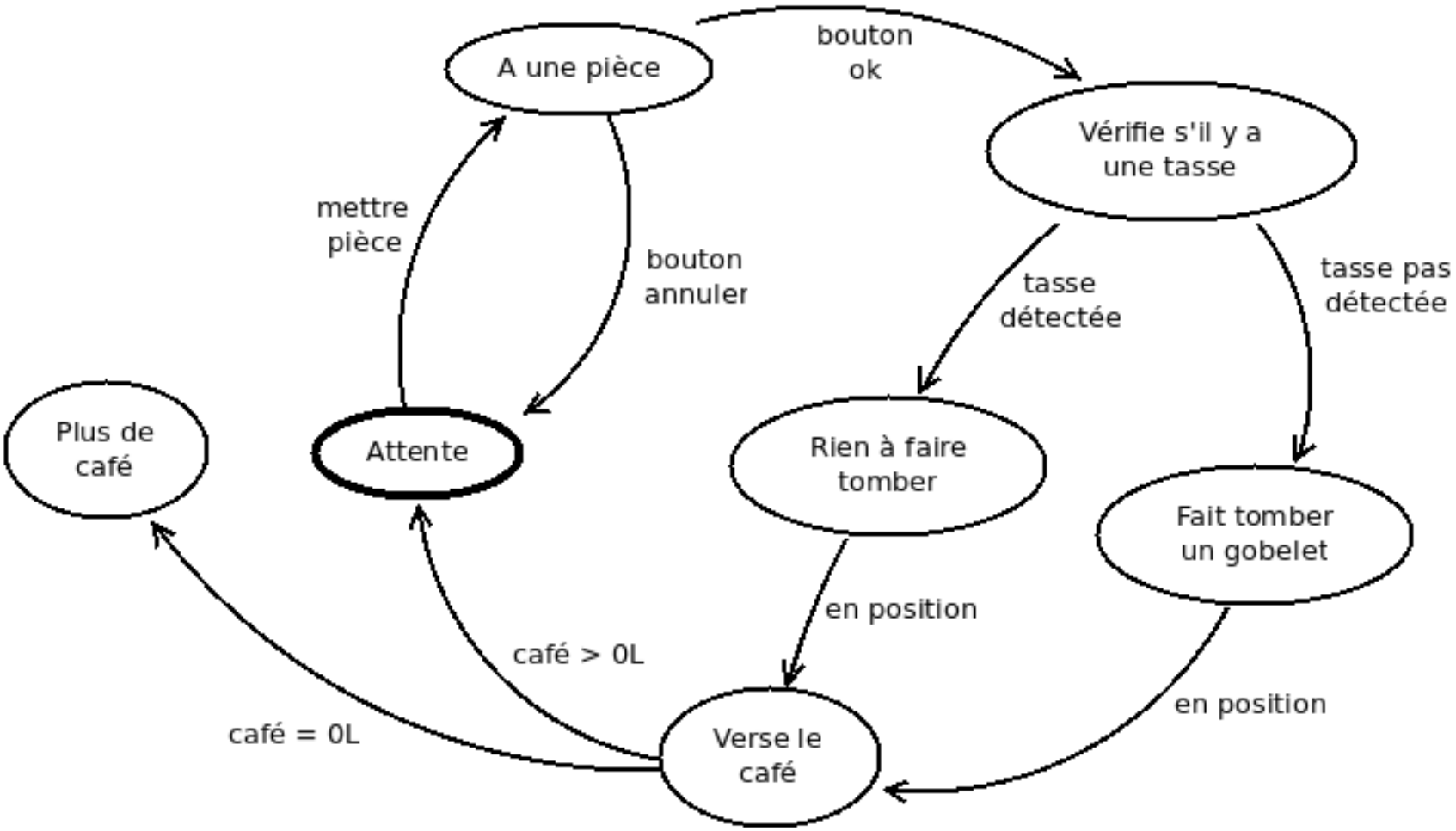
```
Distributeur distrib = new Distributeur(2);
distrib.afficherDose(); // 2

// une commande normale
distrib.insererPiece();
distrib.boutonOk();
distrib.afficherDose(); // 1

// annuler puis ok
distrib.insererPiece();
distrib.boutonAnnuler();
distrib.boutonOk();
distrib.afficherDose(); // 1

// un client shooté
distrib.boutonOk();
distrib.insererPiece();
distrib.insererPiece();
distrib.boutonOk();
distrib.boutonOk();
distrib.afficherDose(); // 0
```

Idée écolo

Nouvelle fonctionnalité

Quelqu'un a une super idée : si on met sa tasse dans la machine, elle ne donne pas de gobelet et verse directement le café.

La nouvelle machine à café



Au bord du suicide

FAIL!

- Introduire deux nouveaux états. Ça, ça va.
- Changer TOUTES les méthodes avec un nouveau else if.
- Ajouter une nouvelle méthode aussi (détection tasse).
- On pourrait imaginer des nouvelles fontionnalités qui modifirai le comportement de méthodes déjà codées (ex : café sans sucre).

Think positive

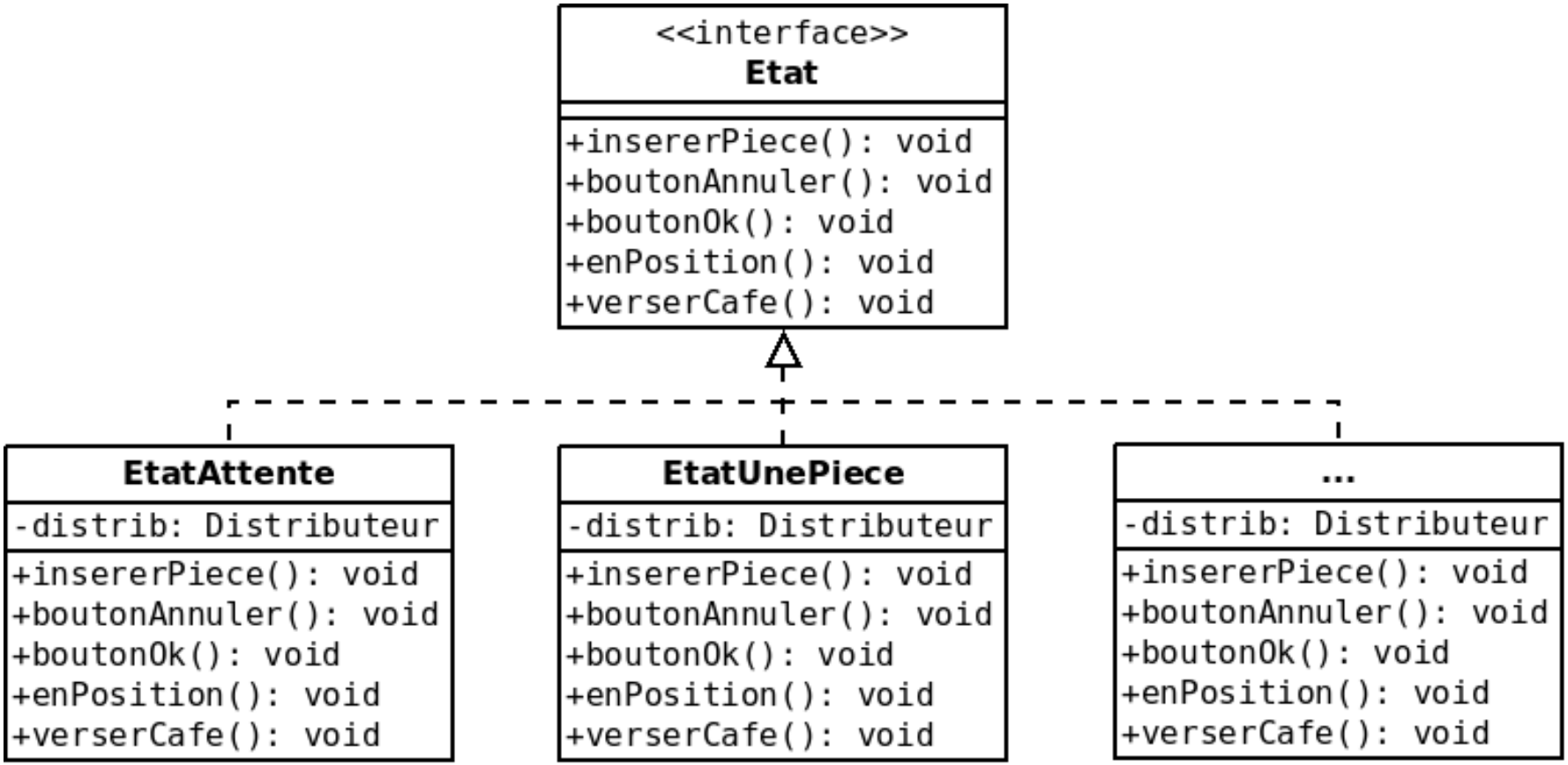
Restructuration de notre programme

- Définir une interface Etat avec une méthode pour chaque transition.
- Implémenter une classe pour chaque état.
- Supprimer les conditionnelles et faire de la composition.

Remise à zéro

Restructurons déjà le distributeur classique.

Interface Etat et ses implémentations



Classe Distributeur 1/4

```
class Distributeur
{
  // Attributs
  private Etat etatAttente_;
  private Etat etatUnePiece_;
  private Etat etatTombeGob_;
  private Etat etatVerse_;
  private Etat etatVide_;

  private Etat etat_;
  private int doseCafe_ = 0;

  ...
```

Classe Distributeur 2/4

```
class Distributeur
{
  // Constructeur
  public Distributeur(int doseCafe)
  {
    etatAttente_   = new EtatAttente (this);
    etatUnePiece_  = new EtatUnePiece(this);
    etatTombeGob_  = new EtatTombeGob(this);
    etatVerse_     = new EtatVerse   (this);
    etatVide_      = new EtatVide    (this);

    doseCafe_ = doseCafe;
    if( doseCafe_ > 0 )
      etat_ = etatAttente_;
    else
      etat_ = etatVide_;
  }
  ...
```

Classe Distributeur 3/4

```
class Distributeur
{
  // Méthodes
  public void insererPiece()
  {
    etat_.insererPiece();
  }

  public void boutonAnnuler()
  {
    etat_.boutonAnnuler();
  }

  public void boutonOk()
  {
    etat_.boutonOk();
    etat_.enPosition(); // action interne
    etat_.verserCafe(); // action interne
  }
```

Classe Distributeur 4/4

```
class Distributeur
{
  ...

  // Accesseurs get pour chaque état
  public Etat getEtatAttente()
  {
    return etatAttente_;
  }

  public Etat getEtatUnePiece()
  {
    return etatUnePiece_;
  }

  // etc
  ...
}
```

Classe EtatAttente

```
class EtatAttente implements Etat {
  private Distributeur distrib_;

  public EtatAttente(Distributeur distrib) {
    distrib_ = distrib; }

  public void insererPiece() {
    print("Pièce␣insérée");
    distrib_.setEtat( distrib_.getEtatUnePiece() ); }

  public void boutonAnnuler() {
    print("Rien␣à␣annuler"); }

  public void bontonOk() {
    print("Insérez␣d'abord␣une␣pièce"); }

  public void enPosition() {
    print("Commande␣en␣cours"); }

  public void verserCafe() {
    print("Commande␣en␣cours"); } }
```

Ce que l'on vient de faire

Nouvelle structure

Avec cette nouvelle structure, nous avons :

- encapsulé le comportement de chaque état, et ainsi isolé ces comportements de la classe cliente (ici, le distributeur).
- supprimé cette liste de if else imbriqués qui n'en finissait plus.
- laissé les états indépendants entre eux : ils ne se connaissent pas les uns les autres, notre programme reste faiblement couplé.
- fixé le comportement des états (normalement, on ne modifie pas ce qui a été écrit) tout en laissant flexible le distributeur pour de futures extensions.

Quelques bons principes de POO

Moins de classes concrètes, plus d'abstraction

Principe d'abstraction : structurer autant que possible avec des interfaces et des classes abstraites.

Moins d'héritage, plus de composition

Principe de composition : préférer la composition à l'héritage.

Moins de connaissance, plus d'indépendance

Principe de découplage : avoir des classes qui en sâche le moins possible sur les autres.

Moins de modifications, plus de souplesse

Principe ouvert-fermé : les classes doivent être ouvertes à l'extension mais fermées à la modification.

Pattern State

Nous venons de voir le pattern State (État), un autre pattern de comportement.

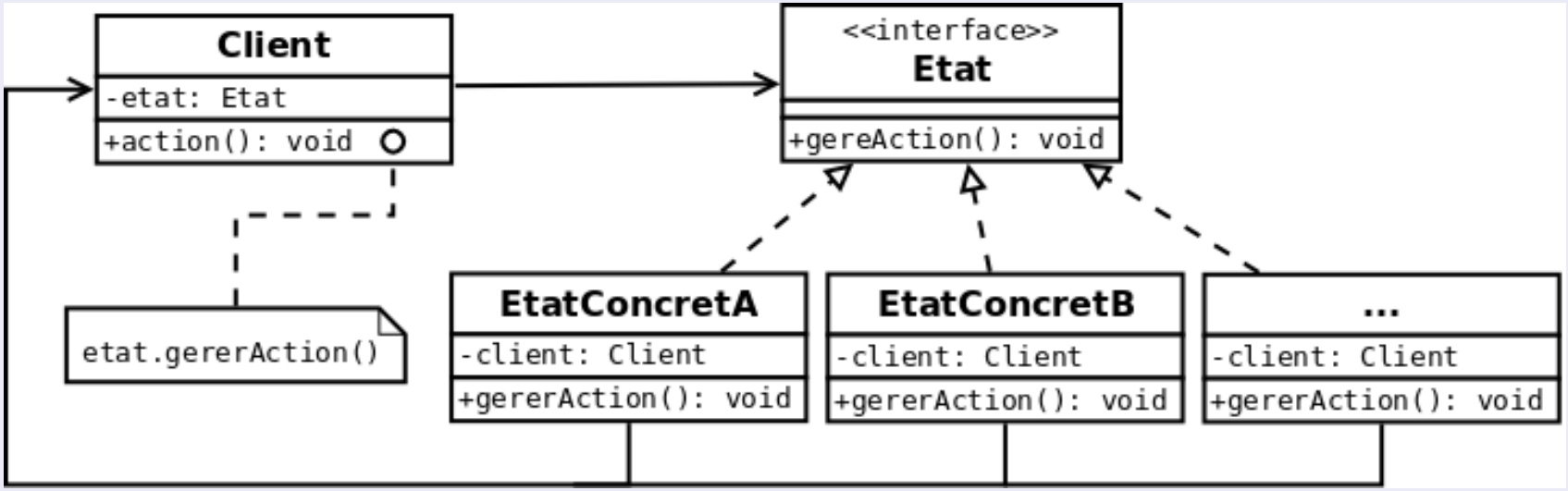
Définition

Le pattern State permet à un objet de modifier son comportement quand son état interne change. Tout se passera comme si l'objet changeait de classe.

Traduction de la 2ème phrase

Un objet qui change de comportement donne l'illusion d'être un objet d'une autre classe.

Diagramme UML



Récapitulatif

Quand utiliser le pattern State ?

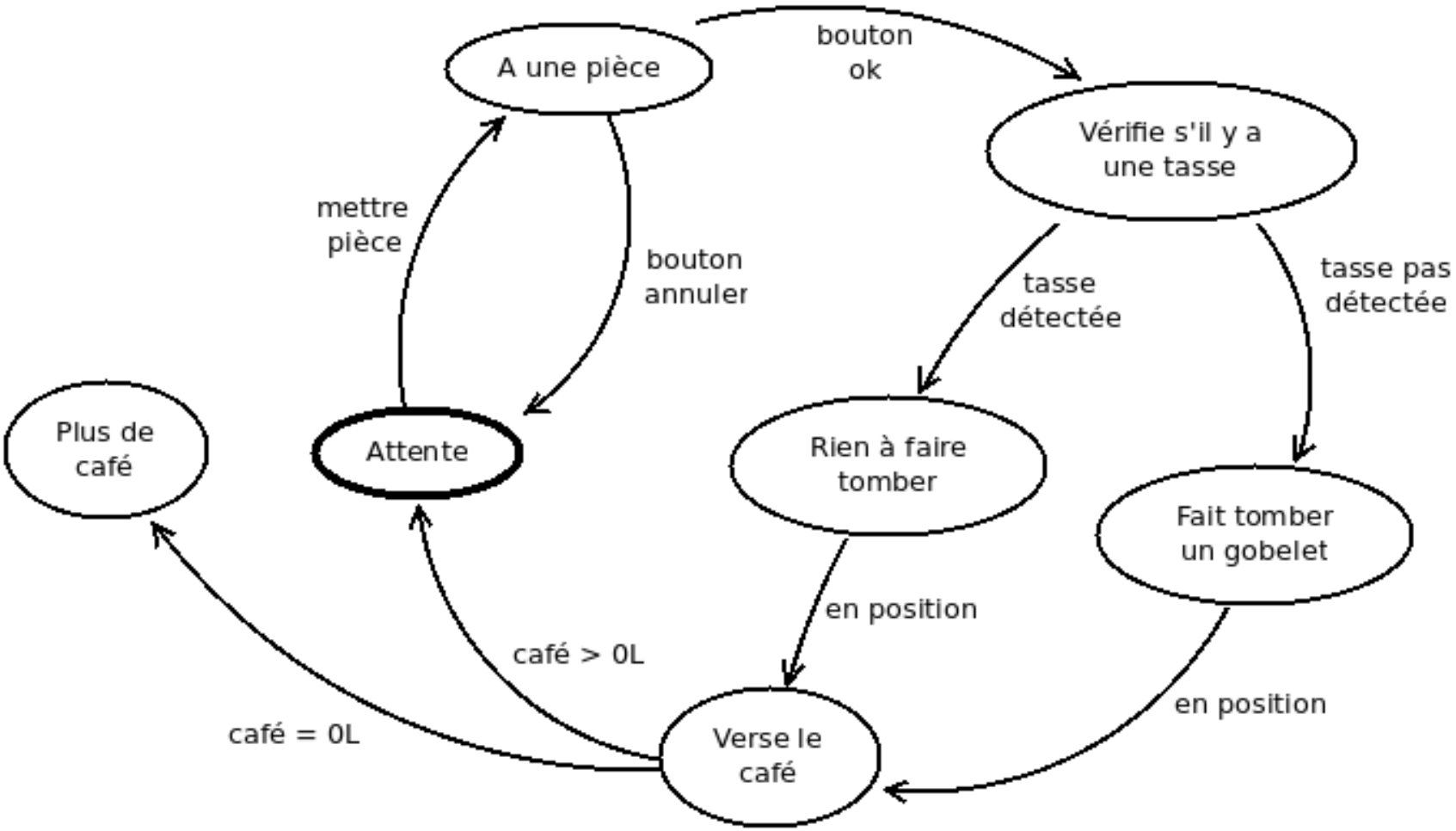
- On ne peut pas utiliser ce pattern seulement pour remplacer des if-else ou un switch.
- Il faut un changement d'état dans votre programme, et des transitions entre ces états.

Comment repérer un changement d'état ?

Il y a changement d'état si :

- un ensemble de méthodes dans votre programme doit changer de comportement en fonction d'un contexte.
- ces comportements sont mutuellement exclusifs.

Retour sur la nouvelle machine à café



Modifications 1/3

```
class Distributeur
{
  // Attributs
  private Etat etatDetecte_;
  private Etat etatTasse_;

  // Dans le constructeur
  etatDetecte_ = new EtatDetecte(this);
  etatTasse_ = new EtatTasse(this);

  public void boutonOk()
  {
    etat.boutonOk();
    etat.detecte();
    etat.enPosition();
    etat.verserCafe()
  }
  // Plus les accesseurs
}
```

Modifications 2/3

```
interface Etat
{
  public void detecte();
}

class EtatDetecte implements Etat
{
  public void detecte()
  {
    if( tassePresente )
        distrib.setEtat( distrib.getEtatTasse() );
    else
        distrib.setEtat( distrib.getEtatTombeGob() );
  }

  // Plus toutes les autres méthodes
}
```

Modifications 3/3

```
class EtatUnePiece implements Etat
{
  public void boutonOk()
  {
    distrib.setEtat( distrib.getEtatDetecte() );
  }
}

class EtatTasse implements Etat
{
  public void enPosition()
  {
    distrib.setEtat( distrib.getEtatVerse() );
  }

  // Plus toutes les autres méthodes
}
```

On peut faire encore mieux !

Java nous offre une possibilité pour encore améliorer notre code.

Laquelle ?

La semaine prochaine

On verra un dernier pattern de comportement, très différent cette fois.

Sujet projets !

Quelques trucs en C++

Fuite mémoire

Sous Linux :

```
valgrind --leak-check=full ./executable
```

Compilé avec option -g : valgrind indique où se trouve le problème !

Smart pointers (comme shared_ptr)

- Dans le standard C++11 (compilez avec un g++ récent avec l'option `-std=c++11`)
- Dans la bibliothèque Boost.

Utiliser des assert

On peut désactiver les assert avec l'option `-DNDEBUG` à la compilation.

La solution contre les segmentation faults

Compiler avec -g, utiliser `gdb` et la commande `bt` (backtrace).