CHAPITRE 2: PATRONS DU GANG OF FOUR

PATRONS DE COMPORTEMENT

Observer, Iterator, Visitor, State, Mediator, Strategy, Template Method, Chain of Responsability, Command, Interpreter, Memento

Behavioural Patterns

- Abstraction du comportement
  - Structure algorithmique
  - Affectation de responsabilités aux objets
  - Communication entre objets : permettent donc de se dégager du problème du flots de contrôle et de se concentrer sur les relations entre objets.
- Niveau classe
  - Utilisation de l'héritage
  - Répartition du comportement
  - Template Method, Interpreter
- Niveau objet
  - Utilisation de la composition
  - Coopération d'objets pour effectuer une tâche (Mediator, Chain of Responsability, Observer)
  - Encapsulation du comportement dans un objet auquel sont déléguées les requêtes.

PATRONS DE COMPORTEMENT

Observer

Abonnement à un journal

- Un éditeur se lance dans les affaires et commence à diffuser des journaux (ou des magazines).
- Vous souscrivez un abonnement.
- Chaque fois qu'il y a une nouvelle édition, vous la recevez.
- Tant que vous êtes abonné, vous recevez de nouveaux journaux.
- Quand vous ne voulez plus de journaux, vous résiliez votre abonnement. On cesse alors de vous les livrer.
- Tant que l'éditeur reste en activité, les particuliers, les hôtels, les compagnies aériennes, etc., ne cessent de s'abonner et de se désabonner.

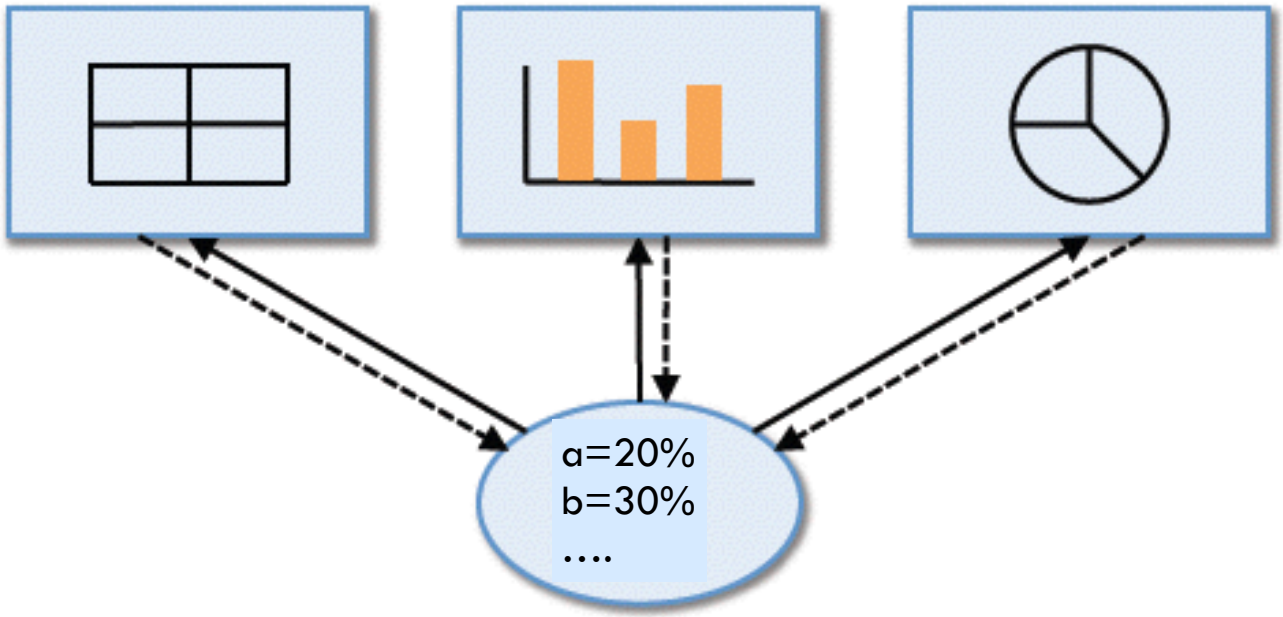
Abonnement à un journal

- Un éditeur se lance dans les affaires et commence à diffuser des journaux (ou des magazines).
- Vous souscrivez un abonnement.
- Chaque fois qu'il y a une nouvelle édition, vous la recevez.
- Tant que vous êtes abonné, vous recevez de nouveaux journaux.
- Quand vous ne voulez plus de journaux, vous résiliez votre abonnement. On cesse alors de vous les livrer.
- Tant que l'éditeur reste en activité, les particuliers, les hôtels, les compagnies aériennes, etc., ne cessent de s'abonner et de se désabonner.

C'est l'essentiel du pattern Observateur, sauf que nous appelons l'éditeur le SUJET et les abonnés les OBSERVATEURS.

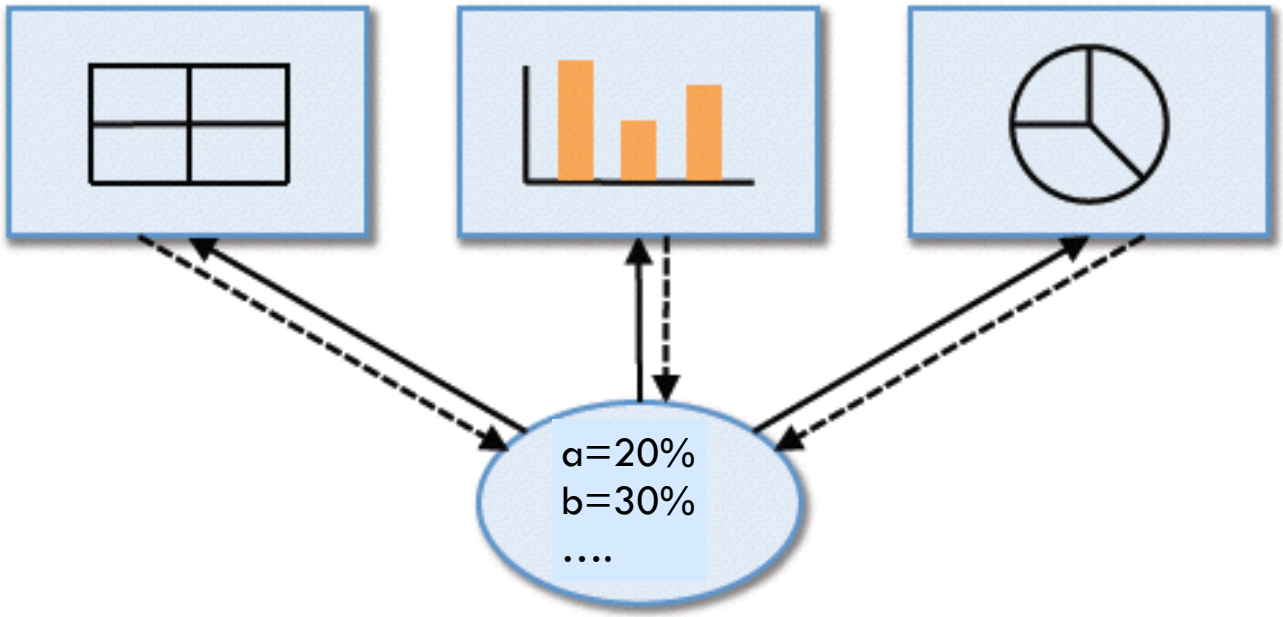
Notification du changement

- Pour des données sources (ex. statistique de visites de votre page web), vous souhaitez avoir plusieurs représentations.
- Comment mettre à jour vos représentations à chaque fois que les données changent?
  - Souscrire les formes de représentations auprès du système de gestion des données
  - Notifier les représentations du changement d'état des données



Notification du changement

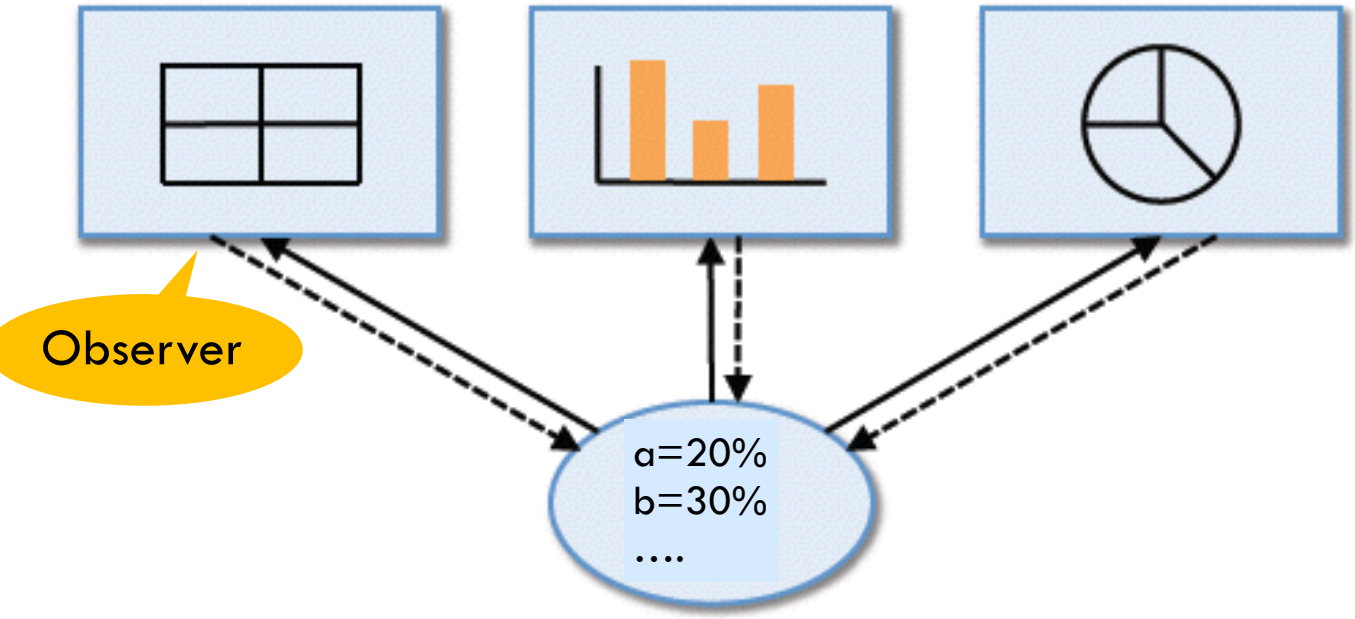
- Pour des données sources (ex. statistique de visites de votre page web), vous souhaitez avoir plusieurs représentations.
- Comment mettre à jour vos représentations à chaque fois que les données changent?
  - Souscrire les formes de représentations auprès du système de gestion des données
  - Notifier les représentations du changement d'état des données



Le pattern Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

Notification du changement

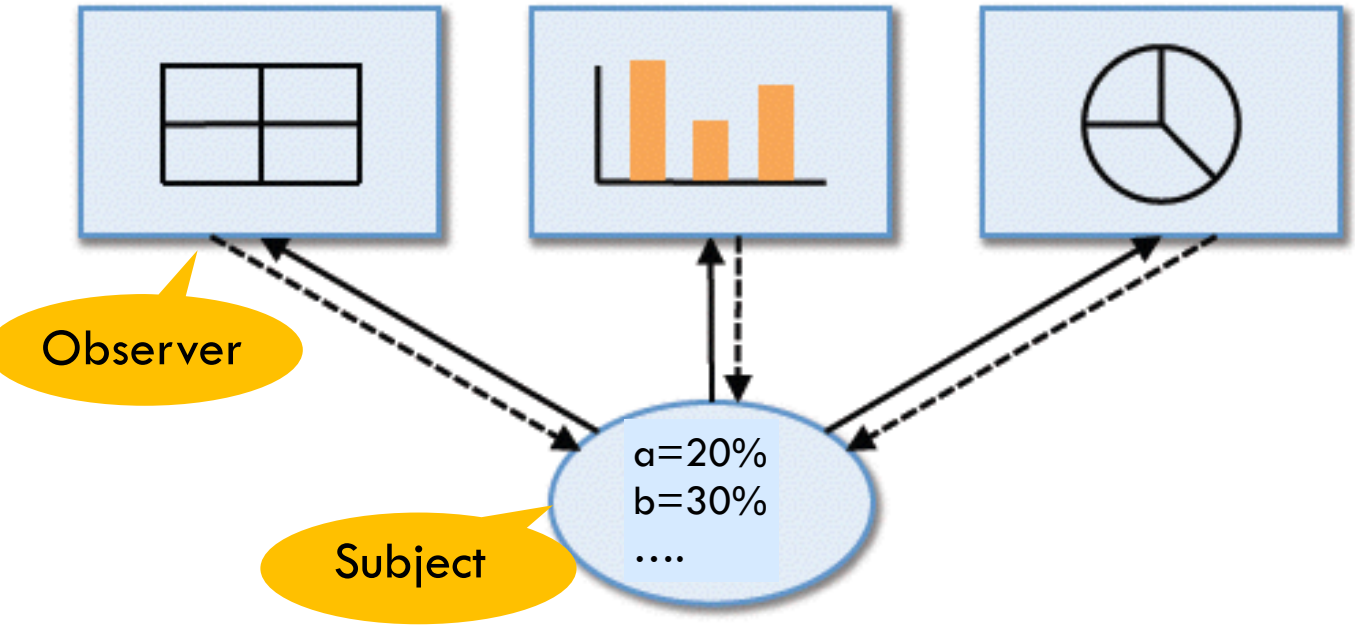
- Pour des données sources (ex. statistique de visites de votre page web), vous souhaitez avoir plusieurs représentations.
- Comment mettre à jour vos représentations à chaque fois que les données changent?
  - Souscrire les formes de représentations auprès du système de gestion des données
  - Notifier les représentations du changement d'état des données



Le pattern Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

Notification du changement

- Pour des données sources (ex. statistique de visites de votre page web), vous souhaitez avoir plusieurs représentations.
- Comment mettre à jour vos représentations à chaque fois que les données changent?
  - Souscrire les formes de représentations auprès du système de gestion des données
  - Notifier les représentations du changement d'état des données



Le pattern Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

Pattern Observer

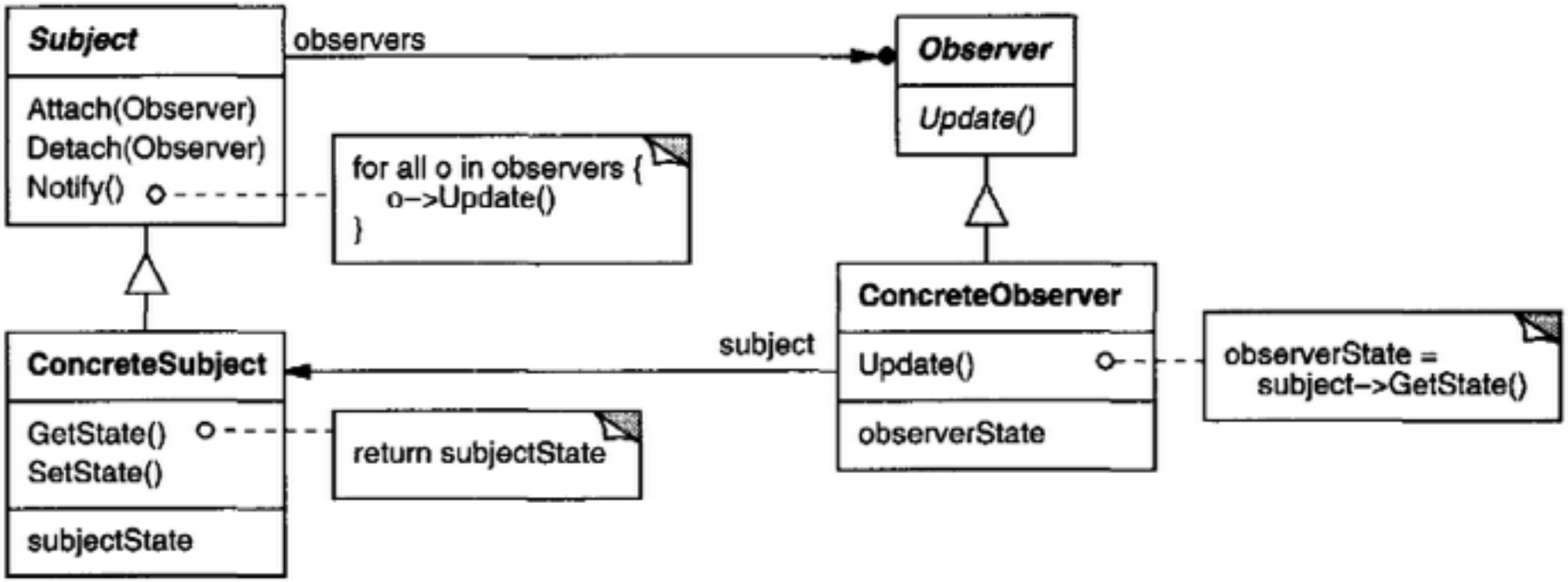
- Intention
  - Créer une dépendance un–à–plusieurs entre des objets de sorte que lorsque un objet change d'état, tous ses dépendants sont notifiés et mis à jour
- Synonymes
  - Dependents (Dépendants), Publish-Subscribe (Diffusion-souscription), Observateur, Modèle de délégation d'événements
- Utilisation connus
  - Interface graphique
- Patrons associés
  - Singleton, Mediator

Pattern Observer

- **Problème (Quand l'utiliser)**
  - Quand une abstraction a deux aspects, l'un dépendant de l'autre. L'encapsulation de ces aspects dans des objets distincts permet de les faire varier et de les réutiliser de façon indépendante.
  - Lorsque le fait de changer un objet implique d'en changer d'autres, sans que l'on sache combien d'autres objets doivent être changés.
  - Quand un objet doit être en mesure d'informer d'autres sans faire de supposition sur la nature de ces autres objets. En d'autres termes, vous ne voulez pas que ces objets soit fortement couplés.

Pattern Observer

- Structure

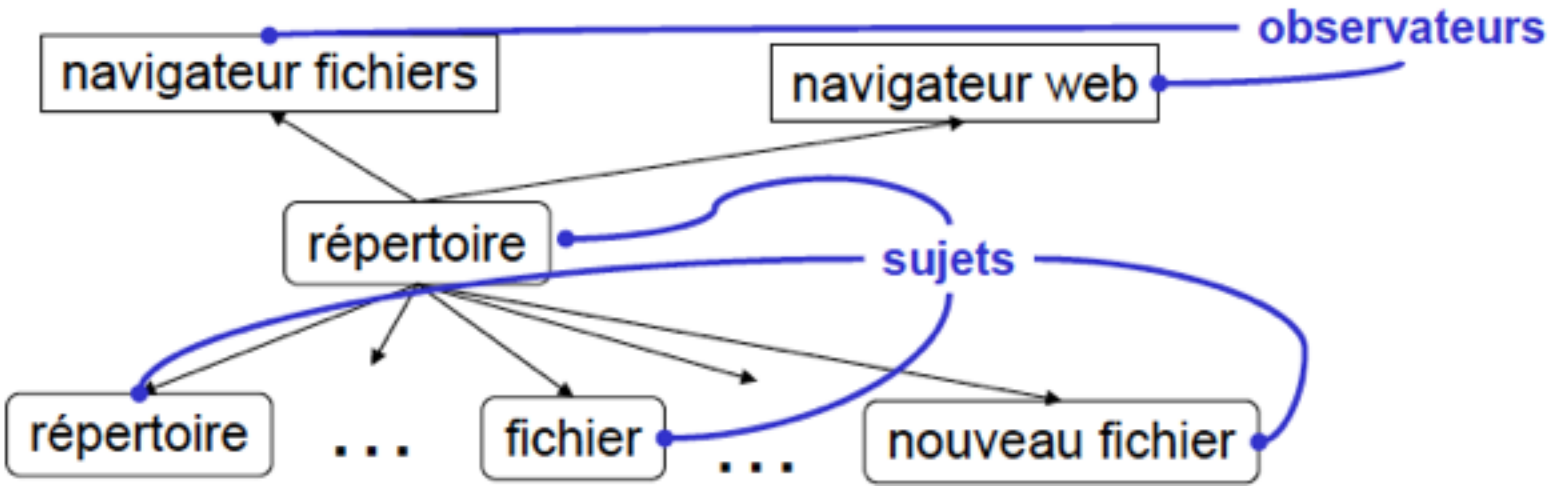


Pattern Observer

- **Participants**
  - Subject
    - Garde une trace de ses observateurs
    - Propose une interface pour ajouter ou enlever des observateurs
  - Observer
    - Définit une interface pour les objets observateur qui comporte une méthode update qui sera appelée lors de changements du sujet
  - ConcreteSubject
    - C'est l'objet observé
    - Enregistre son état intéressant les observateurs
    - Envoie une notification à ces observateurs quand il change d'état
  - ConcreteObserver
    - Il maintient une référence à l'objet observé
    - Il implémente l'interface de mise à jour permettant à son état de rester consistent avec celui de l'objet observé

Exemple : Avertissement de changement

- Identifier les classes qui joueront les rôles de sujets et d'observateurs dans le système de fichier.



- Les répertoires seront observés par les observateurs qui seront intéressés à être avertis de changements. Les répertoires sont donc les sujets. On peut étendre cette définition aux fichiers et, de manière générale, aux noeuds du système de fichiers.
- Les éléments observateurs seront par exemple les navigateurs de fichiers ou toute autre entité qui veulent être avertis automatiquement de changements au système de fichiers.

Exemple


```
public interface Observer {
public void update();
}
public abstract class Subject {
private List<Observer> observers;
public void attach(Observer o) {
observers.add(o); }
public void detach(Observer o) {
observers.remove(o);}
public void notifyObservers() {
for (Observer o: observers) {
o.update();}
}

public class Timer extends Subject {
public int getHours() { ... }
public int getMinutes() { ... }
public int getSeconds() { ... }
public void tick() {
// update internal state
notifyObservers();
}}
```

```
public class DigitalClock implements
Observer {
private Timer timer;
public DigitalClock(Timer timer) {
this.timer = timer;
timer.attach(this);
}
protected void finalize() throws Throwable {
timer.detach(this);
}
public void update() {
int hours = timer.getHours();
int minutes = timer.getMinutes();
int seconds = timer.getSeconds();
// update display
}
}
public class AnalogClock implements
Observer {…}
```

Pattern Observer

- **Conséquences**
  - Couplage minimal entre le sujet et l'observateur
  - On peut réutiliser les sujets sans leurs observateurs et vice-versa
  - Tout ce que le sujet connaît c'est la liste de ses observateurs
  - Le sujet n'a pas besoin de connaître la classe concrête de ses observateurs
  - Des observateurs peuvent être ajoutés/enlevés dynamiquement (à n'importe quel moment)
    - Modularité: Le sujet et les observateurs peuvent varier de façon indépendante.
    - Extensibilité: On peut définir et ajouter autant d'observateurs que nécessaire.
    - Adaptabilité: Différents observateurs fournissent différentes vues du Subject
  - Mises à jour inattendues: les Observers ne se connaissent pas
  - Coût de la mise à jour: certains Observers peuvent avoir besoin d'indices sur ce qui a changé

Pattern Observer

- **Implémentation**
  - Correspondance Sujet-Observateur… il y a plusieurs façon de l'implanter: Array, linked list
  - Observer plus d'un sujet: le sujet informe l'observateur qui il est via l'interface update
  - Qui déclenche la mise à jour?
    - Le sujet à chaque fois qu'il change d'état
    - Le Client
    - Les observateurs après qu'ils causent un ou plusieurs changements d'état
  - références à des sujets supprimés
  - S'assurer que l'état du sujet est cohérent (mis à jour) avant la notification
  - Éviter des protocoles de mise à jour spécifiques à un Observer
  - Combien d'informations sur le changement le sujet devrait envoyer aux observateurs?
    - Beaucoup : Push Model (modèle « tirer »)
    - Très peu : Pull Model (modèle « pousser »)
  - Spécifier explicitement l'intérêt à des modifications (évènements) particulières: publish-subscribe

Pattern Observer

- **Implémentation**
  - Un Observateur peut être aussi un Sujet
  - Encapsulation de mise à jour complexes
    - Exemple: notifier un observateur après la mise à jour de plusieurs sujets
    - Utilisez un objet intermédiaire qui agit comme un médiateur (Change-Manager)
    - Les sujets envoient les notifications à l'objet médiateur qui effectue les traitements nécessaires avant de notifier les observateurs
  - Combinaison des classes Subject et Observer (ex. SmallTalk)
  - Java fournit les classes Observable/Observer comme support pour coder ce patron.
    - La classe java.util.Observable est la classe de base pour coder le sujet. Toute classe qui veut être observée étend cette classe de base.
    - L'interface java.util.Observer est l'interface qui doit être implémentée par les classes qui veulent être des observateurs.

Exemple: java.util.Observable


```
//Les observables (sujet) doivent spécialiser Observable
class Timer extends java.util.Observable {
private long zzz = 1000;
private long zero;

Timer(long zzz) {
this.zzz = zzz;
}

public void run () throws InterruptedException {
 zero = System.currentTimeMillis();
 while (true) {
   setChanged();

     //On notifie les observateurs… appel de update(…)
   notifyObservers(new Long(System.currentTimeMillis()- zero));

     TimeUnit.MILLISECONDS.sleep(zzz);
   }
 }
}
```

Exemple: java.util.Observer


```
class observ implements java.util.Observer {
        //Tous les observateurs doivent posséder //une méthode update(…)
        public void update(Observable o, Object arg) {
        System.out.println("Temps : "+ ((Timer)o).getSeconds()+ " secs");
        }
      }

      public class Test {
       Timer timer;
       public Test() {
          timer = new Timer();
          Toto toto = new Toto();
          timer.addObserver(toto); //toto est écouteur du timer
        }
       public static public void main(String[] args) {
            Test t = new Test();
            t.timer.run();
       }
      }
```

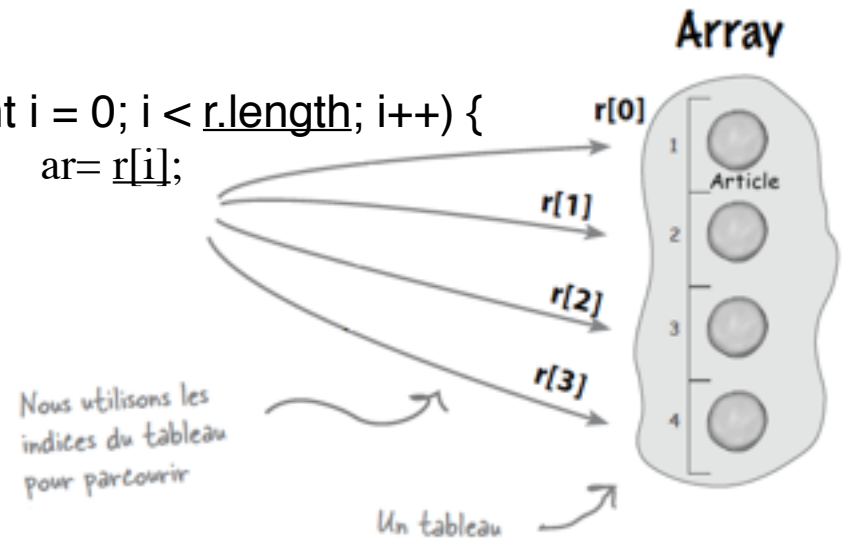
PATRONS DE COMPORTEMENT

Iterator

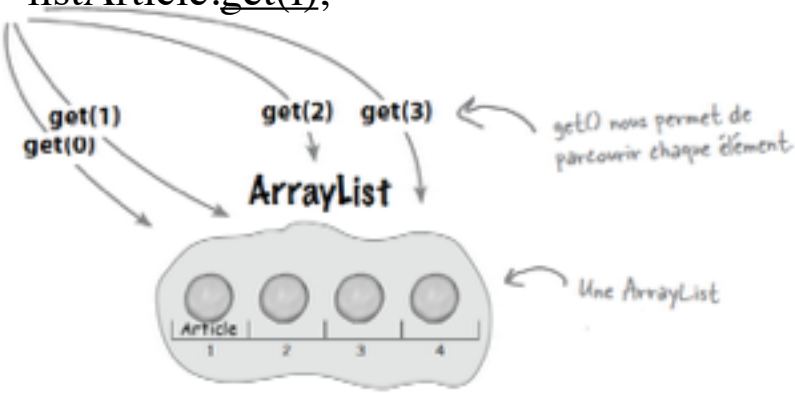
Motivation

- Fusion de deux systèmes de gestion d'articles
  - le premier géré avec un tableau
  - le second avec une ArrayList
- Pour afficher tous les articles, sans modifier le code, il faut parcourir les deux groupes

```
for (int i = 0; i < r.length; i++) {
        ar= r[i];
}
```



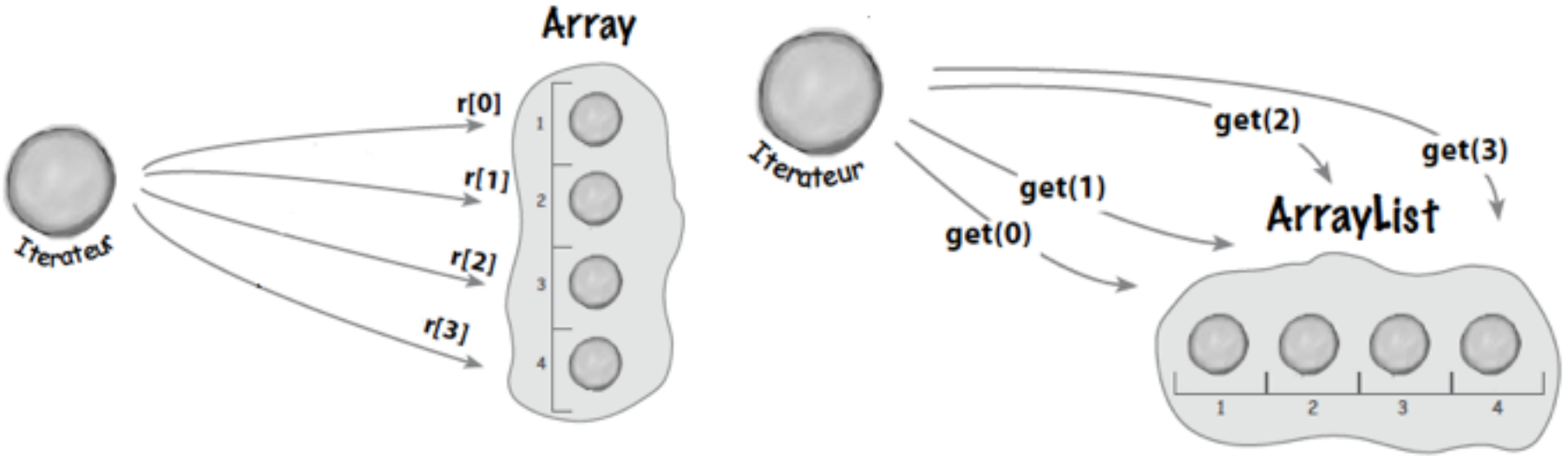

```
for (int i = 0; i < listArticle.size(); i++) {
ar = listArticle.get(i);
}
```



Encapsuler l'itération

- Et si nous créions maintenant un objet, appelons-le Iterateur, qui encapsule l'itération sur une collection d'objets ?

```
Iterateur iterateur = monSystemeArticle.creerIterateur();
while (iterateur.encore()) {
Article a = (Article)iterateur.suivant();
}
```



Définir l'intégrateur


```
public interface Iterateur{
boolean encore();
Object suivant();
}

public class IterateurSysteme1 implements Iterateur {
Article[] elements;
int position = 0;

public IterateurSysteme1(Article[] elements) {
this.elements = elements;
}
public Object suivant() {
Article a = elements[position];
position = position + 1;
return a;
}
public boolean encore() {
if (position >= elements.length || elements[position] == null) {
return false;
} else {
return true;
}
}
}
```

Pattern Iterator

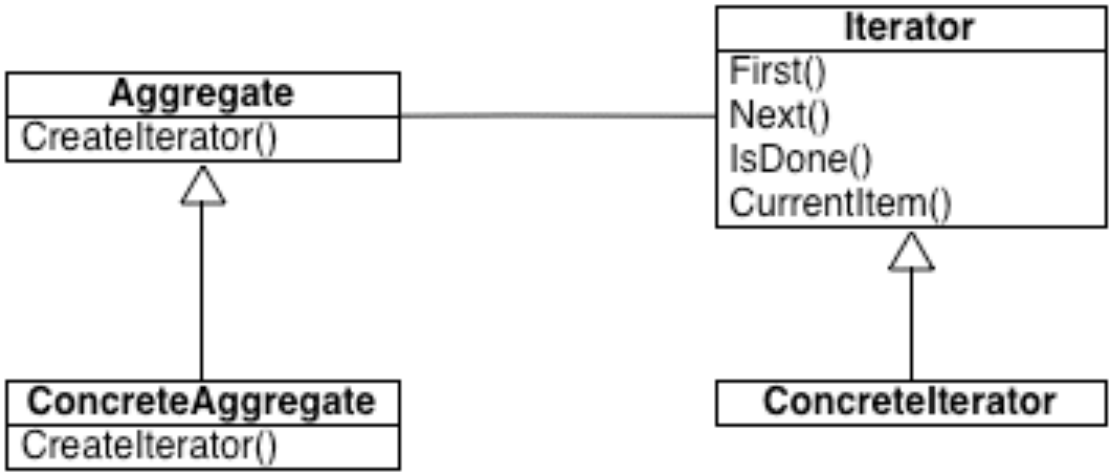
- Intention
  - Fournir un accès séquentiel aux éléments d'un agrégat (groupe) sans exposer sa structure interne
- Synonymes
  - Cursor (curseur), itérateur
- Utilisation connus
  - Itérateurs de Java, C++, ... sur toute séquence Itérable
- Patrons associés
  - Composite : itérateurs récursifs
  - Factory Method : abstrait la création des itérateurs
  - Memento : un itérateur peut utiliser un Memento pour conserver en interne l'état de l'itération.

Pattern Iterator

- **Problème (Quand l'utiliser)**
  - Accéder aux éléments d'un contenant sans en révéler la représentation interne
  - Supporter plusieurs types d'accès
  - On ne veut (peut) pas charger une représentation avec des opérations d'accès
  - Permettre plusieurs parcours simultanés (et dissociés)
  - Offrir une interface commune d'itération sur diverses représentations (polymorphes) et en conséquence une algorithmique indépendante (abstraite)

Pattern Iterator

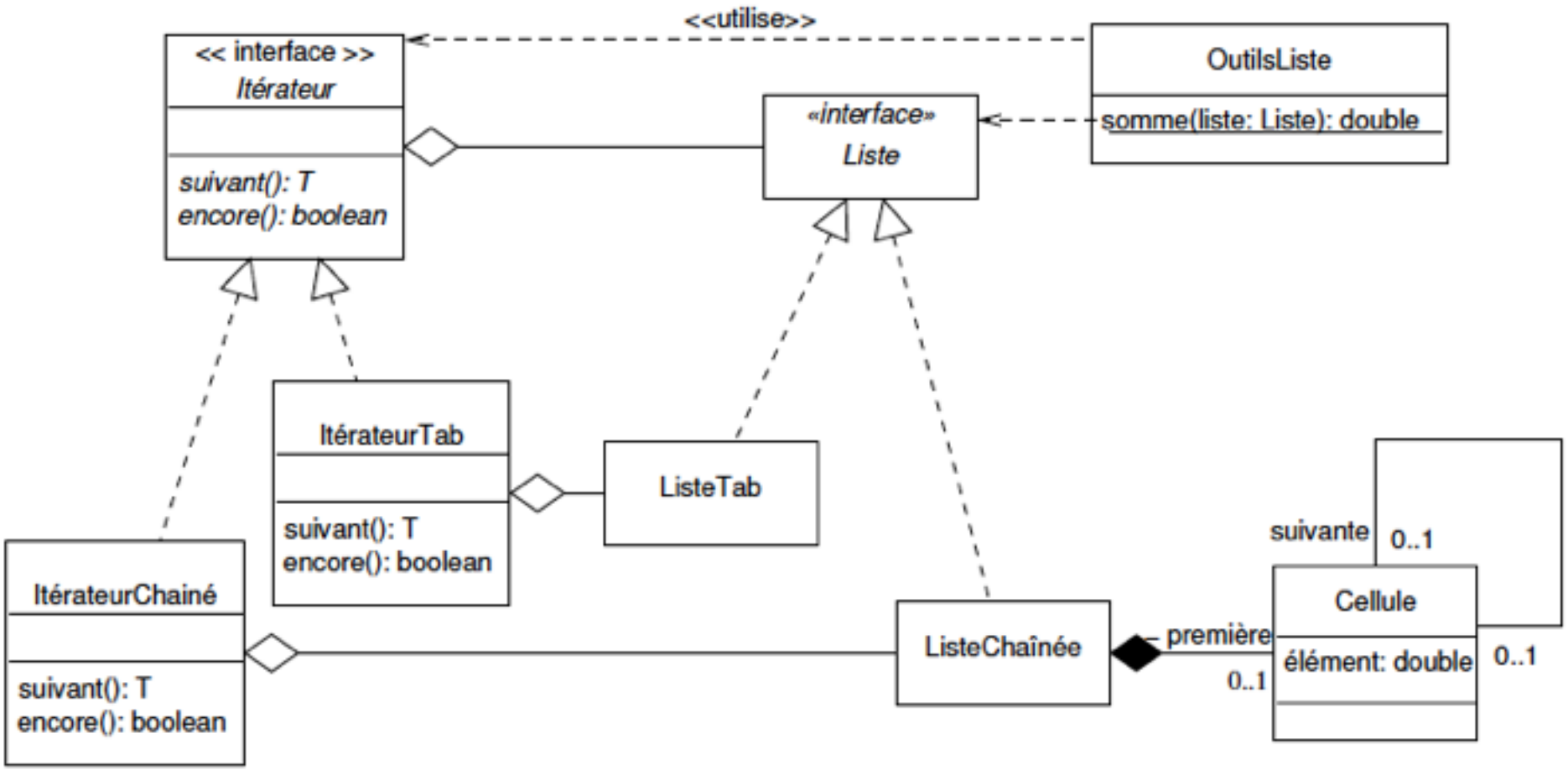
- Structure



Pattern Iterator

- **Participants**
  - **Iterator** : définit une interface pour accéder et parcourir les éléments.
  - **ConcreteIterator**
    - implémente l'interface *Iterator*
    - garde trace de la position courante dans le parcourt de l'agrégat
  - **Aggregate** : définit une interface pour créer un objet *Iterator*.
  - **ConcreteAggregate:** implémente l'interface de création de l'*Iterator* pour créer une instance de *ConcreteIterator*.

Exemple


<<utilise>>
<< interface >>
Itérateur
suivant(): T
encore(): boolean
OutilsListe
somme(liste: Liste): double
«interface»
Liste
ItérateurTab
suivant(): T
encore(): boolean
ListeTab
ItérateurChainé
suivant(): T
encore(): boolean
ListeChaînée
première
0..1
suivante
0..1
Cellule
élément: double
0..1

Exemple


```
public class ArrayList {
private Object[] data;
private int size;
public Iterator iterator() {
return new Iterator() {
private int index = 0;
public Object next() {
if (!hasNext()) {
throw new NSEException();
}
return data[index++];
}
public boolean hasNext() {
return index < size;
}
};
}
}
```

```
public interface Iterator {
public Object next();
public boolean hasNext();
}
```

Pattern Iterator

- **Implémentation**
  - Qui contrôle l'itération?
    - Le client: itérateur externe
    - L'itérateur: itérateur interne
  - Qui définit l'algorithme de traverse?
    - L'objet Aggregate (curseur)
    - L'itérateur (peut violer les principes d'encapsulation)
  - L'itérateur est-il robuste en présence de modifications du groupe?
    - Sans copier le groupe!
    - Enregistrer l'itérateur dans l'agrégat
  - Accès privilégié.
    - L'itérateur est vu comme une extension de son agrégat
    - Possibilité en C++ de définir l'intérateur comme *freind* de son agrégat
  - Itérateur nul : utile dans les structure d'arbre, ex. Composite
    - Composite:createIterator() => Iterator sur ses fils
    - Feuille: createIterator() => NullIterator

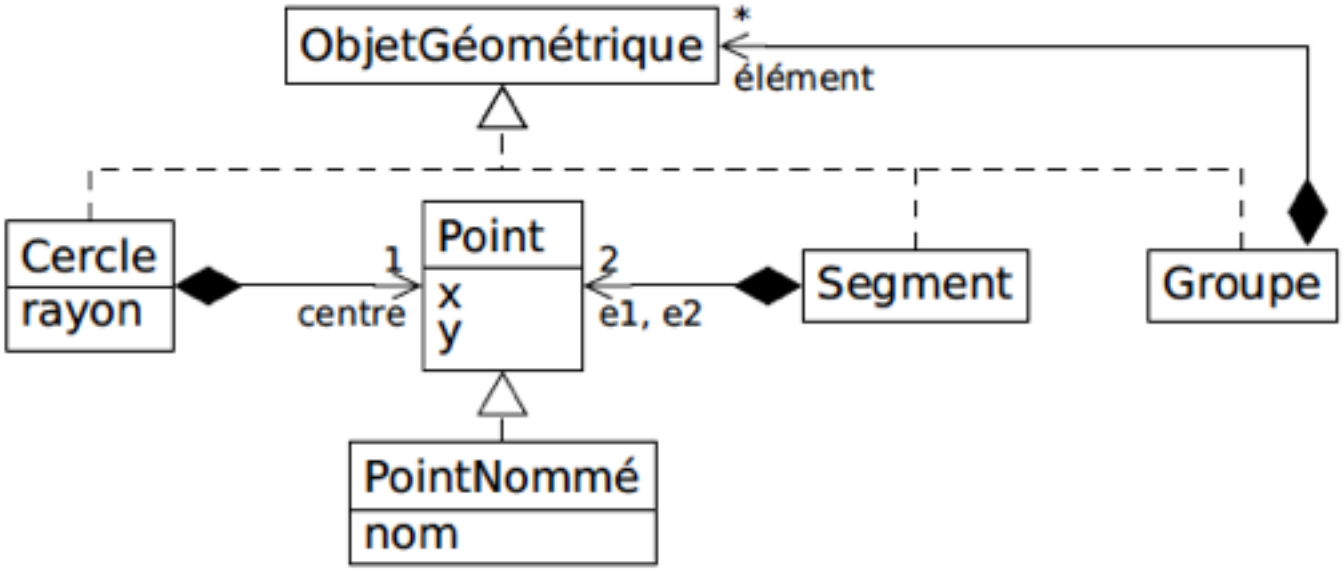
Pattern Iterator

- **Conséquences**
  - Possibilité de définir plusieurs parcours (infixe et préfixe par exemple)
  - Simplification de l'interface de l'agrégat
  - Parcours simultanés possibles sur un même agrégat (possible vu que chaque intérateur garde trace de l'état de son parcours)

PATRONS DE COMPORTEMENT

Visitor

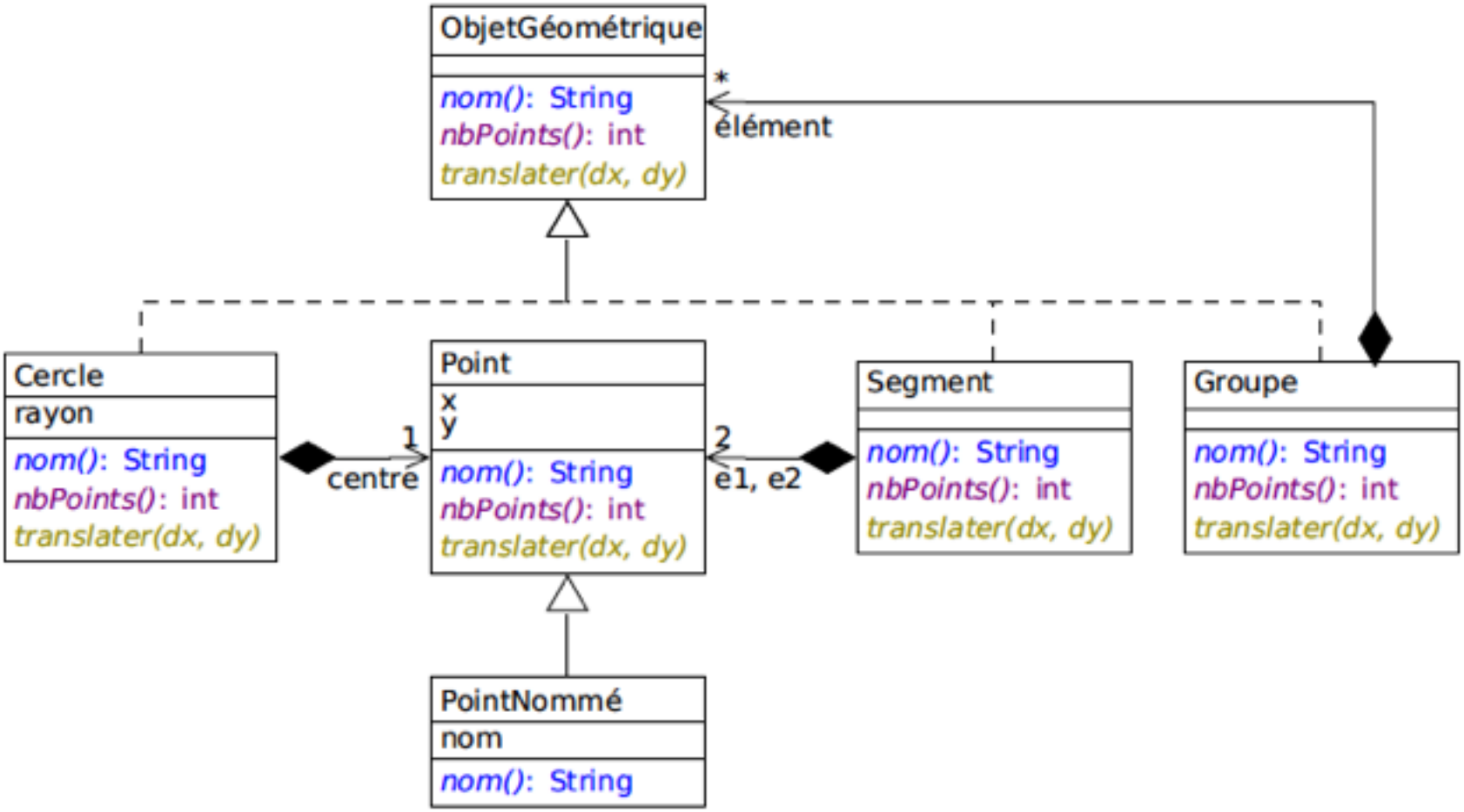
Points dans un objet géométrique



- Traitements à implanter :
  - obtenir le nom en français d'un élément : segment, point…
  - obtenir le nom en anglais
  - nombre de points utilisés pour caractériser un objet géométrique
  - afficher un objet géométrique
  - dessiner un objet géométrique
  - translater un objet géométrique

Solution Objet Classique

- Pour chaque opération, mettre une méthode générale les types généraux (objet géométrique et point) et la (re)définir dans la sous-classes



Problèmes de la solution

- Solution opérationnelle mais :
  - Dispersion du code: chaque traitement est éclaté sous l'ensemble de la structure de classes
    - Ça rend le code plus difficile à maintenir, étendre, documenter, etc.
  - L'ajout d'une nouvelle fonctionnalité est difficile
    - Il faut modifier toute la hiérarchie
    - Tous les clients doivent re-compiler même s'ils n'utilisent pas la nouvelle fonctionnalité
  - la structure est polluée par les différents traitements
  - Il faut pouvoir modifier les classes pour ajouter de nouveaux traitements

- Il faut:
  - Pouvoir grouper tout le code associé à une fonctionnalité particulière à un seul endroit,
  - Ne pas devoir modifier la hiérarchie de classes du système de fichiers chaque fois qu'une nouvelle fonctionnalité est ajoutée,
  - Pouvoir appliquer ces opérations de l'extérieur

Une classe par traitement

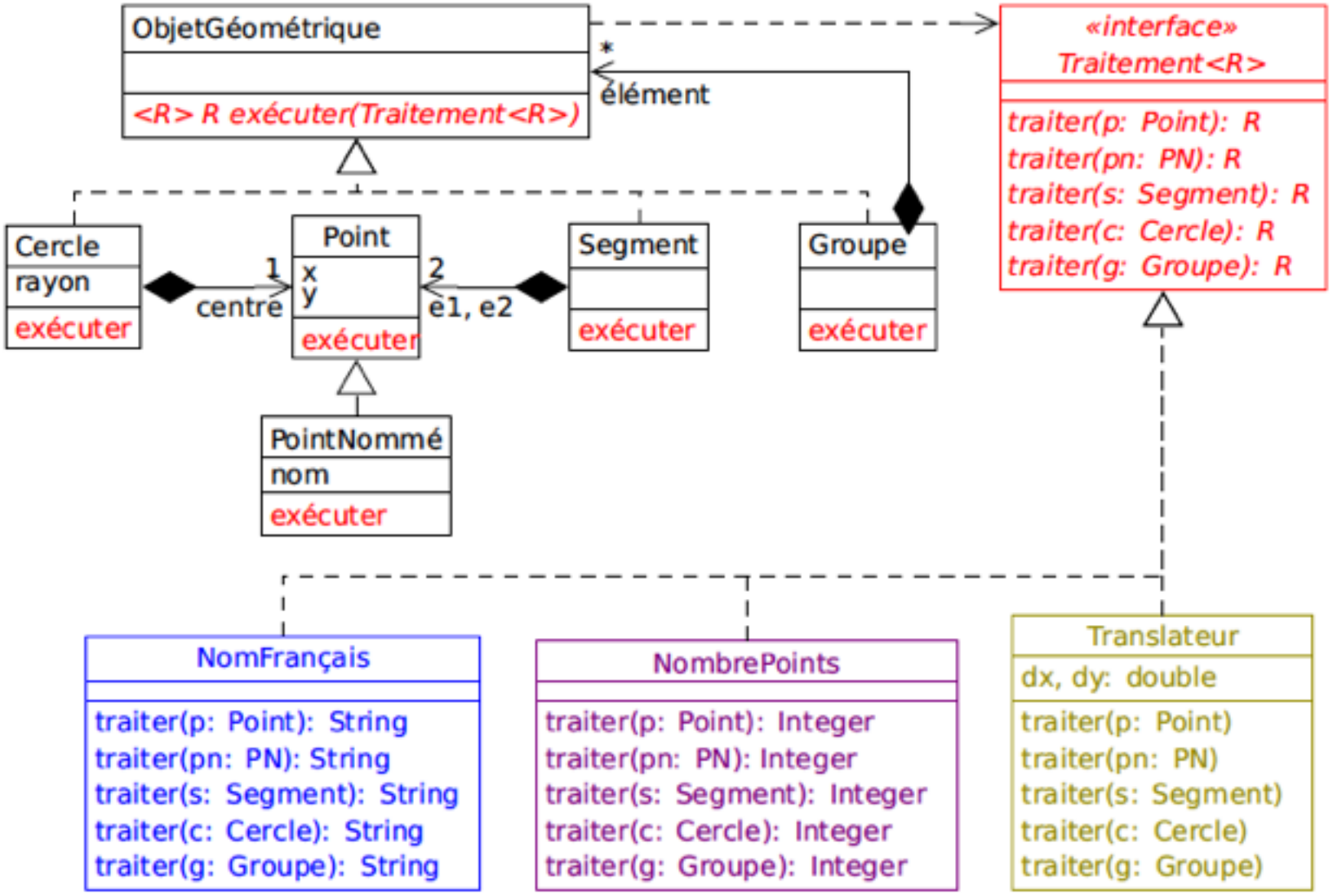

```
 public class NomFrancais {
 public String nom(Point p) { return "point"; }
 public String nom(PointNomme pn) { return
"point nommé"; }
 public String nom(Segment s) { return
"segment"; }
 public String nom(Cercle c) { return "cercle";
}
 public String nom(Groupe g) { return "groupe";
}
 }
 public class NomAnglais {
 public String nom(Point p) { return "point"; }
 public String nom(PointNomme pn) { return
"named point"; }
 public String nom(Segment s) { return
"segment"; }
 public String nom(Cercle c) { return "circle";
}
 public String nom(Groupe g) { return
"group"; }
}
```

```
 public class CompteurPoint {
  public int nbPoints(Point p) { return 1; }
  public int nbPoints(PointNomme pn){ return
1;}
  public int nbPoints(Segment s) { return 2; }
  public int nbPoints(Cercle c) { return 1; }

  public int nbPoints(Groupe g) {
    int somme = 0;
     for (ObjetGeometrique og : g.elements()) {
       if (og instanceof Segment) {
       somme += nbPoints((Segment) og);
       } else if (og instanceof Cercle) {
       somme += nbPoints((Cercle) og);
       } else if (og instanceof Groupe) {
       somme += this.nbPoints((Groupe) og);
       } else {
       throw new RuntimeException("Erreur dans
le traitement par cas !");
      }
  }
       return somme;
       }
  }
```

Critique et... une solution

- Ça marche mais :
  - Faire des instanceof est une mauvaise idée ! (voir l'exception levée…)
  - De nombreux traitements nécessitent ce traitement par cas : afficher, translater, dessiner, etc.
- Solution :
  - Faire la sélection de la bonne méthode sans instanceof
  - L'idée est de ne définir qu'une seule méthode, et non une par traitement
  - Il faut généraliser les différents traitements : c'est le Traitement/Visiteur !
  - Il faut unifier le nom des méthodes : traiter/visiter
  - Le type de retour change ? La généricité
  - Et la méthode sur OG ? exécuter(Traitement)/accepter(Visiteur)

L'architecture du Visiteur



Pattern Visitor

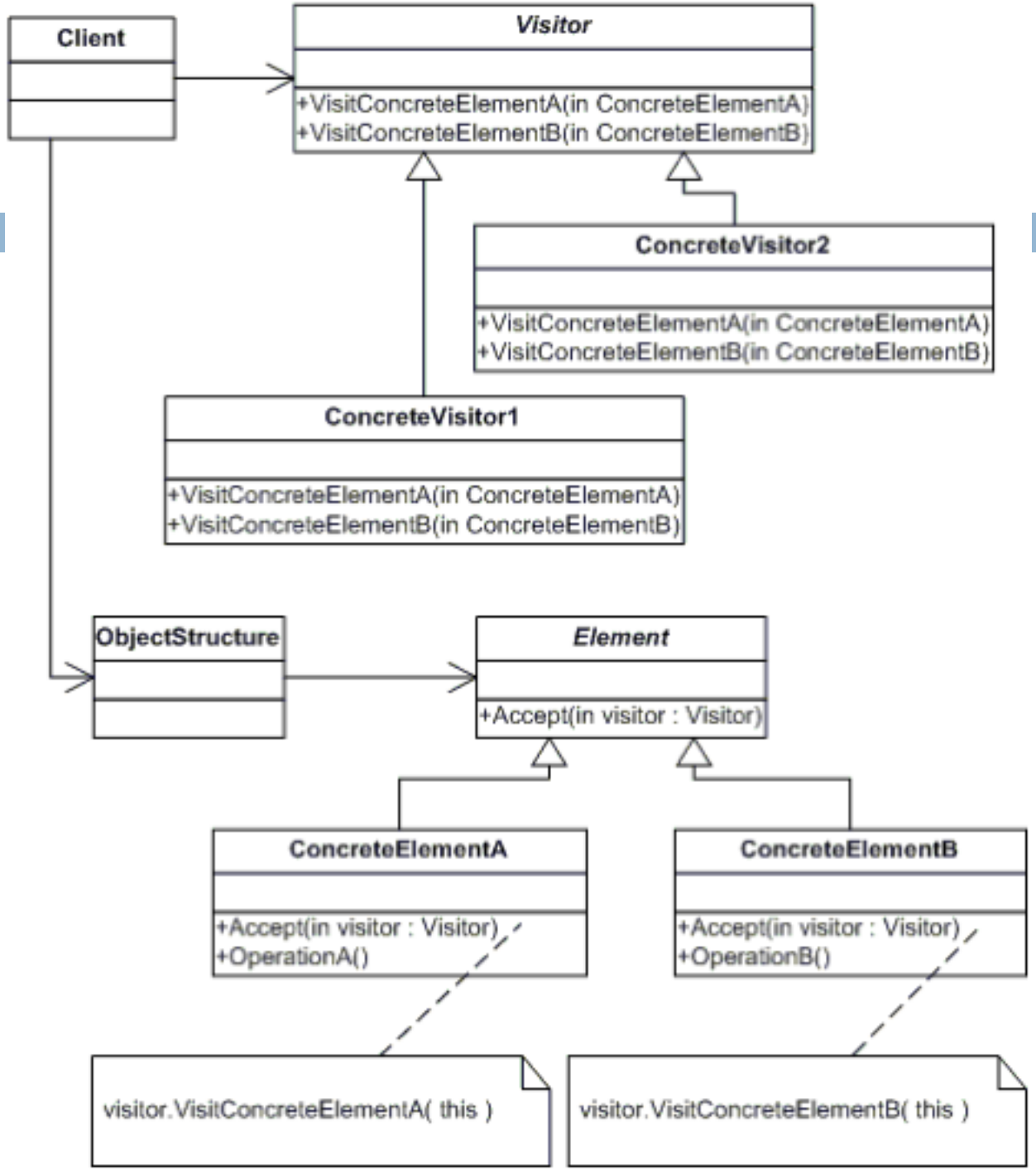
- Intention
  - Représenter une opération qui doit être appliquée sur les éléments d'une structure d'objets. Un Visitor permet de définir une nouvelle opération sans modification aux classes des objets sur lesquels l'opération va agir.
- Synonymes
  - Visiteur
- Utilisation connus
  - Différents traitement sur le même arbre, Compilateur, bibliothèques C++, Java…
- Patrons associés
  - Composite, Interpreter

Pattern Visitor

- **Problème (Quand l'utiliser)**
  - Lorsqu'une structure d'objets contient plusieurs classes avec des interfaces différentes.
  - Lorsque plusieurs opérations distinctes et sans liens entre elles doivent agir sur des objets conservés dans une structure
    - Pour éviter la pollution des classes de la structure
  - Lorsque les classes définissant la structure des objets changent rarement, mais que l'on veut fréquemment définir de nouvelles opérations sur cette structure.

Visitor

- Structure

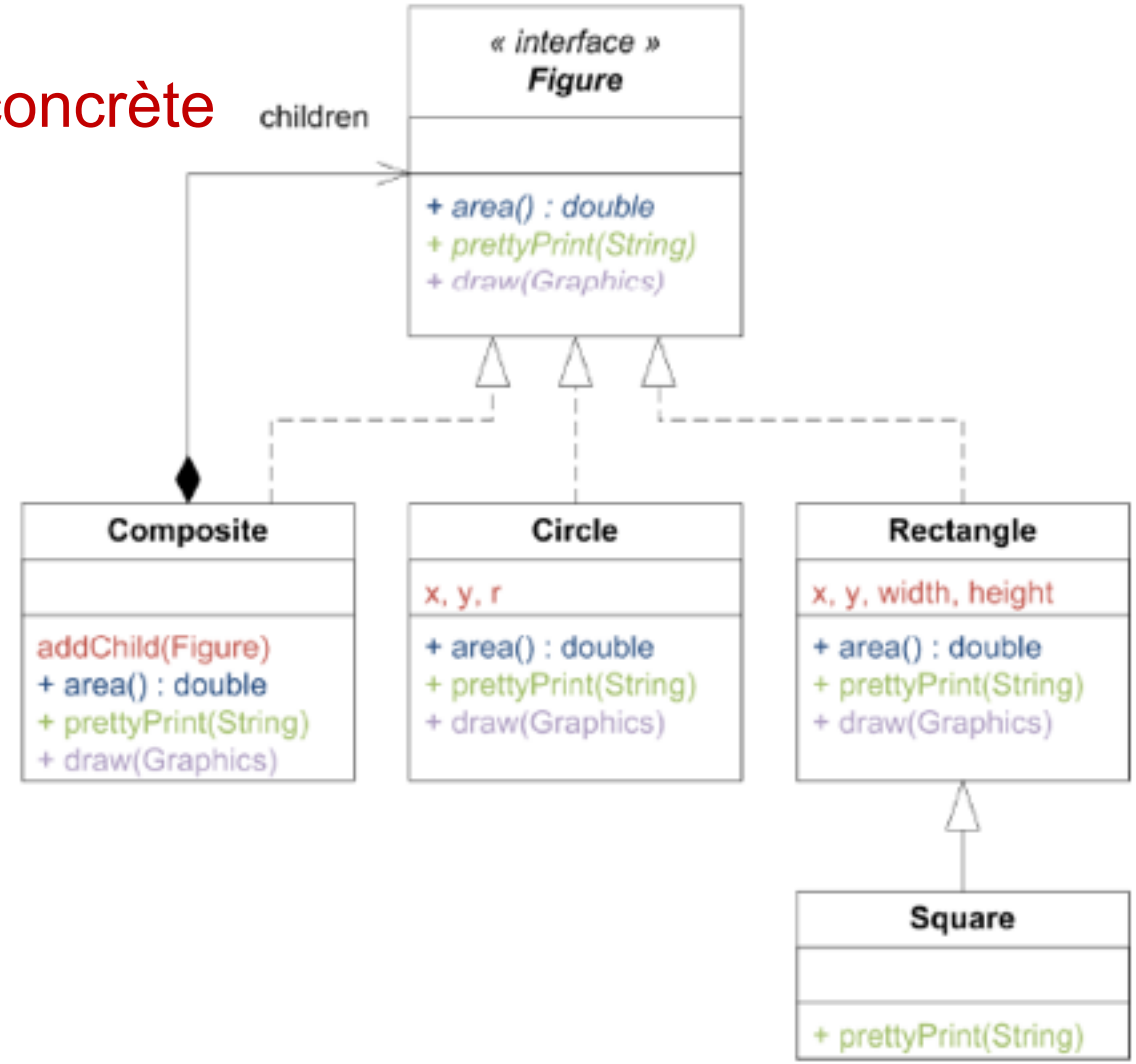


Pattern Visitor

- **Participants**
  - Visitor
    - Déclare l'opération de visite « visitX » pour chaque classe de ConcreteElement dans la structure
    - Le nom et la signature de l'opération identifie la classe qui envoie la requête de visite au visiteur. Le visiteur détermine alors la classe concrète et accède à l'élément directement
  - ConcreteVisitor
    - Implémente chaque méthode définit par l'interface Visitor
    - Chaque opération implémente un fragment de l'algorithme, et un état local peut être stocké pour accumuler les résultats de la traversée de la structure
  - Element
    - Définit une opération Accept qui prend un visitor en paramètre
  - ConcreteElement
    - Implémente l'opération Accept pour appeler Visitor.visitX(this)
  - ObjectStructure
    - Peut énumérer ses éléments et peut être un Composite, ou une collection.

Exemple: Figures sans visiteurs

- Métier Figure
- Spécifique classe concrète
- Service PrettyPrint
- Service draw



Exemple: Figures sans visiteurs


```
interface Figure {
void prettyPrint(String tab); // autant de méthode publiques que de services
void draw(Graphics g);
}
```

```
class Composite implements Figure {
Figure[] children;

public void prettyPrint(String tab) {
tab += "\t";
for(Figure child : children)
child.prettyPrint(tab);}

public void draw(Graphics g) {
for(Figure child : children)
child.draw(g);
}}
```

```
class Circle implements Figure {
double x, y, r;
public void prettyPrint(String tab) {
System.out.println(tab+"Circle:"+this);}

public void draw(Graphics g) {
g.drawEllipse(x, y, r*2, r*2);
}}
```

```
class Rectangle implements Figure {
double x, y, width, height;

public void prettyPrint(String tab) {
System.out.println(tab+ »Rectangle:"+this);}

public void draw(Graphics g) {
g.drawRect(x, y, width, height);
}}

class Square extends Rectangle {

public void prettyPrint(String tab) {
System.out.println(tab+"Square:"+this);
} // hérite de la méthode draw
}
```

Exemple: Figures avec visiteurs


```
interface FigureVisitor {
void visit(Composite c);
void visit(Circle c);
void visit(Rectangle r);
void visit(Square s);
}

class Composite implements Figure,
Iterable<Figure> {
Figure[] children;

public void accept(FigureVisitor visitor) {
visitor.visit(this);
}

}
class Circle implements Figure {
double x, y, r;

public void accept(FigureVisitor visitor) {
visitor.visit(this);
}
}
```

```
class Rectangle implements Figure {
double x, y, width, height;

public void accept(FigureVisitor visitor) {
visitor.visit(this);
}
}

class Square implements Figure {
Square(double x, double y, double length) {
super(x, y, length, length);
}
public void accept(FigureVisitor visitor) {
visitor.visit(this);
}
}
```

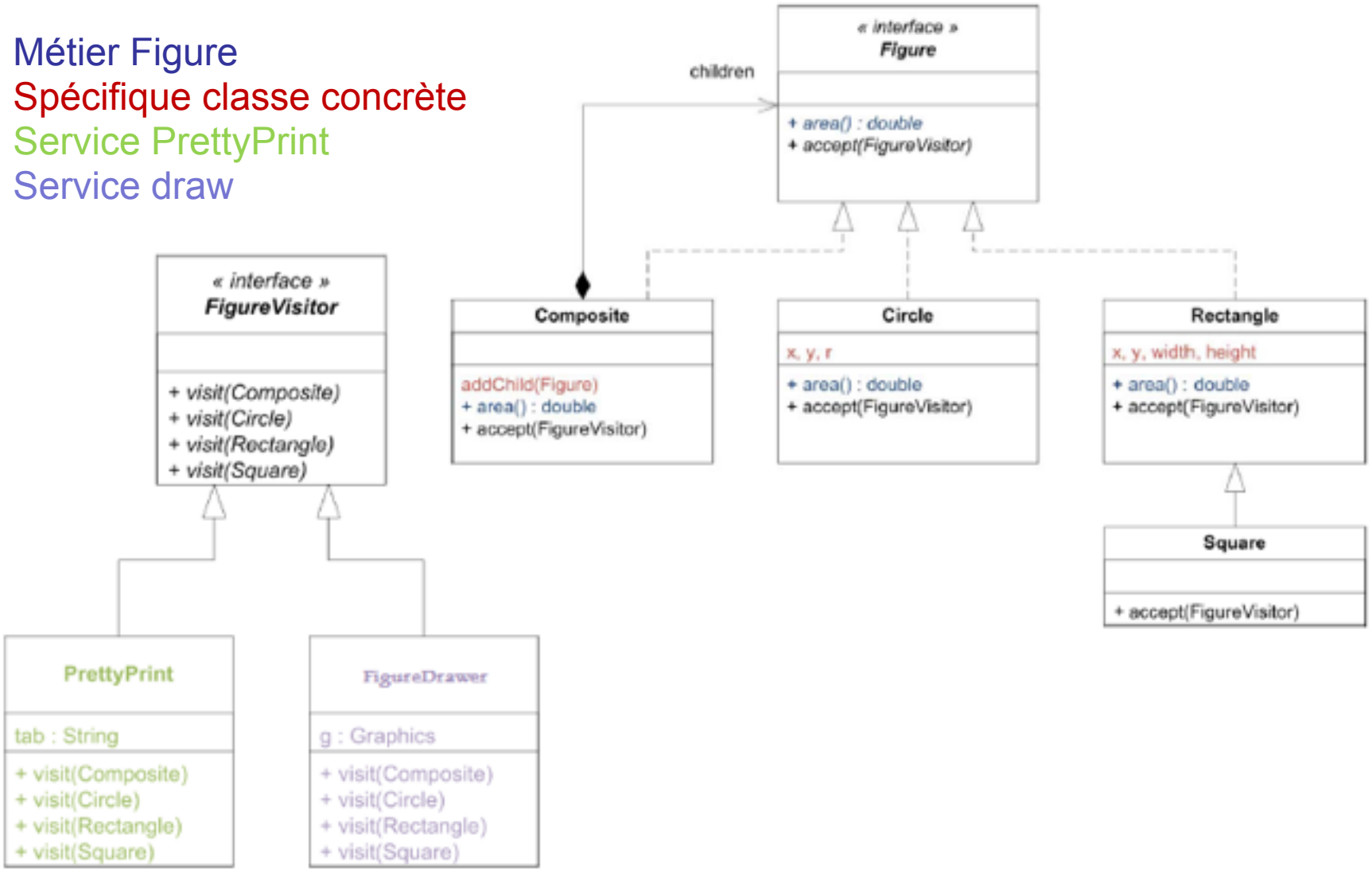
Exemple: Figures avec visiteurs


```
class PrettyPrint implements FigureVisitor {
String tab = "";

public void visit(Composite c) {
String oldTab = tab;
tab += "\t";

for(Figure child : c)
child.accept(this);
tab = oldTab;
}
public void visit(Circle c) {
System.out.println(tab + "Circle:" + c);
}
public void visit(Rectangle r) {
System.out.println(tab + "Rectangle:" + r);
}
public void visit(Square s) {
System.out.println(tab + "Square:" + r);
}
}
```

```
class FigureDrawer implements FigureVisitor {
Graphics g;
FigureDrawer(Graphics g) { this.g = g; }

public void visit(Composite c) {
for(Figure child : c)
child.accept(this);
}
public void visit(Circle c) {
g.drawEllipse(c.x, c.y, c.r*2, c.r*2);
}
public void visit(Rectangle r) {
g.drawRect(r.x, r.y, r.width, r.height);
}
public void visit(Square s) {
this.visit((Rectangle)s);
}
}
```

Exemple: Figures avec visiteurs

Métier Figure
Spécifique classe concrète
Service PrettyPrint
Service draw



Pattern Visitor

- **Conséquences**
  - Flexibilité: les Visitors et la structure d'objets sont indépendants
    - Ajout de nouveaux traitements très facile : Il suffit de créer un nouveau visiteur
  - Fonctionnalité localisée: tout le code associé à une fonctionnalité se retrouve à un seul endroit bien identifié.
    - Groupement/séparation des opérations communes
  - Visitor traverse des structures où les éléments sont de types complètement différents (différent de Iterator où les éléments doivent être de la même hiérarchie)
  - Accumulation d'état dans le visiteur plutôt que dans des arguments
  - Mais difficile d'ajouter un nouvel élément
    - Il faut ajouter une méthode dans chaque visiteur
  - Suppose que l'interface de ConcreteElement est assez riche pour que le visiteur fasse son travail cela force à montrer l'état interne et à casser l'encapsulation

Pattern Visitor

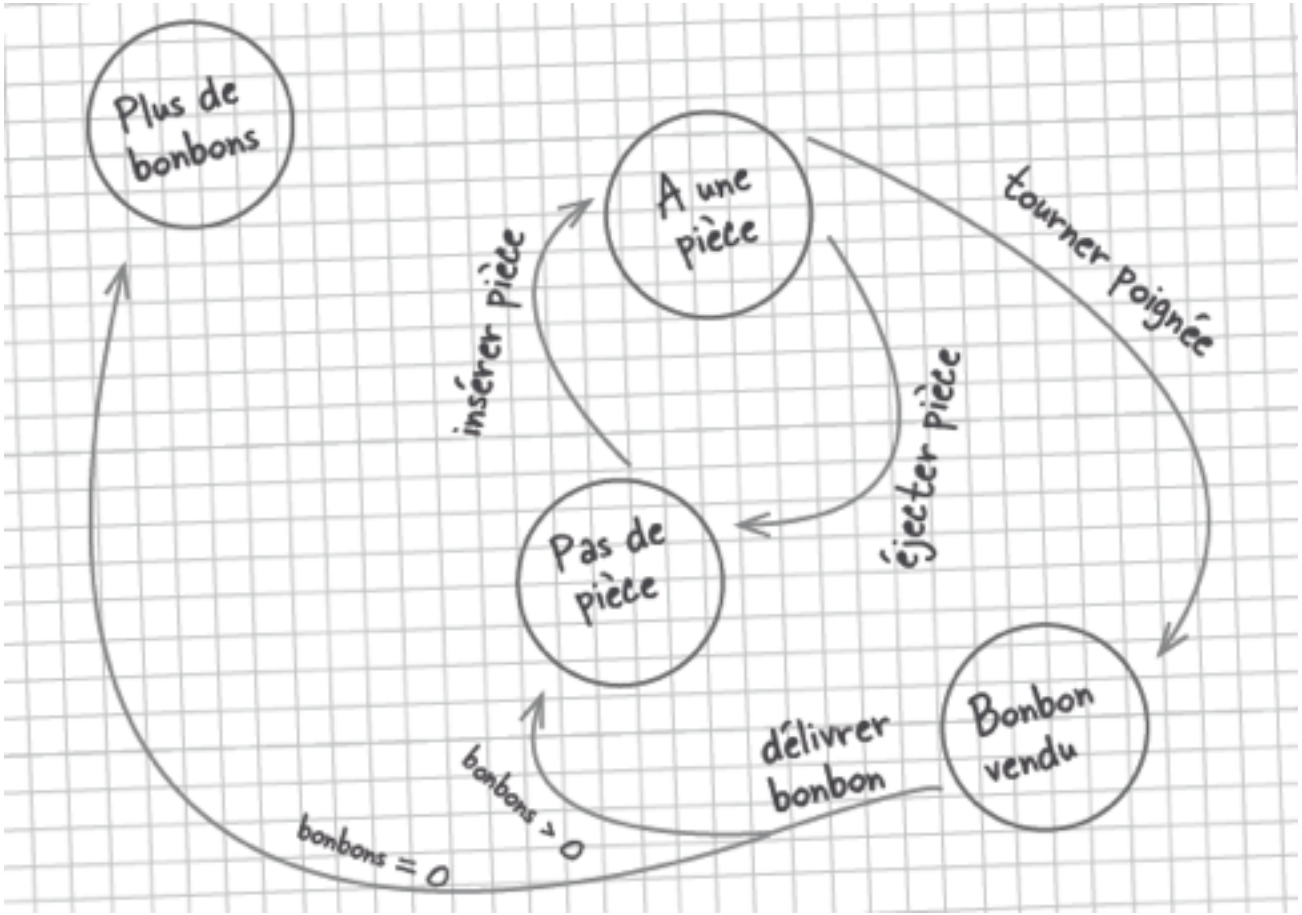
- **Implémentation**
  - Double invocation/aiguillage (double dispatch), : deux critères déterminent l'opération à effectuer, le Visiteur et l'Élément.
  - Qui organise le parcours de la structure d'objet :
    - Le visiteur
    - La structure d'objet (souvent choisi)
    - Un itérateur à part

PATRONS DE COMPORTEMENT

State

Motivation

- Distributeur de bonbons

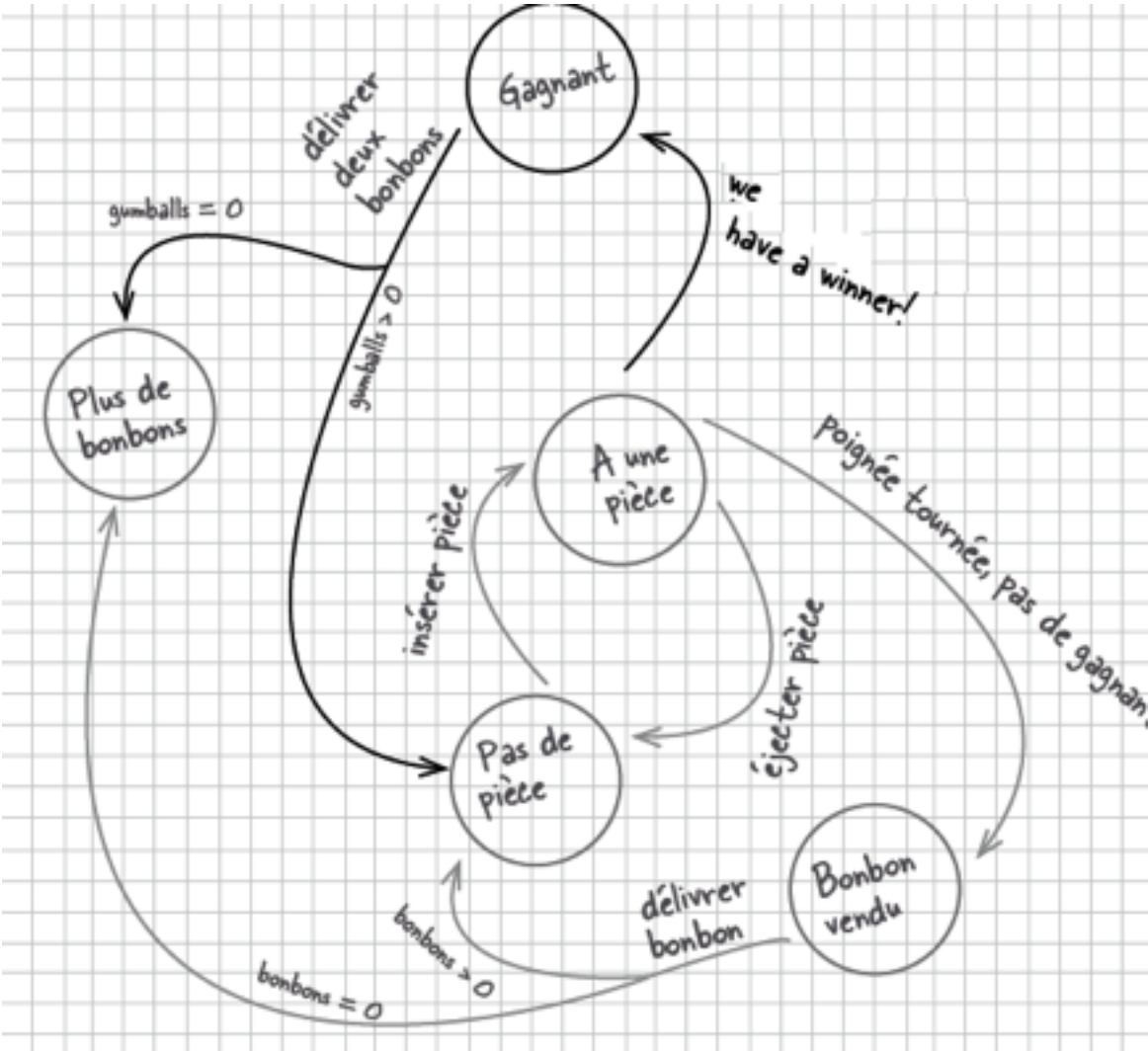


Code avec des fragments if/else


```
public class Distributeur {
final static int EPUISE = 0; // plus de bonbons
final static int SANS_PIECE = 1;
final static int A_PIECE = 2;
final static int VENDU = 3;

int etat = EPUISE;
int nombre = 0;

public Distributeur(int nombre) {
 this.nombre = nombre;
 if (nombre > 0) {
   etat = SANS_PIECE;
 }
}

public void insererPiece() {
 if (etat == A_PIECE) {
    System.out.println("Vous ne pouvez plus
insérer de pièces");
  } else if (etat == SANS_PIECE) {
    etat = A_PIECE;
    System.out.println("Vous avez inséré une
pièce");
  } else if (etat == EPUISE) {
    System.out.println("Vous ne pouvez pas
insérer de pièces, nous sommes en rupture
de stock");
}
else if (etat == VENDU) {
   System.out.println("Veuillez patienter, le
   bonbon va tomber");
   }
  }
}
public void ejecterPiece() {
if (etat == A_PIECE) {
System.out.println("Pièce retournée");
etat = SANS_PIECE;
} else if (etat == SANS_PIECE) {
System.out.println("Vous n'avez pas inséré de
pièce");
} else if (etat == VENDU) {
System.out.println("Vous avez déjà tourné la
poignée");
} else if (etat == EPUISE) {
System.out.println("Éjection impossible, vous
n'avez pas inséré de pièce");
}
}
public void tournerPoignee() {
..
}
public void delivrer() {
..
}
// autres méthodes comme toString() et remplir()
}
```

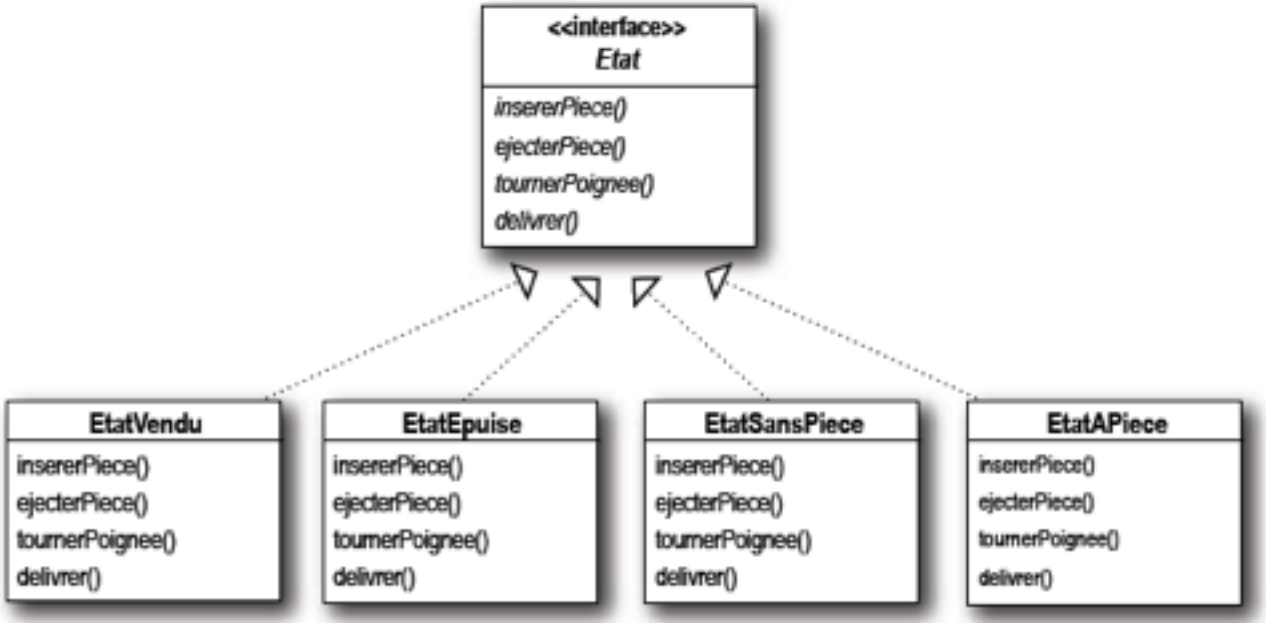
Changement des spécifications

- Transformer l'achat de bonbons en jeu
- Un bonbon gratuit sur 10: 10 % du temps, quand le client tourne la poignée, il reçoit deux bonbons au lieu d'un.
  - Un nouveau état: « Gagné »
  - il faudrait ajouter de nouvelles instructions conditionnelles dans chaque méthode pour gérer l'état GAGNANT.
  - tournerPoignee() doit tester si vous avez un GAGNANT, puis passer soit à l'état GAGNANT soit à l'état VENDU.



Nouvelle conception plus facile à maintenir

- Définir une interface Etat qui contiendra une méthode pour chaque action liée au Distributeur.
- 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.
- Enlever toutes les instructions conditionnelles et les remplacer par une délégation à la classe état



Nouveau Code


```
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...
}

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");
}
}
```

Pattern State

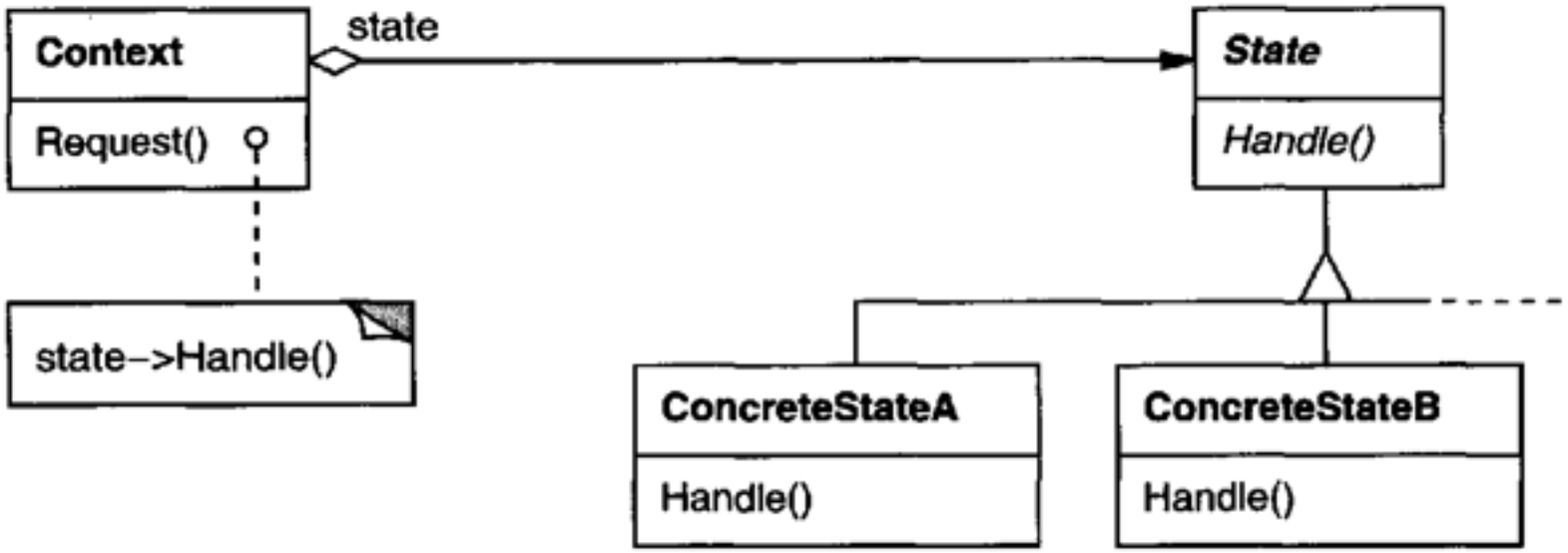
- Intention
  - Permet à un objet de modifier son comportement quand son état interne change. Donne l'impression que l'objet change de classe.
- Synonymes
  - Etat, Objects for States
- Utilisation connus
  - Protocole de connection TCP, processeurs de commandes et serveurs (FTP, mail,), implémentation des machines à états (Diagrammes états transitions d'UML), IHM contextuels (menus contextuels, actions dépendantes de l'état).
- Patrons associés
  - Flyweight: détermine quand et comment partager les états
  - Singleton: les objets « état » sont souvent des singleton

Pattern State

- **Problème (Quand l'utiliser)**
  - Quand le comportement d'un objet dépend de son état et ce changement est dynamique, au moment de l'exécution (polymorphisme d'état).
  - Quand l'algorithme d'une méthode est trop complexe et comporte des instructions conditionnelles qui dépendent de l'état de l'objet, souvent représenté sous forme d'une énumération de constantes.
    - Souvent plusieurs opérations contiennent ces parties conditionnelles invariante d'un algorithme

Pattern State

- Structure

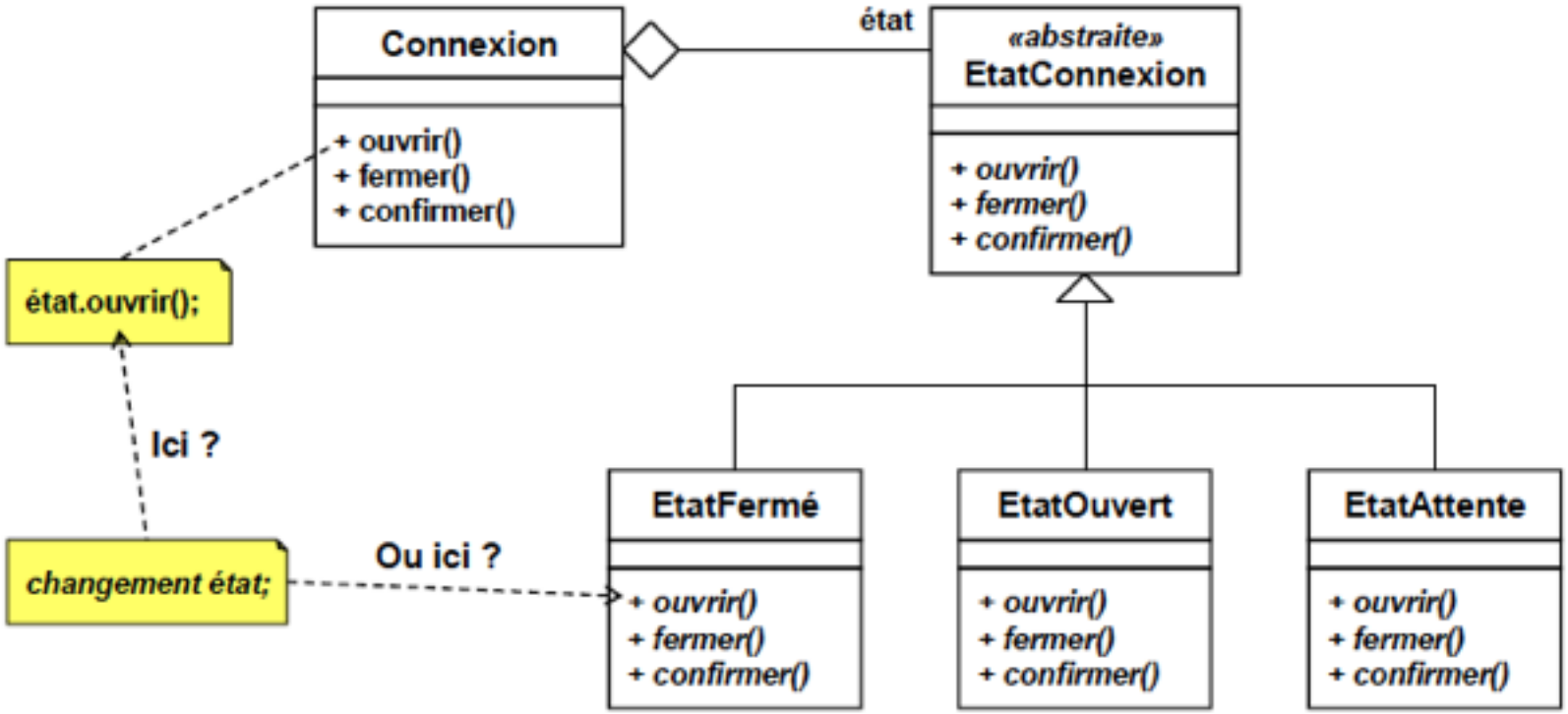


Pattern State

- **Participants**
  - **Context:** est une classe qui permet d'utiliser un objet à états et qui gère une instance d'un objet ConcreteState
  - **State :** définit une interface qui encapsule le comportement associé avec un état particulier de Context
  - **ConcretState:** implémente un comportement associé avec l'état de Context
- **Collaborations**
  - Il revient soit à Context, soit aux ConcreteState de décider de l'état qui succède à un autre état

Exemple

- **Réponses différentes suivant l'état d'une connexion réseau**
  - Etats: établie, en attente, fermée



Pattern State

- **Conséquences**
  - Séparation des comportements relatifs à chaque état
  - Transitions plus explicites
  - Les objets état peuvent être partagés s'ils n'ont pas de variable d'instance et définissent uniquement des comportement (indépendant du contexte)
  - Avantages
    - modularise le comportement de l'objet, explicite sa structuration en états isolés
    - facilite l'ajout de nouvelles modalités par l'introduction de nouvelles sous-classes d'Etat.
  - Inconvénients
    - multiplie le nombre de classes et d'objets mais cette complexité est à comparer avec la complexité du code des méthodes (switch).
    - cet inconvénient peut être limité si l'on dispose de modules ou de classes internes (« inner classes » en Java)

Pattern State

- **Implémentation**
  - Qui définit les transitions d'état ? Le contexte ou les objets état ?
    - Si les critères de transition sont fixes, la classe contexte peut s'occuper des transitions.
    - Sinon, c'est aux sous-classes représentant les états de définir les transitions. Il faut donc que la classe contexte ajoute à son interface une opération permettant aux sous-classes d'état de changer l'état. Cela sous-entend également que la classe de base Etat doit maintenir une référence à la classe Contexte. L'inconvénient de la décentralisation est qu'un état doit au moins connaitre un autre état vers qui il transite.
  - Alternative de table d'état: se focalise sur la determination des transition d'états tandis que le pattern State s'intéresse au comportement

Pattern State

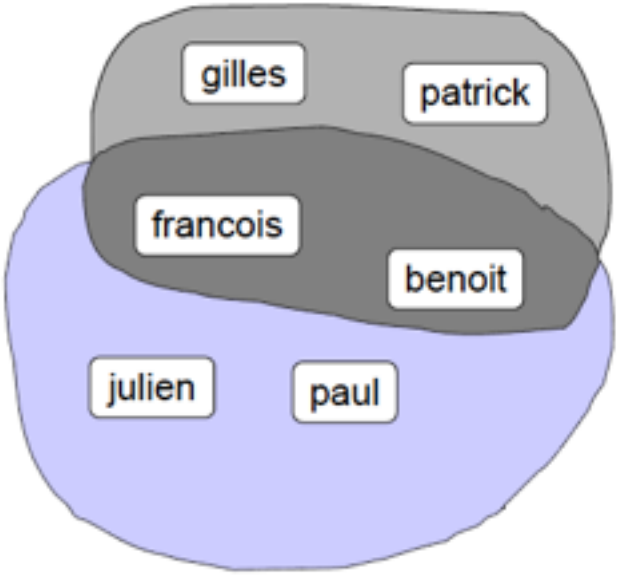
- **Implémentation**
  - Création et destruction des objets états, deux possibilités
    - Tous les états sont construits une fois pour toutes et jamais détruits : Quand les changements d'état sont fréquents et apparaissent rapidement
    - Les états sont créés uniquement au moment où ils sont nécessaires : Quand les changements d'état ne sont pas fréquents, et connus au moment de l'exécution.
  - Héritage dynamique:
    - Changer le type d'un objet à l'exécution…
    - N'est pas supporté par la plupart des langages OO..

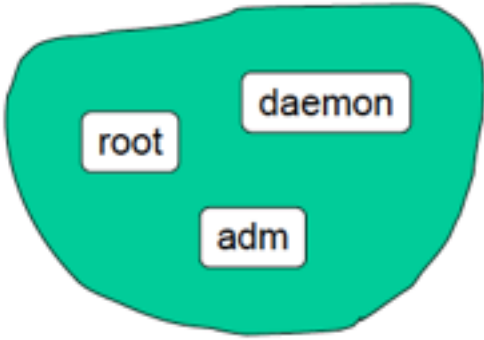
PATRONS DE COMPORTEMENT

Mediator

Protection multi usagers

- Exemples de groupes

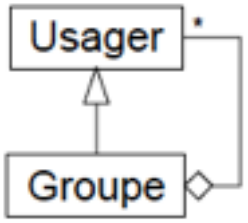




- Des groupes comme des objets
  - Sous-classés d'une hiérarchie existante, ou une nouvelle classe (hiérarchie) ?
  - Supposons qu'il s'agisse d'une sous-classe d'Usager

Protection multi usagers

- Un composite d'Usager ?



- Pourquoi composite n'est pas applicable:
  - La relation Usager – Groupe n'est pas récursive, du moins pas sous UNIX,
  - Il ne s'agit pas d'une relation strictement hiérarchique
    - Un usager peut appartenir à plusieurs groupes
  - Traiter les usagers et les groupes de façon uniforme est très discutable:
    - Se connecter comme un groupe ?
    - Utiliser un groupe comme identification ?

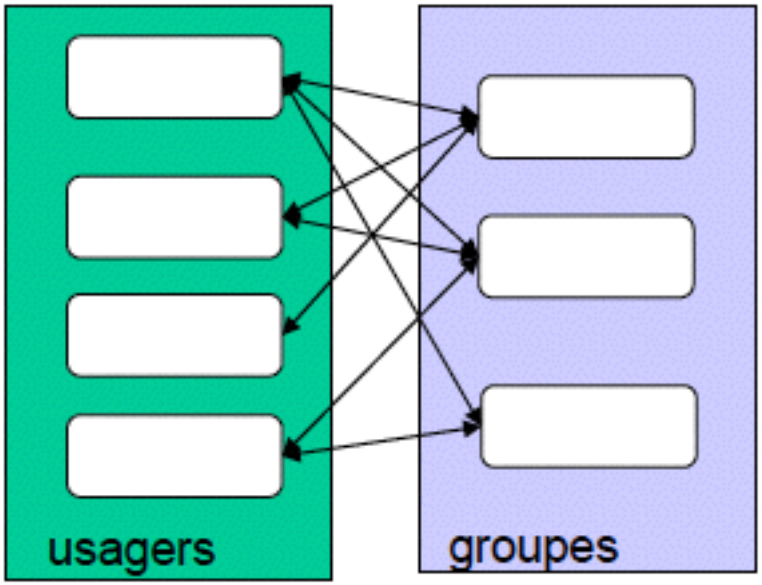
Protection multi usagers

- On a quand même besoin d'associer des usagers et des groupes.
- Il faut une association bidirectionnelle pour des raisons d'efficacité:
  - Le nombre d'usagers est beaucoup plus grand que le nombre de groupes
  - On veut trouver tous les membres d'un groupe sans parcourir tous les usagers du système.
- Vérifier si un usager est membre d'un groupe devrait être rapide aussi

Protection Multi-Usagers

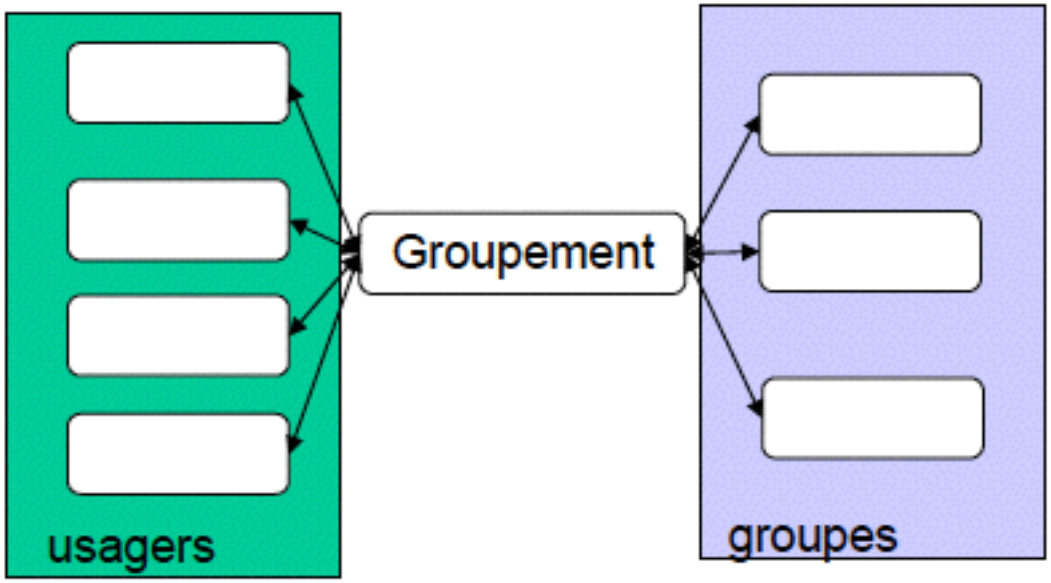
- Une solution:

- Désavantages:
  - Les références sont difficiles à changer de façon non invasive,
  - Tous les objets doivent payer le prix de références à un ensemble d'objets.
  - Enchevêtrement de références entre les objets Usagers et les objets Groupes.
    - Diagramme « spaghetti » !



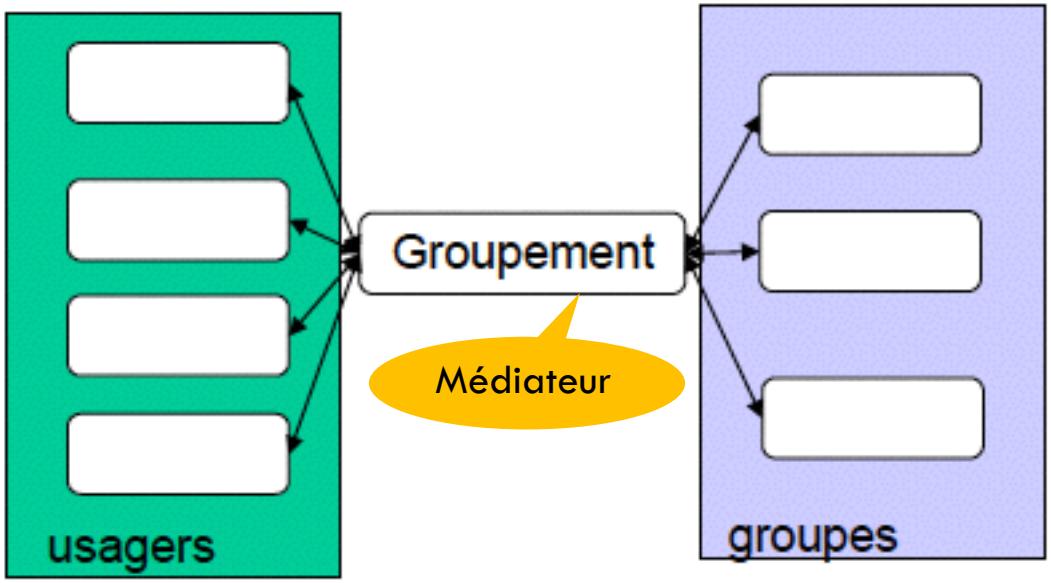
Patron Mediator

- Solution
  - Ajouter une classe pour gérer l'interconnexion entre les objet



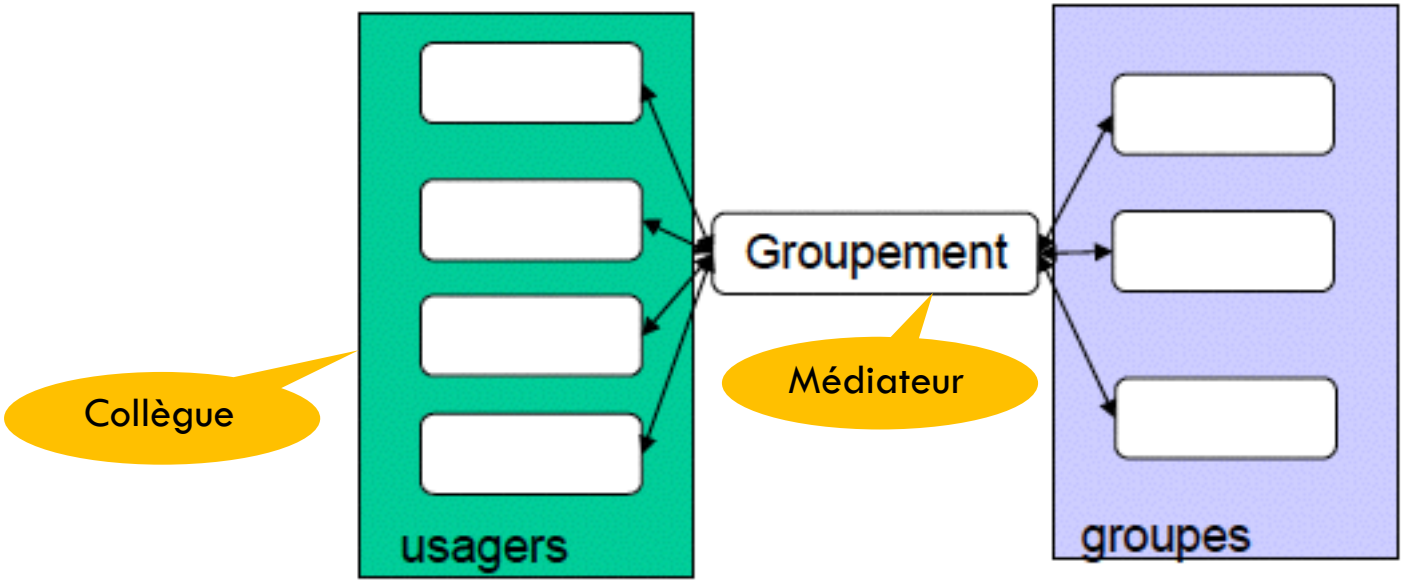
Patron Mediator

- Solution
  - Ajouter une classe pour gérer l'interconnexion entre les objet



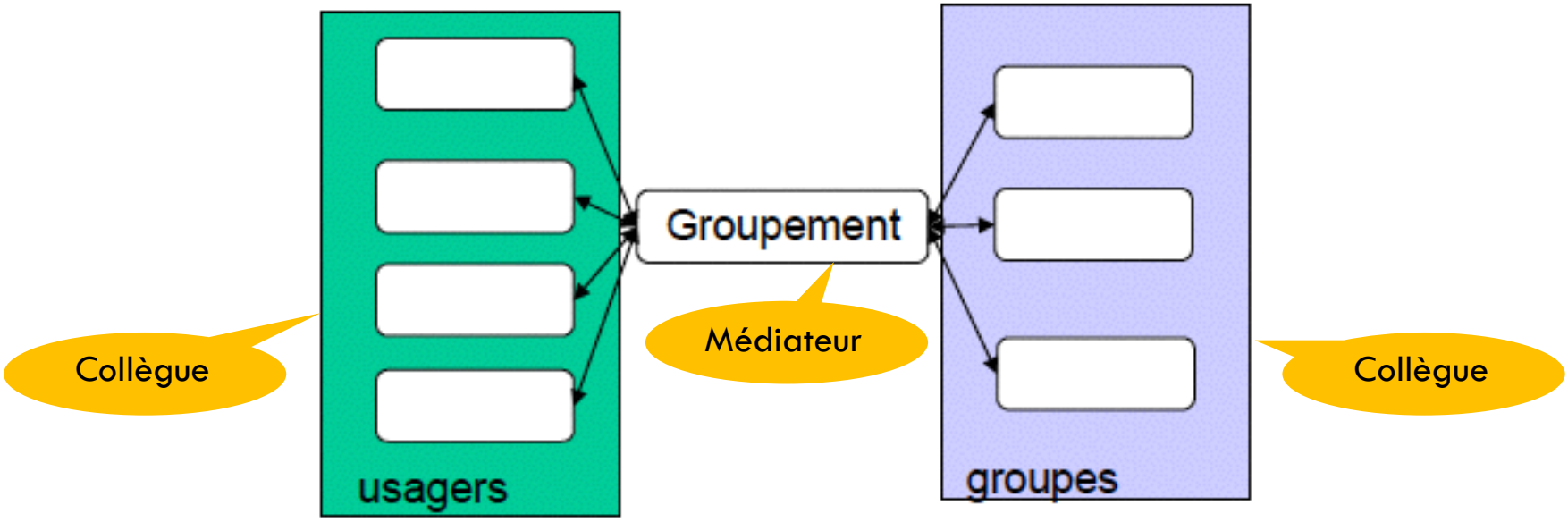
Patron Mediator

- Solution
  - Ajouter une classe pour gérer l'interconnexion entre les objet



Patron Mediator

- Solution
  - Ajouter une classe pour gérer l'interconnexion entre les objet



Pattern Merdiator

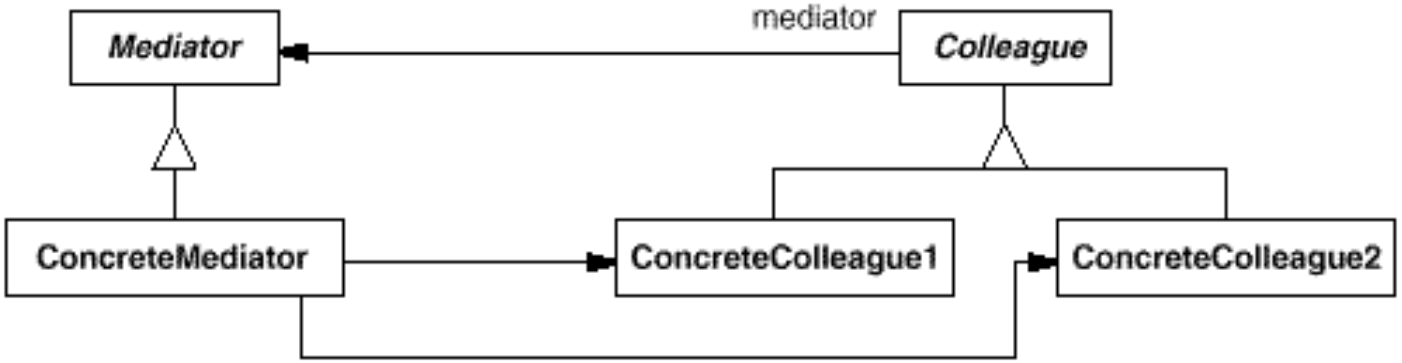
- Intention
  - Définir un objet qui encapsule comment un ensemble d'objets interagissent afin de promouvoir un couplage faible et de laisser varier l'interaction entre les objets de façon indépendante.
- Synonymes
  - Médiateur
- Utilisation connus
  - ViewManager dans les interfaces graphique
- Patrons associés
  - Observer : Moyen de communication avec le médiateur
  - Façade est unidirectionnel, Mediator est multidirectionnel et permet un comportement coopératif

Pattern Mediator

- **Problème (Quand l'utiliser)**
  - Un ensemble d'objets communiquent entre eux de façon bien définie mais complexe : inter-dépendances non structurées et difficiles à appréhender
  - La réutilisation d'un objet est difficile car il fait référence à beaucoup d'objets
  - Un comportement distribué entre plusieurs classes doit pouvoir être spécialisé sans multiplier les sous-classes

Pattern Mediator

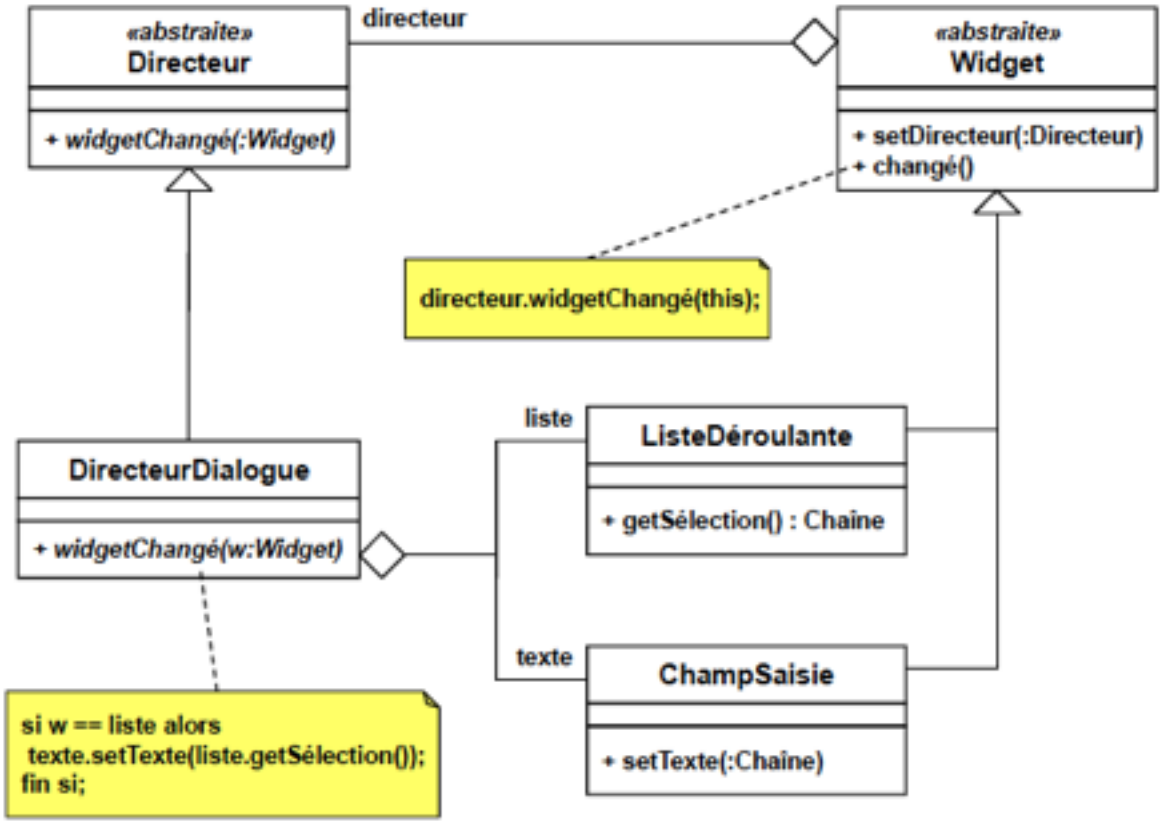
- Structure



- **Participants**
  - Mediator: définit une interface pour communiquer avec des objets Colleague
  - ConcreteMediator
    - Implémente un comportement coopératif en cordonnant des objets Colleague.
    - Connaît et maintient ses Colleagues.
  - Colleague
    - Chaque classe Colleague connaît son objet Mediator.
    - Chaque Colleague communique avec son Mediator à chaque fois qu'il veut communiquer avec un autre objet Colleague

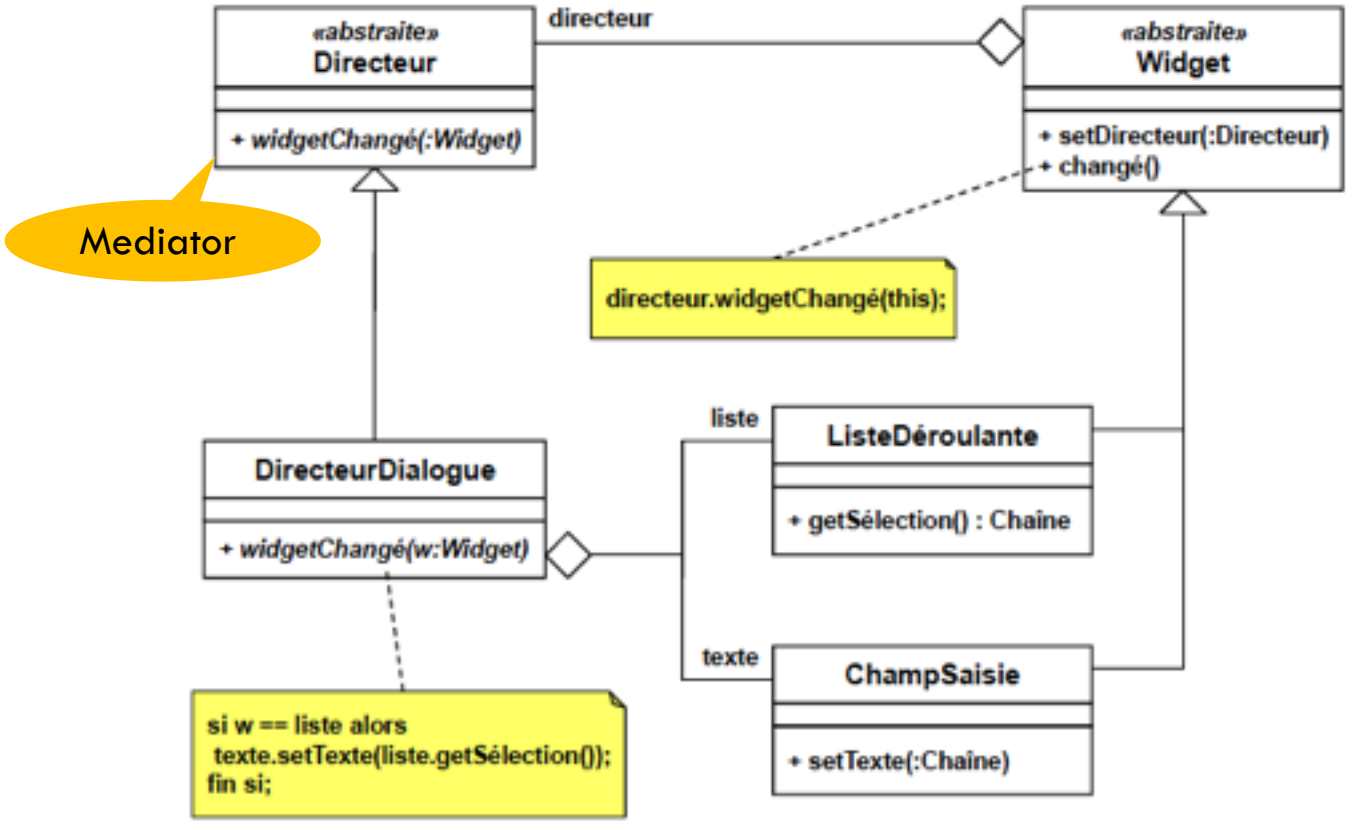
Exemple

- Médiateur entre les éléments graphiques d'une boîte de dialogue



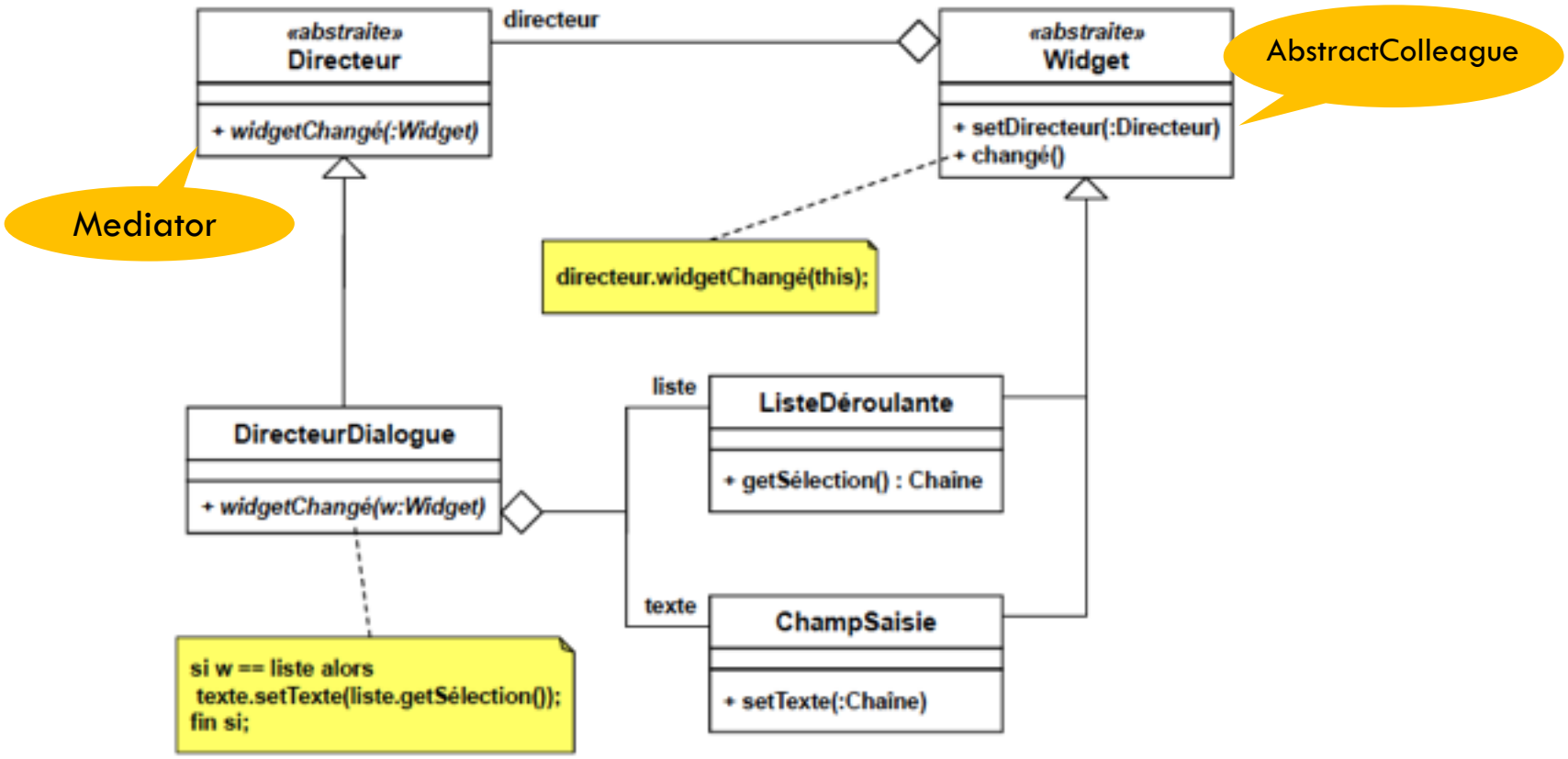
Exemple

- Médiateur entre les éléments graphiques d'une boîte de dialogue



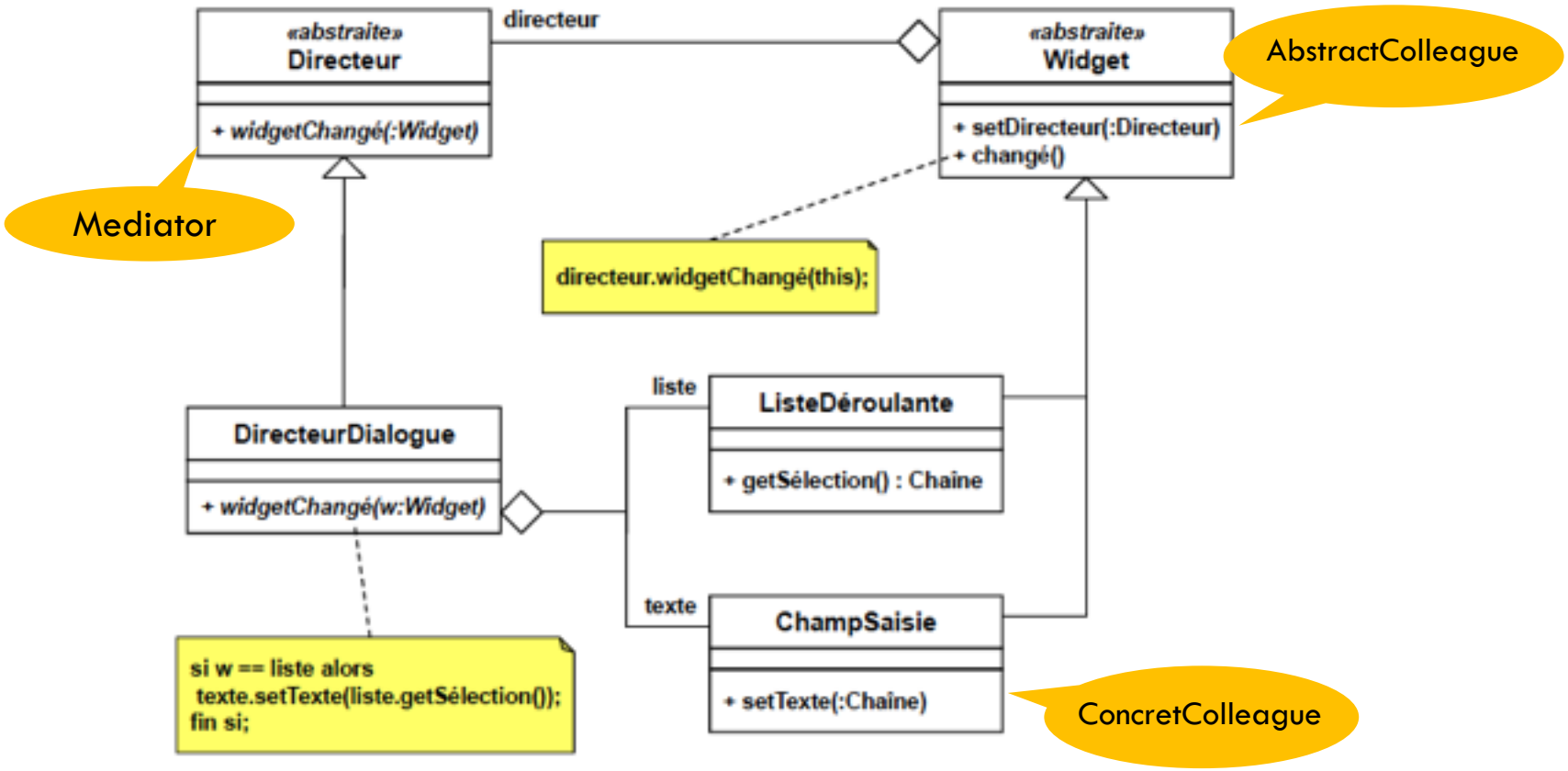
Exemple

- Médiateur entre les éléments graphiques d'une boîte de dialogue



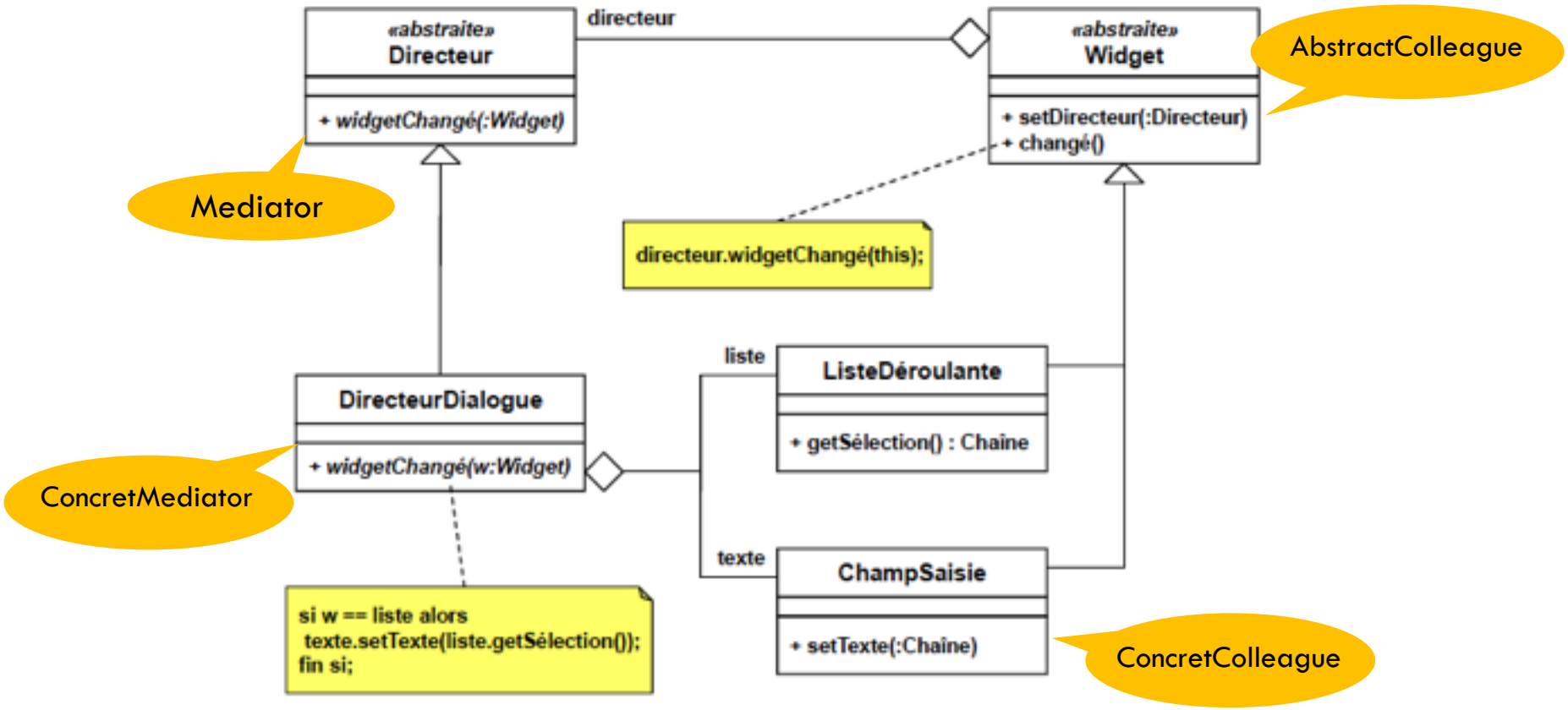
Exemple

- Médiateur entre les éléments graphiques d'une boîte de dialogue



Exemple

- Médiateur entre les éléments graphiques d'une boîte de dialogue



Pattern Mediator

- **Implémentation**
  - Omission de la classe abstraite Mediator quand les classes travaille avec un seul médiateur
  - Communication Colleague-Mediator souvent implémenté avec Observer
- **Conséquences**
  - limite la création de sous-classes. La modification du comportement nécessite de sous-classer uniquement le médiateur, pas les collègues (qui peuvent rester inchangés)
  - réduit le couplage entre collègues
  - simplifie les protocoles objets. le médiateur remplace une interaction plusieurs à plusieurs par une interaction 1-à-plusieurs qui est plus facile à comprendre, maintenir, et étendre.
  - formalise la coopération des objets. Encapsuler la médiation dans un objet permet de se concentrer sur l'interaction des objets dans le système en dehors de leurs comportement individuel.
  - centralise le contrôle. Le médiateur encapsule la complexité de l'intéraction, ce qui risque de le rendre difficile à maintenir.

PATRONS DE COMPORTEMENT

Strategy

Motivation

- Dans un jeu vidéo, des personnages combattent des monstres... ;
  - méthode combat(Monstre m) de la classe Perso ...et le code de combat peut être différent d'un personnage à l'autre
    - Sol. 1 : combat contient un cas pour chaque type de combat
    - Sol. 2 : La classe Perso est spécialisée en sous-classes qui redéfinissent combat
    - Sol. 3 : La classe Perso délègue le combat à des classes encapsulant des codes de combat et réalisant toutes une même interface
- Quelle solution permet facilement de :
  - Ajouter un nouveau type de combat ?
  - Changer le type de combat d'un personnage ?

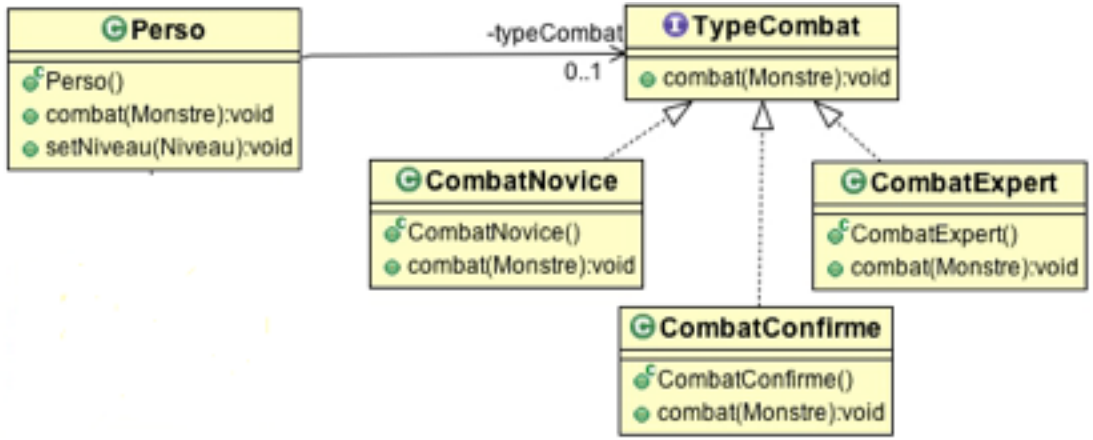
Motivation

- Dans un jeu vidéo, des personnages combattent des monstres... ;
  - méthode combat(Monstre m) de la classe Perso ...et le code de combat peut être différent d'un personnage à l'autre
    - Sol. 1 : combat contient un cas pour chaque type de combat
    - Sol. 2 : La classe Perso est spécialisée en sous-classes qui redéfinissent combat
    - Sol. 3 : La classe Perso délègue le combat à des classes encapsulant des codes de combat et réalisant toutes une même interface
- Quelle solution permet facilement de :
  - Ajouter un nouveau type de combat ?
  - Changer le type de combat d'un personnage ?

Réponse : La solution 3

Motivation

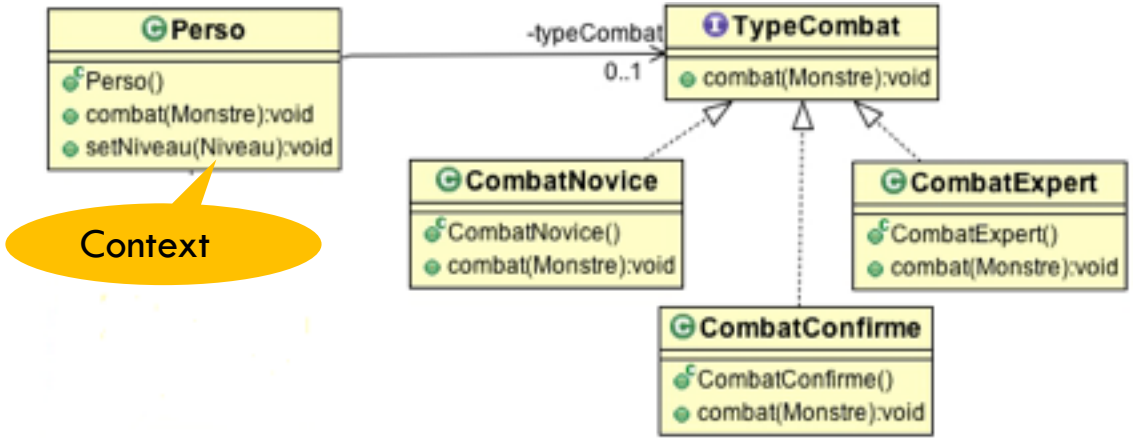
- Diagramme de classes de la solution 3 :




```
public class Perso {

      private TypeCombat typeCombat;

      public Perso(TypeCombat t){
            this.typeCombat=t;
      }

      public void combat(Monstre m){
            typeCombat.combat(m);
      }
   ...
}
```

Motivation

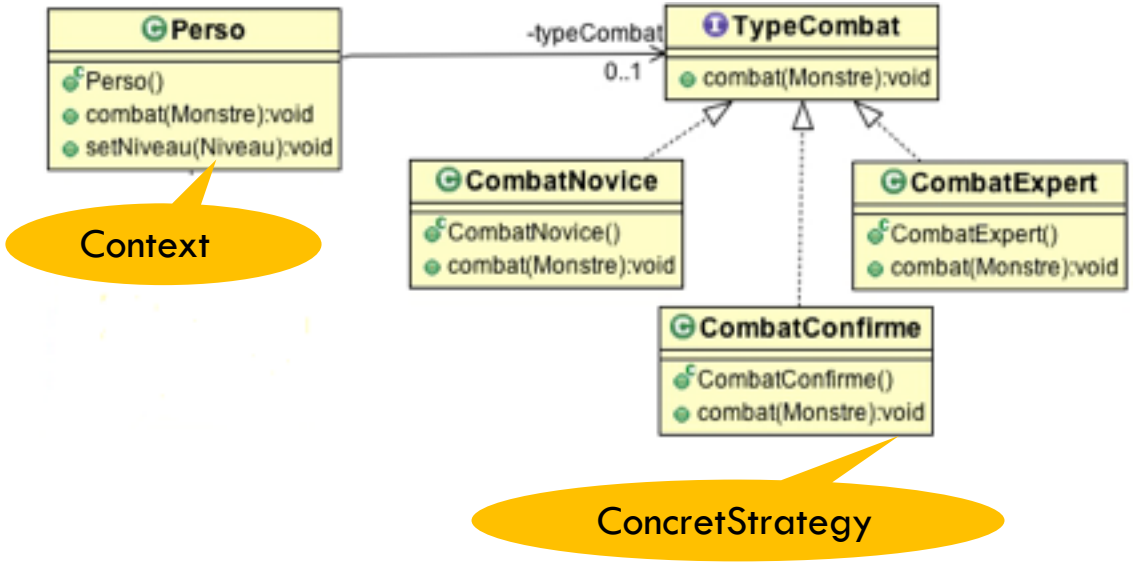
- Diagramme de classes de la solution 3 :




```
public class Perso {

      private TypeCombat typeCombat;

      public Perso(TypeCombat t){
            this.typeCombat=t;
      }

      public void combat(Monstre m){
            typeCombat.combat(m);
      }
   ...
}
```

Motivation

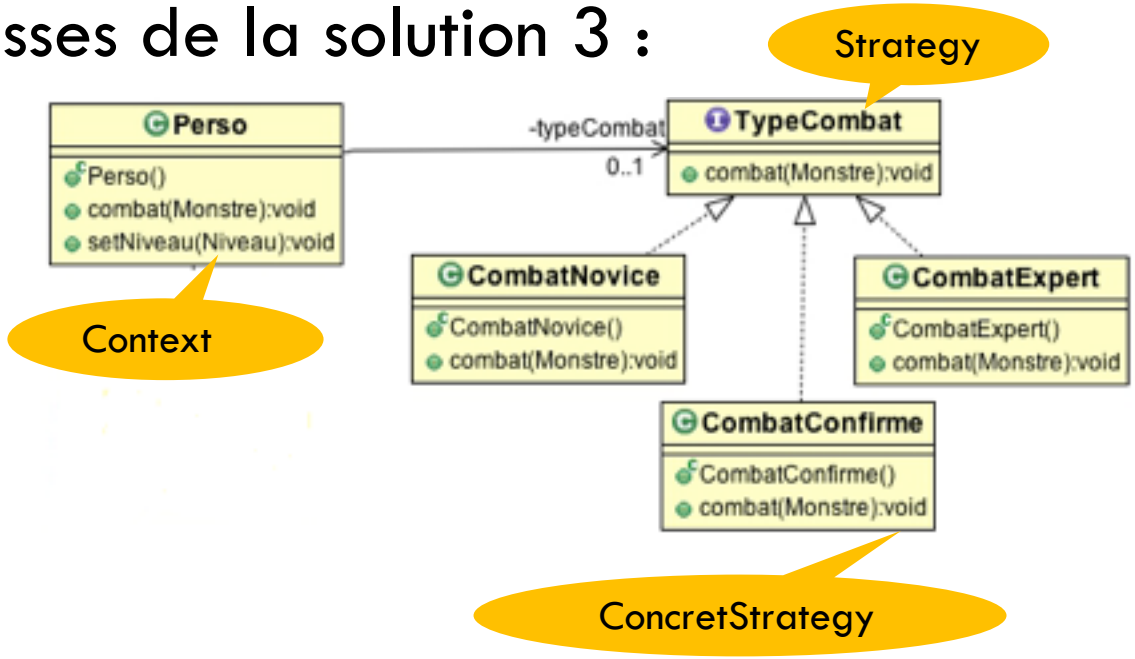
- Diagramme de classes de la solution 3 :




```
public class Perso {

      private TypeCombat typeCombat;

      public Perso(TypeCombat t){
            this.typeCombat=t;
      }

      public void combat(Monstre m){
            typeCombat.combat(m);
      }
   ...
}
```

Motivation

- Diagramme de classes de la solution 3 :




```
public class Perso {

      private TypeCombat typeCombat;

      public Perso(TypeCombat t){
            this.typeCombat=t;
      }

      public void combat(Monstre m){
            typeCombat.combat(m);
      }
   ...
}
```

Pattern Strategy

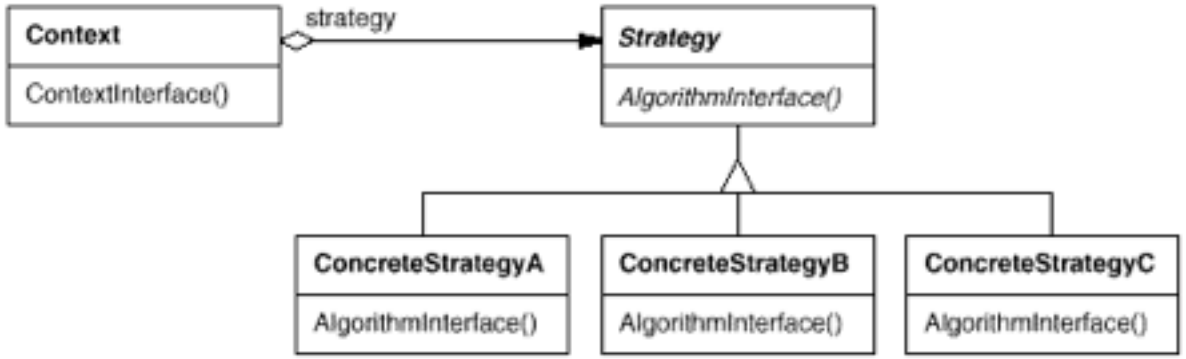
- Intention
  - Définir une hiérarchie de classes pour une famille d'algorithmes, encapsuler chacun d'eux, et les rendre interchangeables.
  - Les algorithmes varient indépendamment des clients qui les utilisent
- Synonymes
  - Stratégie, Policy (Politique)
- Utilisation connus
  - Validation des données en GUI, AWT et Swing en Java
- Patrons associés
  - Flyweight

Pattern Strategy

- Problème (Quand l'utiliser)
  - Lorsque de nombreuses classes associées ne diffèrent que par leur comportement. Stategy permet de configurer une classe avec un comportement parmi plusieurs.
  - Lorsqu'on a besoin de plusieurs variantes d'un algorithme
  - Lorsqu'un algorithme utilise des données que les clients ne doivent pas connaître (masquer les structures complexes)
  - Lorsqu'une classe définit plusieurs comportements, figurant dans des conditionnelles multiples : faire autant de classes Stratégie

Pattern Strategy

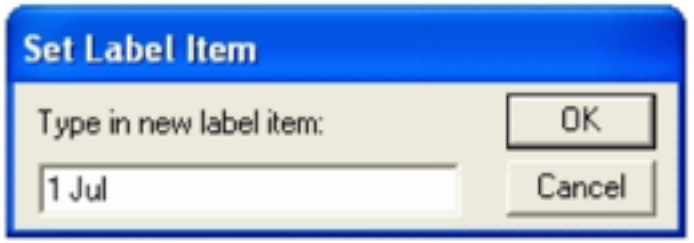
- Structure

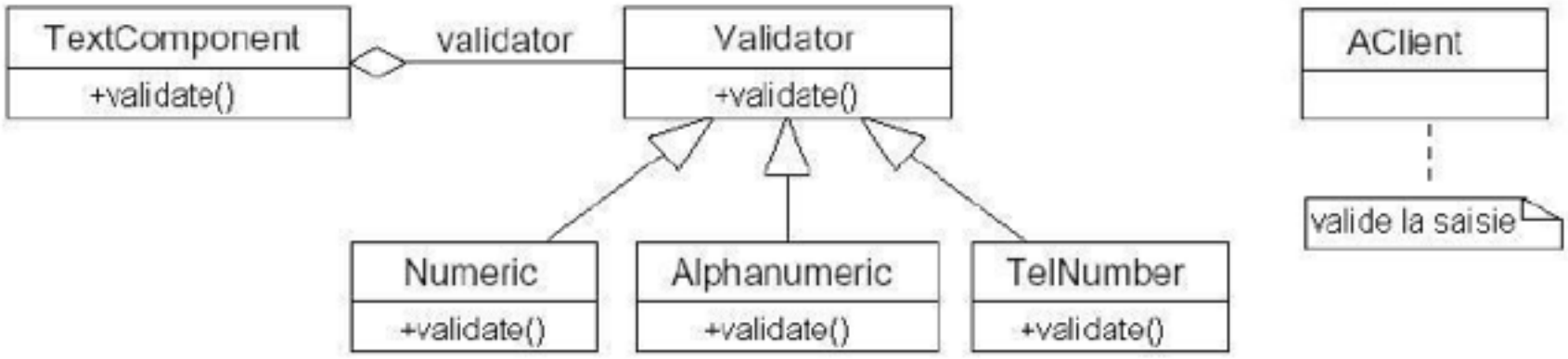


- Participants
  - Context
    - maintient une référence à l'objet Strategy
    - peut définir une interface qui permet à l 'objet Strategy d'accéder à ses données
  - Strategy déclare une interface commune à tous les algorithmes
  - ConcreteStrategy implémente l'algorithme en utilisant l'interface Strategy
- Collaborations
  - Le Context envoie les requêtes de ses clients à l'une de ses stratégies.
  - Les clients créent la classe concrète qui implémente l'algorithme.
  - Puis ils passent la classe au Context.
  - Enfin ils interagissent avec le Context exclusivement.

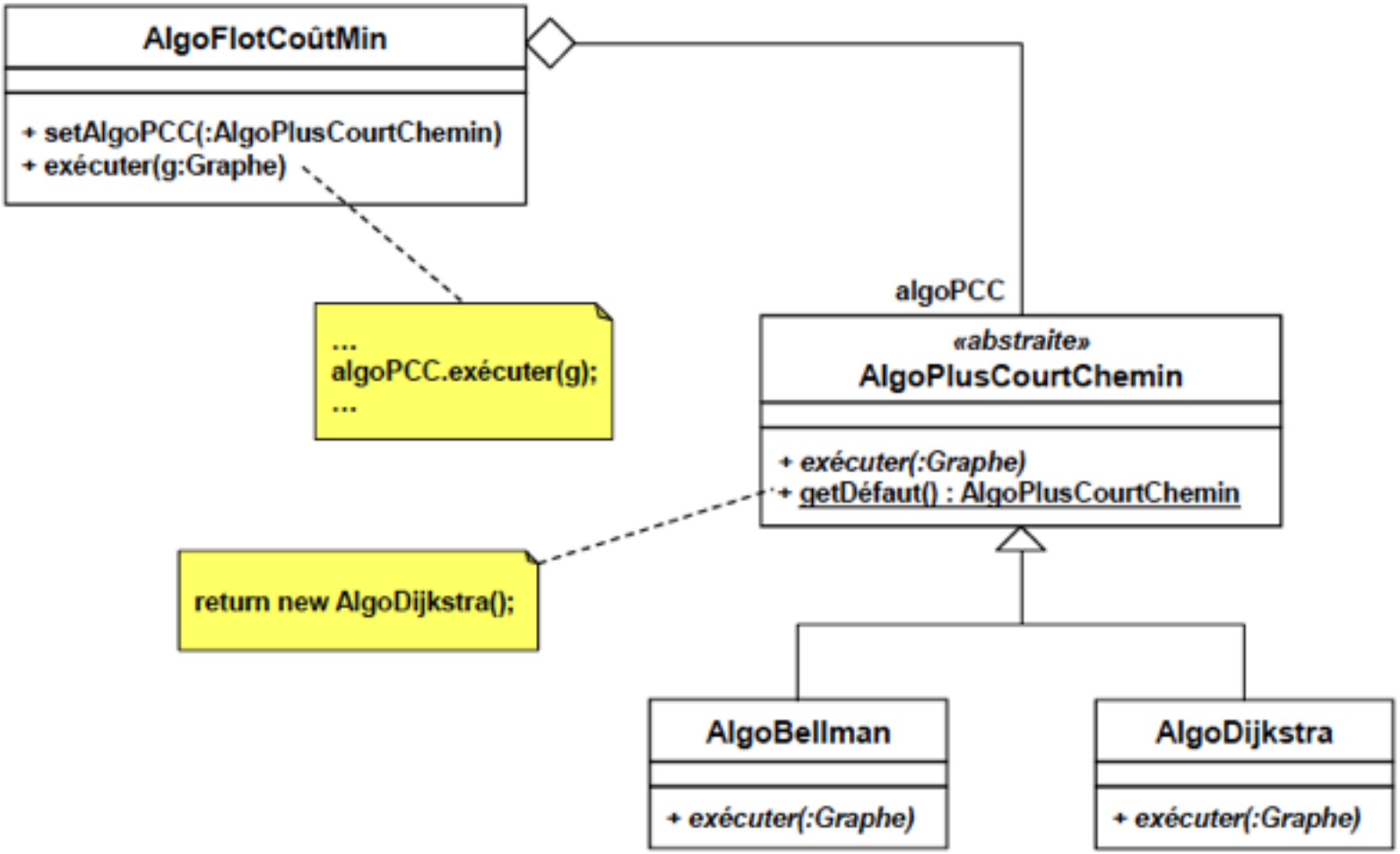
Exemple

- **Validation des données dans un dialogue**
  - Plusieurs stratégies de validation selon le type de données : numériques, alphanumériques





Exemple


AlgoFlotCoûtMin
+ setAlgoPCC(:AlgoPlusCourtChemin)
+ exécuter(g:Graphe)
...
algoPCC.exécuter(g);
...
algoPCC
«abstraite»
AlgoPlusCourtChemin
+ exécuter(:Graphe)
+ getDéfaut() : AlgoPlusCourtChemin
return new AlgoDijkstra();
AlgoBellman
+ exécuter(:Graphe)
AlgoDijkstra
+ exécuter(:Graphe)

Pattern Strategy

- **Conséquences**
  - Avantages
    - Les familles d'algorithmes peuvent être rangées hiérarchiquement
    - Alternative au sous-classement. Permet d'obtenir des combinaisons d'algorithmes en limitant le sous-classement (du context)
    - Élimination des boucles conditionnelles
    - Choix d'implémentations.
  - Inconvénients
    - Les clients doivent connaître les différentes stratégies et comprendre les nuances entre chacune afin de choisir la bonne.
    - Coût de communication supplémentaire : Imaginons une interface Strategy avec beaucoup de méthodes. Chaque sous-classe connaît ses méthodes mais peut être qu'elle ne les utilisera pas
    - Le nombre d'objets peut augmenter beaucoup. Pour y remédier, certaines stratégies peuvent être implantées dans des objets partagés (Flyweight)

Pattern Strategy

- **Implémentation**
  - Définir les interfaces des classes Strategy et du Context :
    - Les stratégies concrètes doivent généralement avoir un accès efficace aux données du contexte.
    - Toutes ces informations pourraient être passées en paramètres à la fonction de la stratégie: Risque de passer de l'information inutile
    - Le contexte pourrait se passer lui-même en paramètre à la méthode de la stratégie et la stratégie pourrait alors interroger le contexte pour obtenir seulement les informations nécessaires: Augmente le couplage entre les stratégies et le contexte.

Pattern Strategy

- **Implémentation**
  - Rendre les objets stratégies optionnels.
    - Dans les cas où il est envisageable de ne pas avoir de stratégie, le contexte peut définir un comportement par défaut qui est utilisé si aucune stratégie n'est spécifiée.
    - Les clients n'ont pas à travailler avec les stratégies si le comportement par défaut leur convient

PATRONS DE COMPORTEMENT

Template Method

Motivation

- Supposons qu'on a une classe PlainTextDocument définit ainsi :

```
public class PlainTextDocument {
...
public void printPage (Page page) {
printPlainTextHeader(); // Unique à PlainTextDocument
System.out.println(page.body());
printPlainTextFooter(); // Unique à PlainTextDocument
}
...
}
```

- Par la suite on écrit la classe HtmlTextDocument :

```
public class HtmlTextDocument {
...
public void printPage (Page page) {
printHtmlTextHeader(); // Unique à HtmlTextDocument
System.out.println(page.body());
printHtmlTextFooter(); // Unique à HtmlTextDocument
}
...
}
```

- La méthode printPage dans PlainTextDocument et HtmlTextDocument sont semblables!

Généraliser

- Mettre la méthode printPage dans une super-classe et permettre aux classes PlainTextDocument et HtmlTextDocument de fournir leurs propres implémentations pour imprimer l'entête et le bas de page

```
public abstract class TextDocument {
...
public final void printPage (Page page) {
printTextHeader();
printTextBody(page);
printTextFooter();
}
public abstract void printTextHeader();
public final void printTextBody(Page page) {
System.out.println(page.body());
}
public abstract void printTextFooter();
...
}
```

```
public class PlainTextDocument extends TextDocument
{
...
public void printTextHeader () {
// Code for header plain text header here.
}
public void printTextFooter () {
// Code for header plain text footer here.
}
...
}
```

Généraliser

- Mettre la méthode printPage dans une super-classe et permettre aux classes PlainTextDocument et HtmlTextDocument de fournir leurs propres implémentations pour imprimer l'entête et le bas de page

```
public abstract class TextDocument {
...
public final void printPage (Page page) {
printTextHeader();
printTextBody(page);
printTextFooter();
}
public abstract void printTextHeader()
public final void printTextBody(Page pag    {
System.out.println(page.body());
}
public abstract void printTextFooter();
...
}
```

```
public class PlainTextDocument extends TextDocument
{
...
public void printTextHeader () {
// Code for header plain text header here.
}
public void printTextFooter () {
// Code for header plain text footer here.
}
...
```

C'est une template méthode

Pattern Template Method

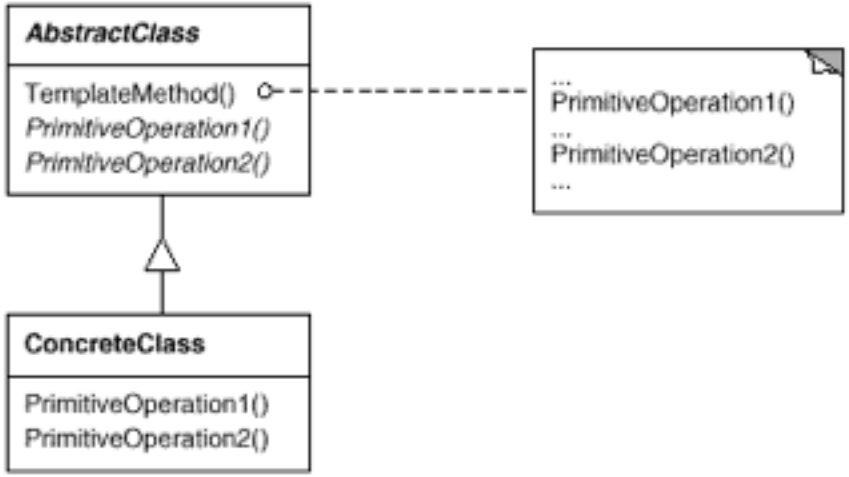
- Intention
  - Définir le squelette d'un algorithme dans une opération, et laisser les sous-classes définir certaines étapes.
  - Permet de redéfinir des parties de l'algorithme sans avoir à modifier celui-ci
- Synonymes
  - Patron de méthode
- Utilisation connus
  - Pratiquement dans toutes les classes abstraites
- Patrons associés
  - Factory Method (souvent appelé par les Template méthodes)
  - Strategy: Template Method utilise l'héritage pour varier les comportement, Strategy utilise la délégation.

Pattern Template Method

- **Problème (Quand l'utiliser)**
  - Pour implanter les aspects invariants d'un algorithme une seule fois et laisser les sous-classes définir les portions variables.
  - Pour situer les comportements communs dans une classe afin d'augmenter la réutilisation de code et éviter la duplication du code
  - Pour contrôler les extensions des sous-classes.

Pattern Template Method

- Structure

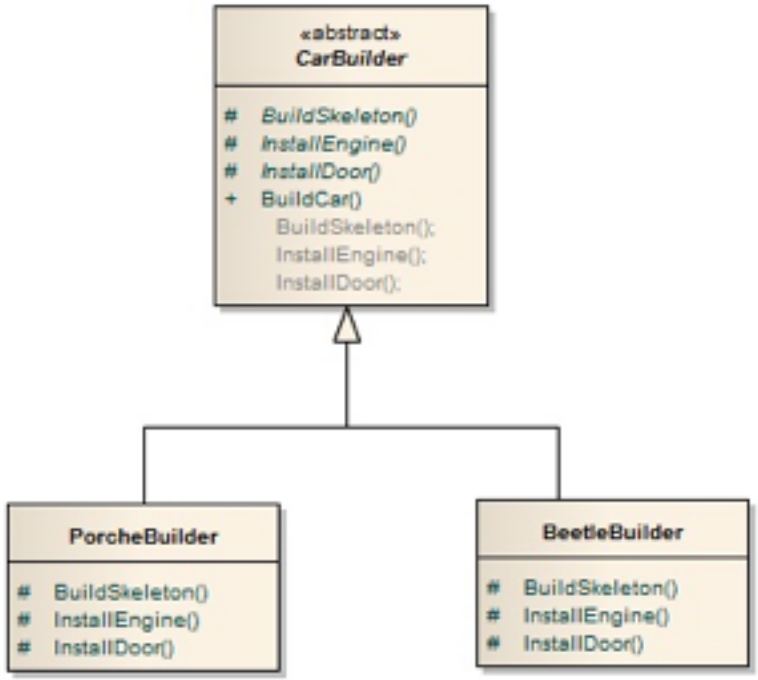


- Participants
  - AbstractClass
    - définit les opérations primitives abstraites que les sous-classes concrètes redéfinissent pour implémenter des étapes d'un algorithme.
    - implémente un patron de méthode définissant le squelette d'un algorithme. Le patron de méthode appel les opérations primitives ainsi que les opérations définies dans AbstractClass ou ceux d'autres objets.
  - ConcreteClass
    - implémente les opérations primitives pour effectuer les étapes de l'algorithme selon les particularités de la sous-classe
- Collaborations
  - ConcreteClass compte sur AbstractClass pour implémenter les étapes invariantes de l'algorithme

Exemple


```
public class Program{
    static void Main(String[] args){
        CarBuilder c = new PorcheBuilder();
        c.BuildCar();

        c = new BeetleBuilder();
        c.BuildCar();
    }
}
 abstract class CarBuilder{
    protected abstract void BuildSkeleton();
    protected abstract void InstallEngine();
    protected abstract void InstallDoor();

   //Template Method that specifies the general logic
    public void BuildCar() {
        BuildSkeleton();
        InstallEngine();
        InstallDoor();
    }
}

 class PorcheBuilder extends CarBuilder{
    protected void BuildSkeleton(){
       System.out.println("Building Porche Skeleton");
    }
    protected void InstallEngine() {
       System.out.println("Installing Porche Engine");
    }
    protected void InstallDoor(){
       System.out.println("Installing Porche Door");
    }
}
```

```
class BeetleBuilder extends CarBuilder {
    protected void BuildSkeleton() {
       System.out.println("Building Beetle Skeleton");
    }

    protected void InstallEngine(){
       System.out.println("Installing Beetle Engine");
    }
    protected void InstallDoor() {
       System.out.println("Installing Beetle Door");
    }
}
```



Pattern Template Method

- **Conséquences**
  - mène à une inversion de contrôle: la classe parente appelle les opérations d'une sous-classe («Principe Hollywood: ne nous appelez pas, nous vous appellerons »)
  - favorise la réutilisation de code
  - appel plusieurs types d'opérations:
    - opérations concrètes (dans les classes concrètes ou les classes clientes)
    - opérations concrètes dans la classes abstraites (méthodes généralement utiles aux sous classes)
    - opérations primitives (i.e. abstraites, qui doivent êtres redéfinies)
    - méthodes de fabriques (voir Factory Method)
    - des opération hook, qui définissent le comportement par défaut et que les sous classes peuvent redéfinir si nécessaire.
  - permet d'imposer des règles de surcharge
  - il faut sous-classer pour spécialiser le comportement.

Pattern Template Method

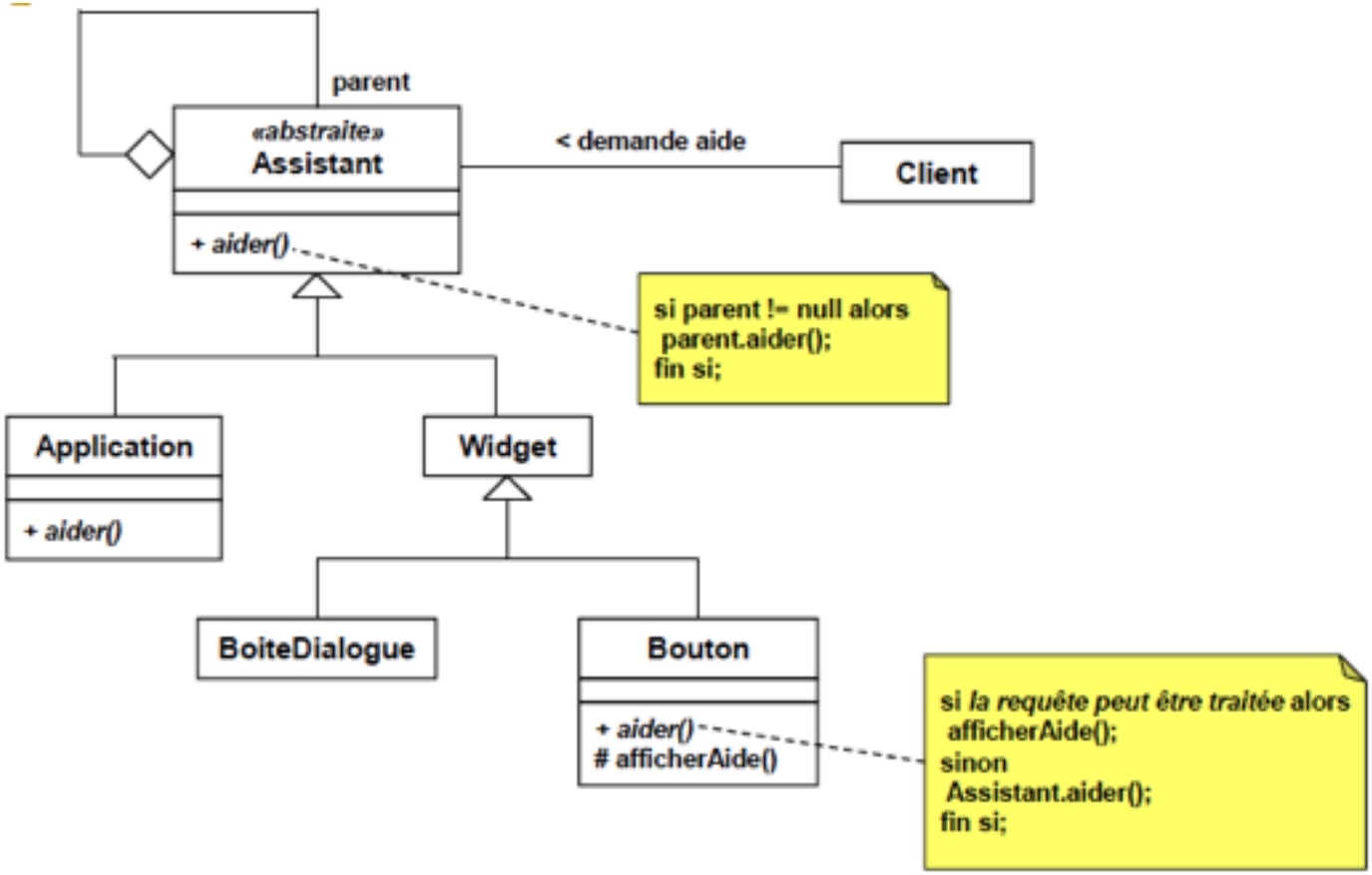
- **Implémentation**
  - Les opérations qui doivent être redéfinies par les sous-classes doivent être déclarées abstraites. Si la méthode Template ne devrait pas être remplacée par une sous-classe, il convient de la déclarer finale.
  - Minimiser le nombre d'opérations que les sous classes doivent redéfinir, sinon l'utilisation de Template méthode risque d'être fastidieux.
  - Convention de noms (préfix do- ou faire-) pour identifier les opérations qui doivent être redéfinies.

PATRONS DE COMPORTEMENT

Chain of Responsability

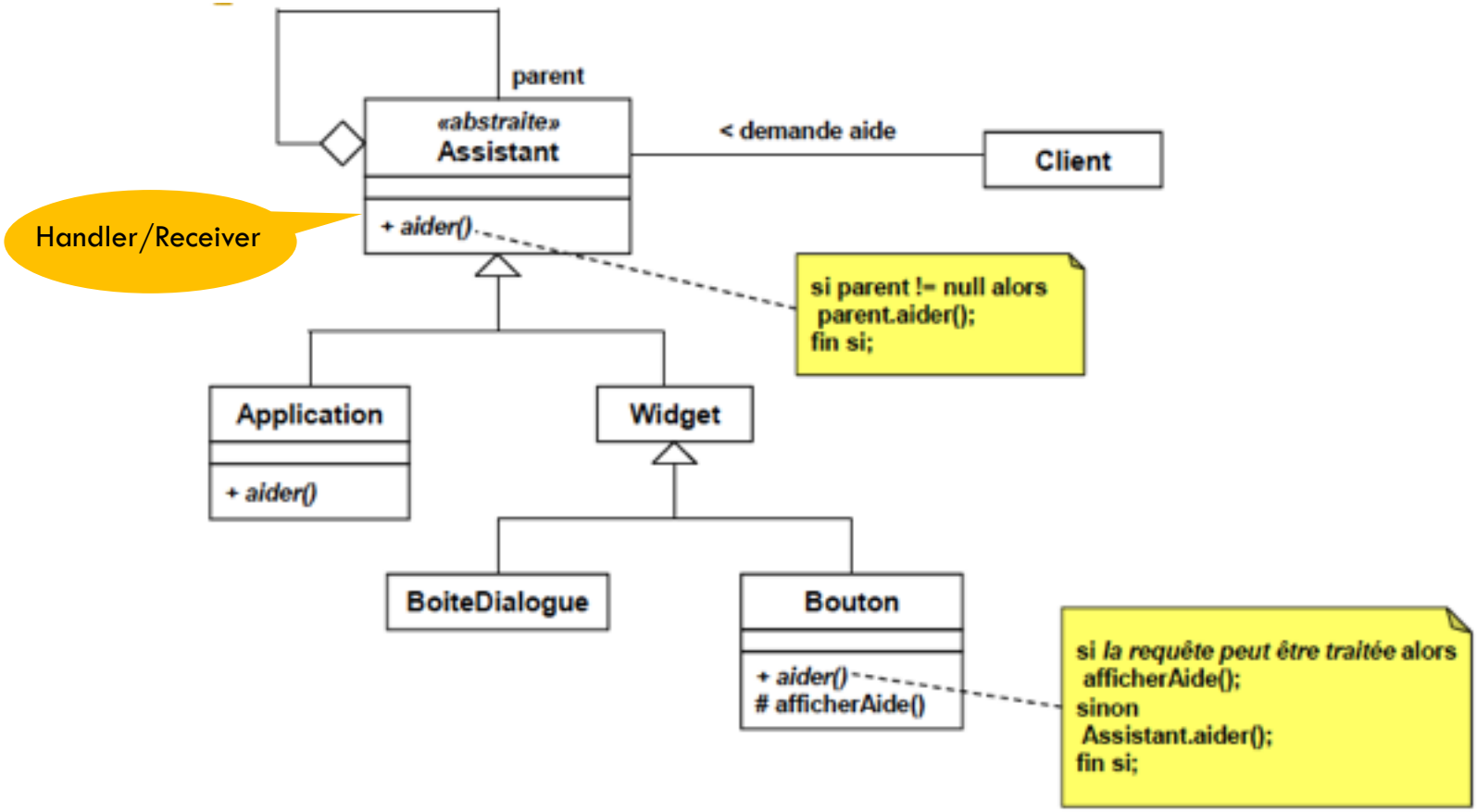
Motivation

- Système d'aide dans une application avec GUI



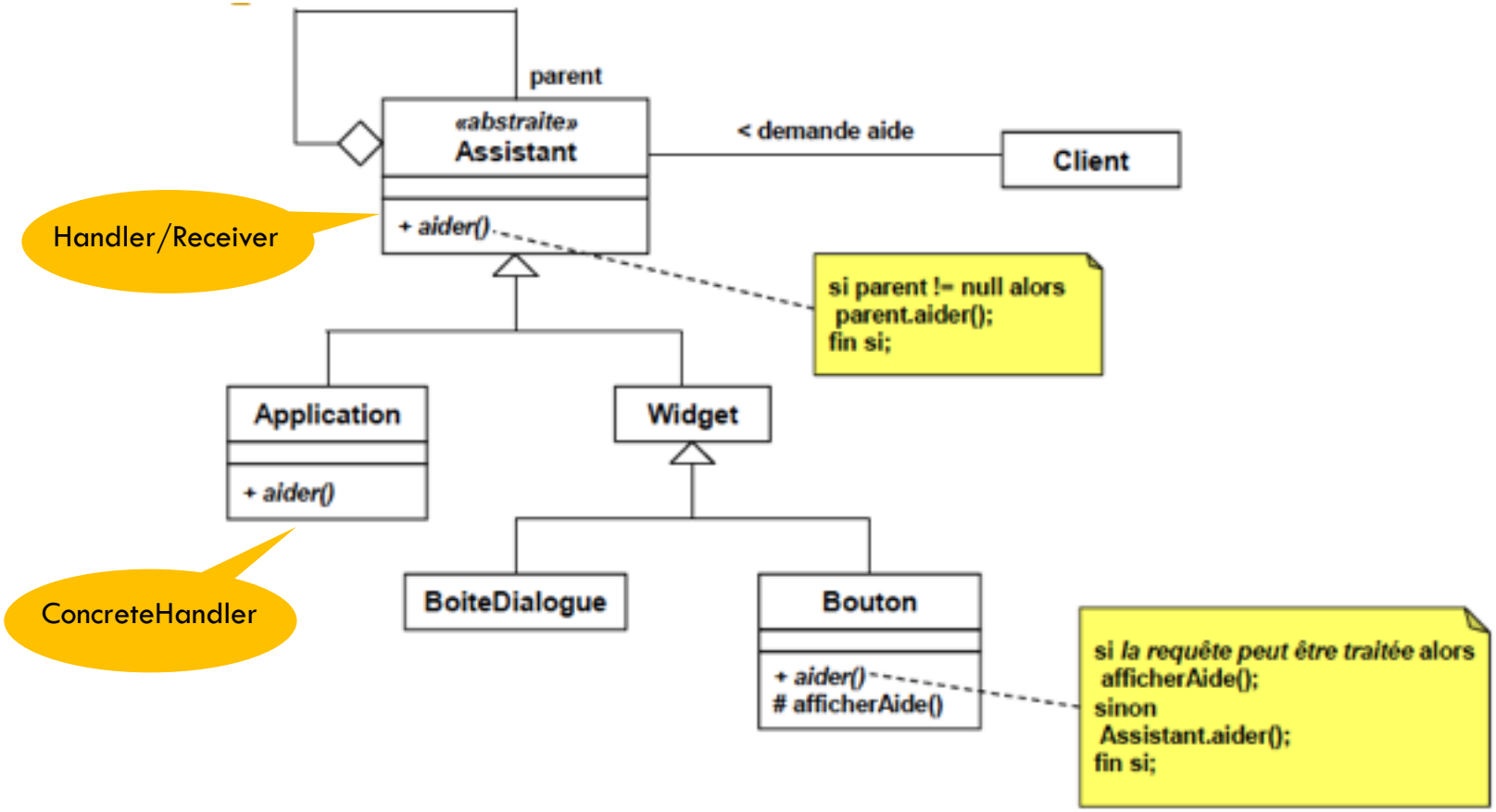
Motivation

- Système d'aide dans une application avec GUI



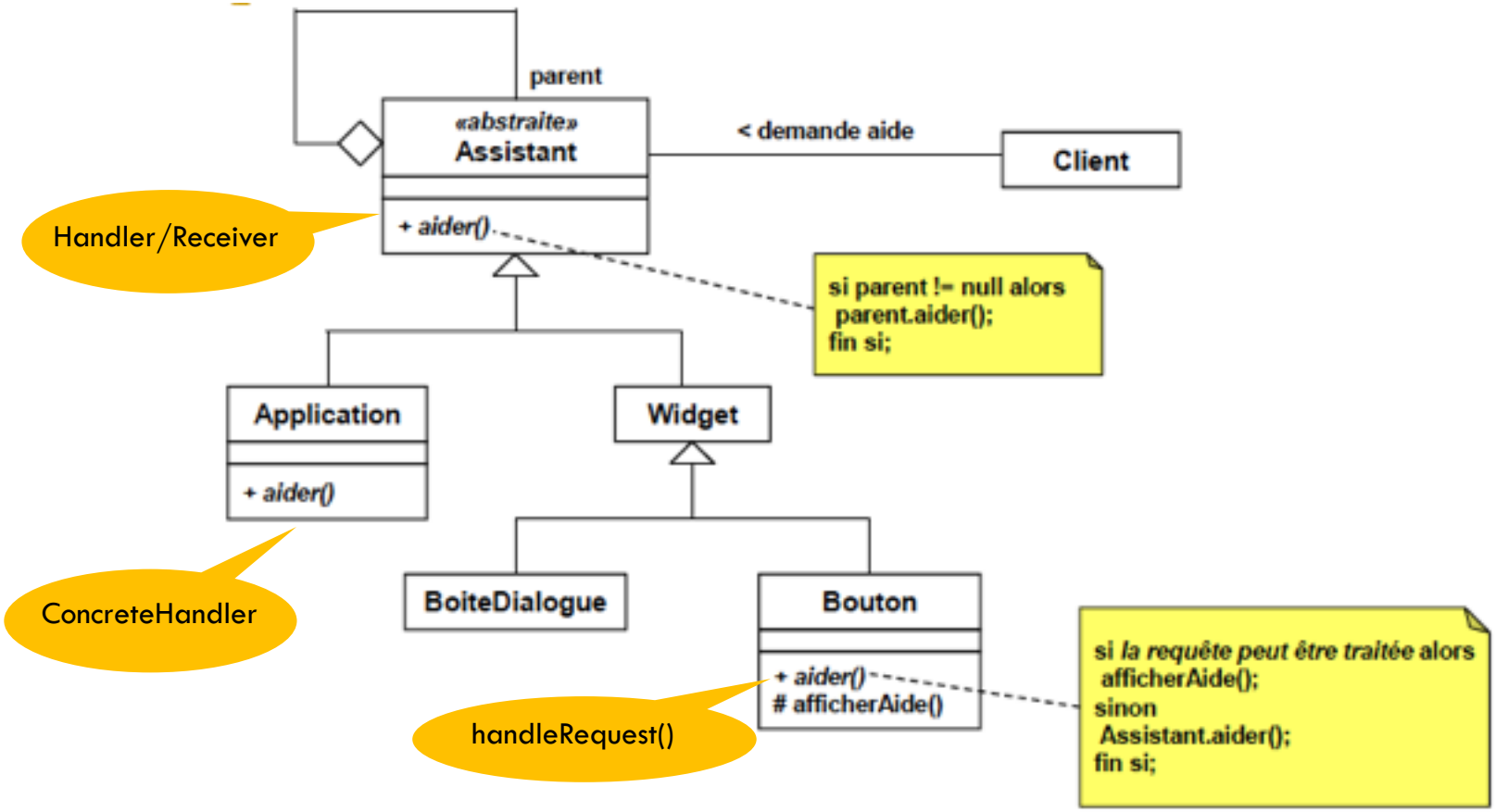
Motivation

- Système d'aide dans une application avec GUI



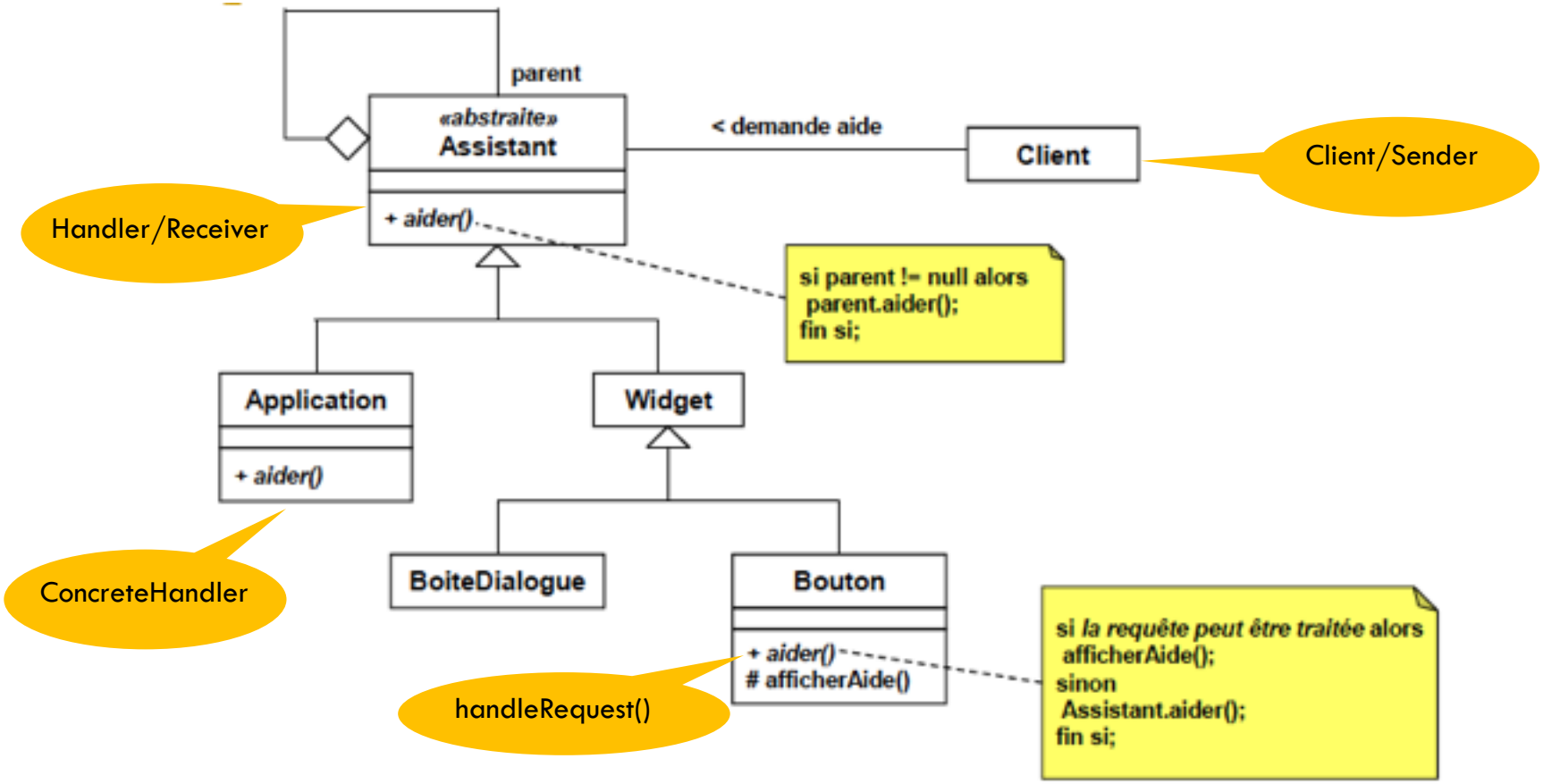
Motivation

- Système d'aide dans une application avec GUI



Motivation

- Système d'aide dans une application avec GUI



Pattern Chain of Responsability

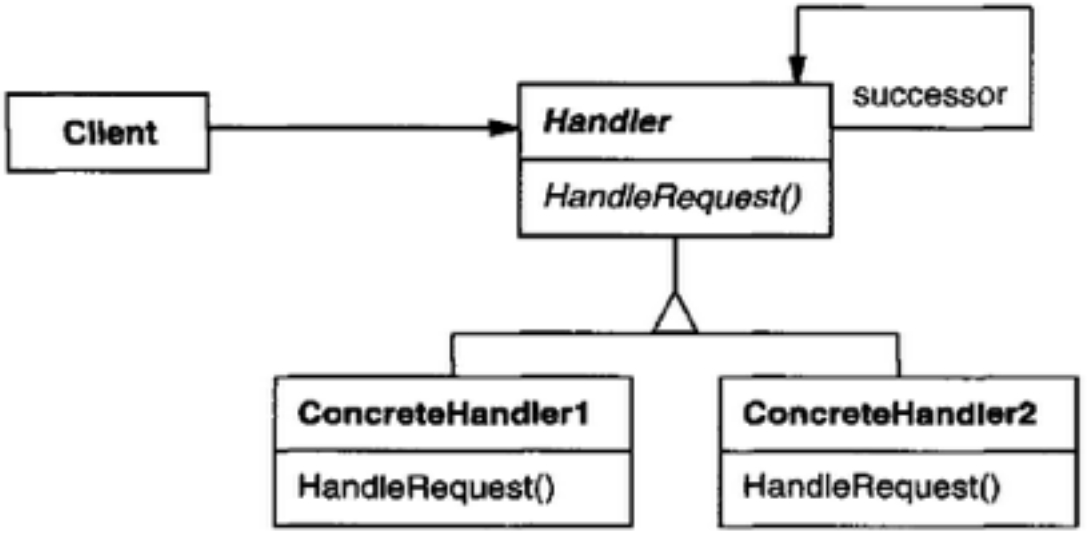
- Intention
  - Évite le couplage entre l'émetteur d'une requête et ses récepteurs en donnant la possibilité à plus d'un objets de traiter la requête.
  - Les objets récepteurs sont chaînés et la requête traverse la chaîne jusqu'à ce qu'elle soit traitée.
- Synonymes
  - Chaine de responsabilité
- Utilisation connus
  - Gestion des évènements dans les interfaces graphiques; Même principe que les exceptions Java : récepteurs = catch
- Patrons associés
  - Composite

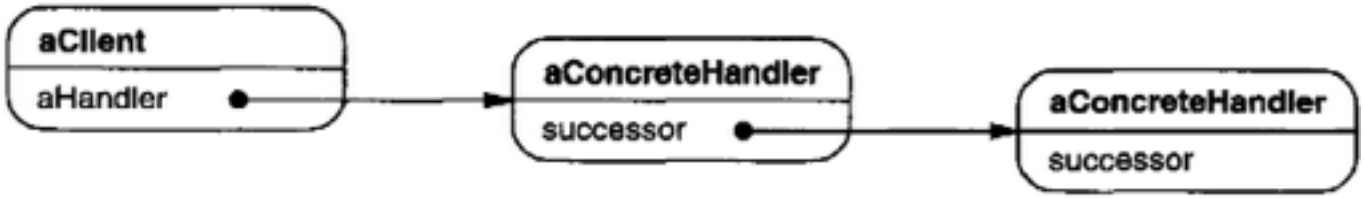
Pattern Chain of Responsability

- **Problème (Quand l'utiliser)**
  - Plus d'un objets peuvent traiter une requête et que l'objet qui traitera la requête n'est pas connu a priori (déterminé dynamiquement).
  - On veut adresser une requête à plus d'un objets, sans spécifier explicitement le récepteur
  - L'ensemble d'objets qui peut traiter la requête doit être spécifié dynamiquement.

Pattern Chain of Responsability

- Structure





Pattern Chain of Responsability

- **Participants**
  - Handler
    - Définit une interface de traitement des requêtes
    - Implémente les liens successeurs (optionnel)
  - ConcreteHandler
    - Traite les requêtes dont il est responsable.
    - Peut accéder à son successeur.
    - Si le ConcreteHandler peut traiter la requête, il le fait; sinon, il la transmet à son successeur.
  - Client
    - Emet la requête à un objet ConcreteHandler de la chaîne.
- **Collaborations**
  - Quand un client émet une requête, la demande se propage le long de la chaîne jusqu'à ce qu'un objet ConcreteHandler prend la responsabilité de la traiter.

Pattern Chain of Responsability

- **Conséquences**
  - *Réduit le couplage*. Ce patron évite à l'émetteur de devoir connaître l'objet récepteur. L'émetteur ne connaît que le point d'entrée de la chaîne.
  - *Augmente la flexibilité avec laquelle les responsabilités peuvent être assignées aux objets*. On peut modifier la chaîne en cours d'exécution et changer dynamiquement la façon dont les requêtes seront traitées.
  - *La réception et le traitement de la requête n'est pas garantie*. Comme il n'y a pas de récepteur explicite, rien ne garantit à l'émetteur que sa requête sera bien traitée.

Pattern Chain of Responsability

- **Implémentation**
  - Implémenter et connecter la chaîne des suivants. Il y a deux possibilités :
    - *Définir une nouvelle série de liens généralement au niveau de la classe abstraite Handler(Récepteur)*. Dans ce cas, la classe *Récepteur* maintient une référence au suivant de la chaîne et implante un comportement par défaut de la méthode traiterRequete() qui consiste à rediriger la requête au suivant.
    - *Utiliser des liens existants*. Dans certains cas, les liens existants peuvent être utilisés. Par exemple, une fonction find() du système de fichiers qui recherche un noeud à partir d'un point de l'arborescence peut se comporter comme une chaîne de responsabilité qui utilise les liens existants du patron Composite pour accomplir sa tâche.

Pattern Chain of Responsability

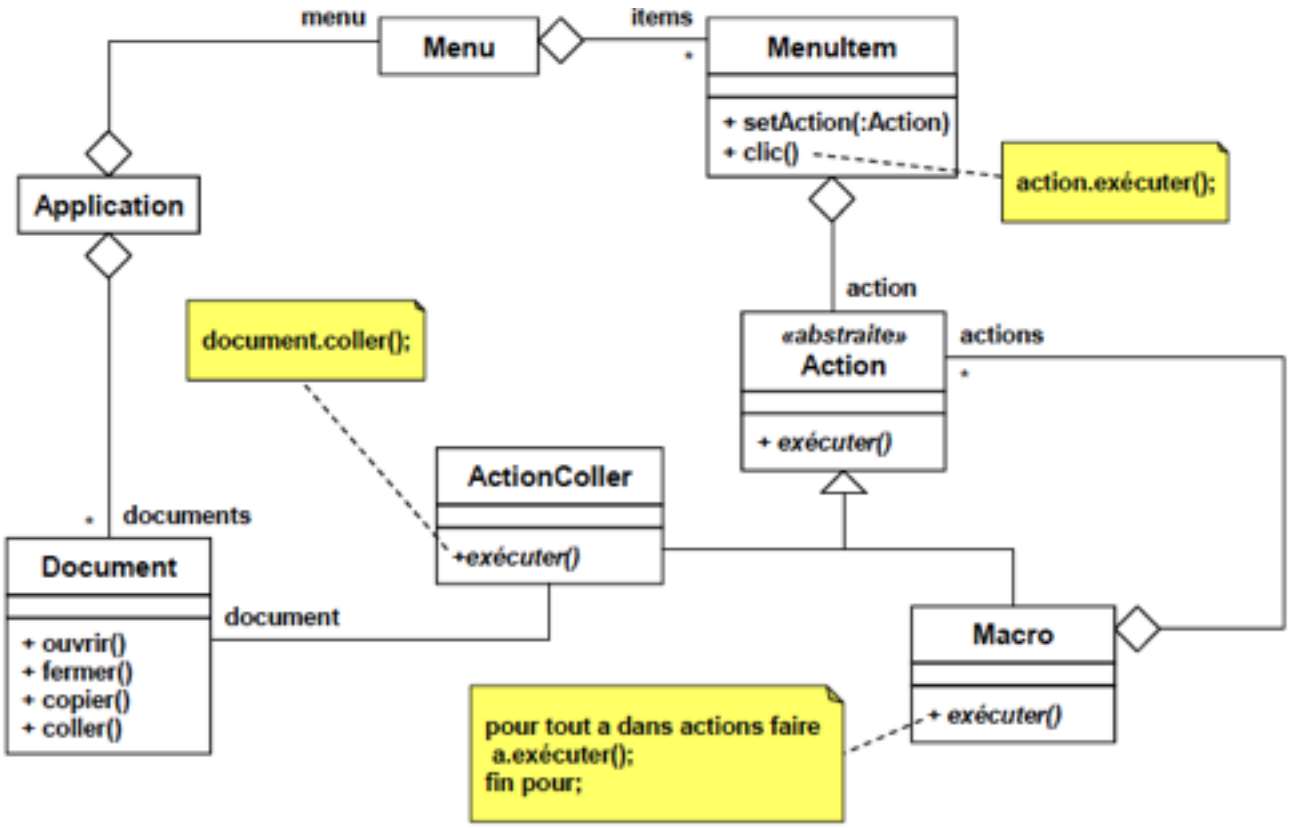
- **Implémentation**
  - Représenter les requêtes
    - Dans le cas le plus simple, une requête est une simple invoquation de la méthode *traiterRequete*(). Ceci est simple et rapide, mais ne permet de représenter qu'un ensemble fixe de requêtes.
    - La méthode de traitement peut utiliser un code de requête (paramètre) pour identifier les types de requêtes. Cette technique est plus flexible, mais moins sûre et nécessite souvent des énoncés conditionnels.
    - On peut utiliser le patron Command. Le patron Command permet d'encapsuler des requêtes dans des objets. Chaque objet requête peut ainsi être passé le long de la chaîne jusqu'à ce qu'il soit traité. Un récepteur peut connaître le type de la requêtes en interrogeant l'objet requête.
  - Dans Smalltalk, possibilité de transfert automatique de requête en utilisant le mécanisme « doesNotUnderstand ».

PATRONS DE COMPORTEMENT

Command

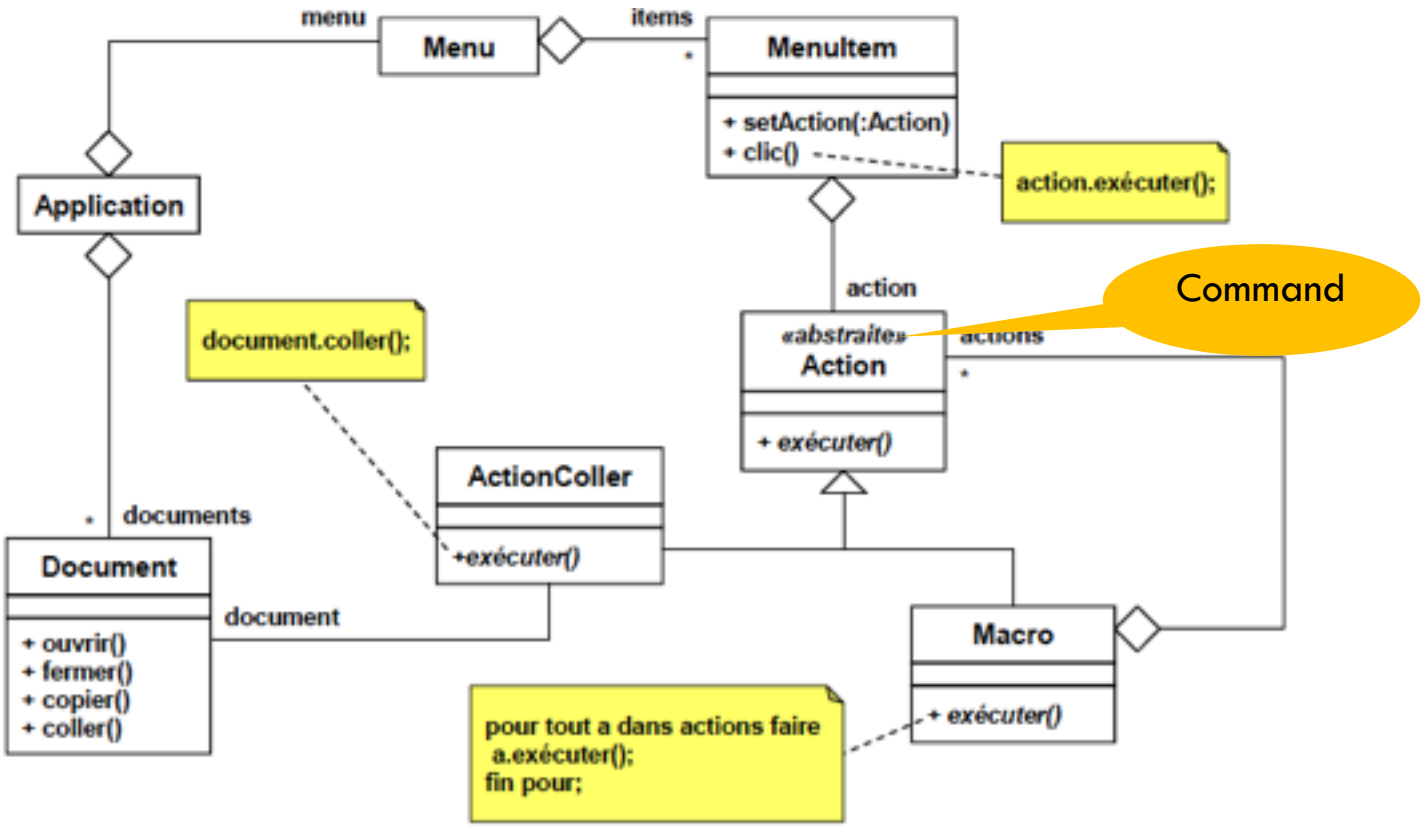
Motivation

- Un toolkit de génération de menu doit être paramétrable pour que l'application générée puisse implémenter les actions associées à chaque composant de l'IG:
  - Associé une classe Action comportant une méthode exécuter qui implémentera le code pour chaque application générée



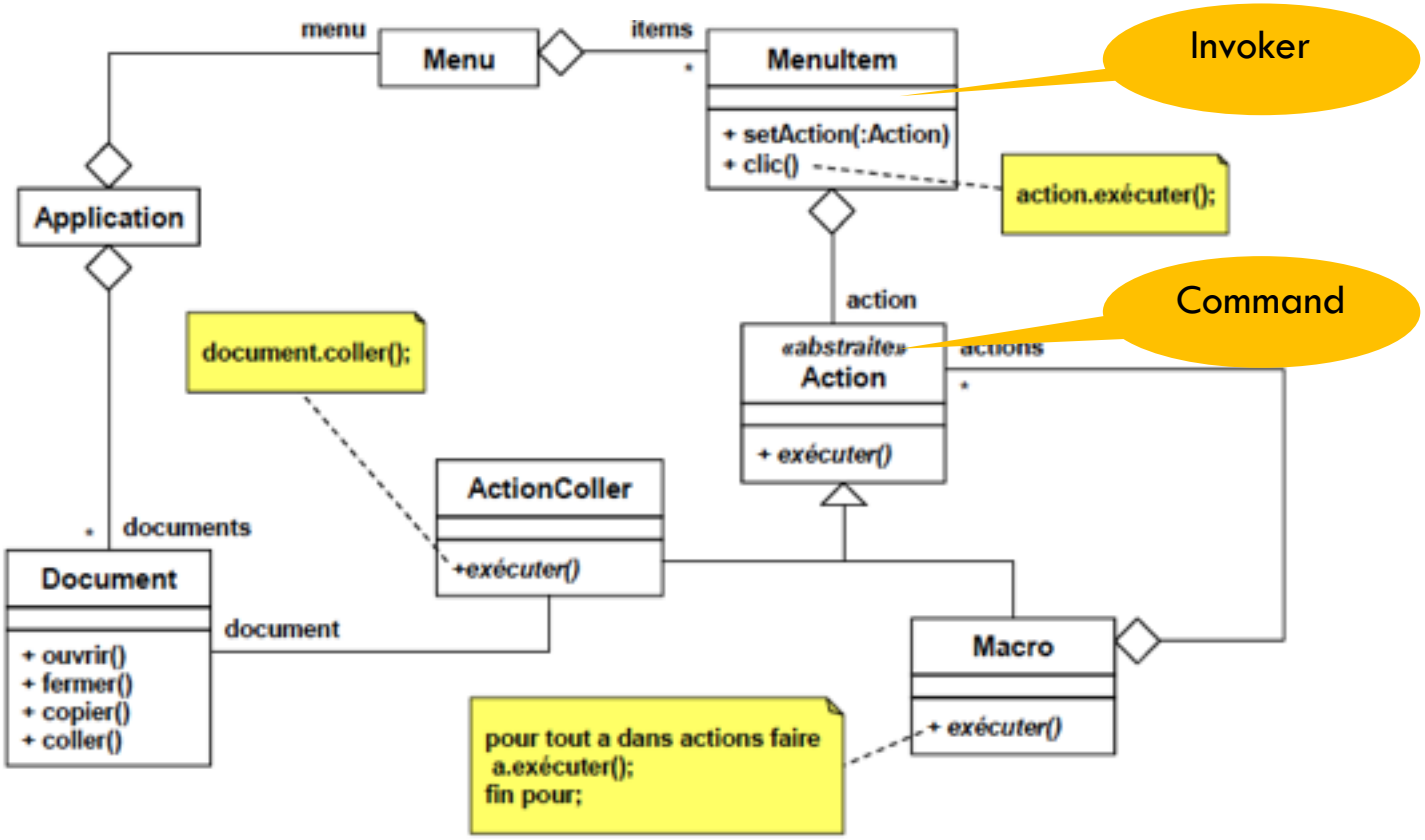
Motivation

- Un toolkit de génération de menu doit être paramétrable pour que l'application générée puisse implémenter les actions associées à chaque composant de l'IG:
  - Associé une classe Action comportant une méthode exécuter qui implémentera le code pour chaque application générée



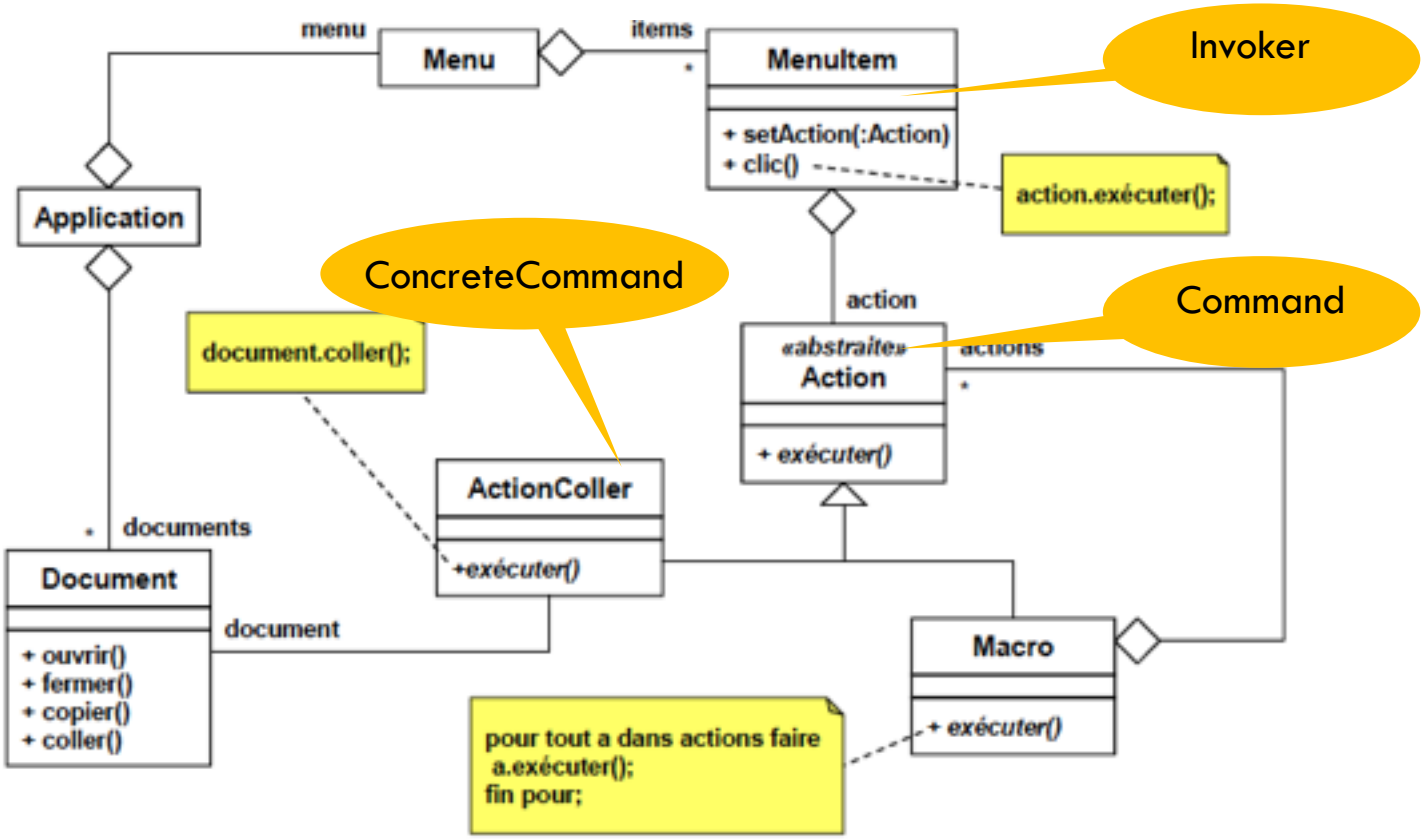
Motivation

- Un toolkit de génération de menu doit être paramétrable pour que l'application générée puisse implémenter les actions associées à chaque composant de l'IG:
  - Associé une classe Action comportant une méthode exécuter qui implémentera le code pour chaque application générée



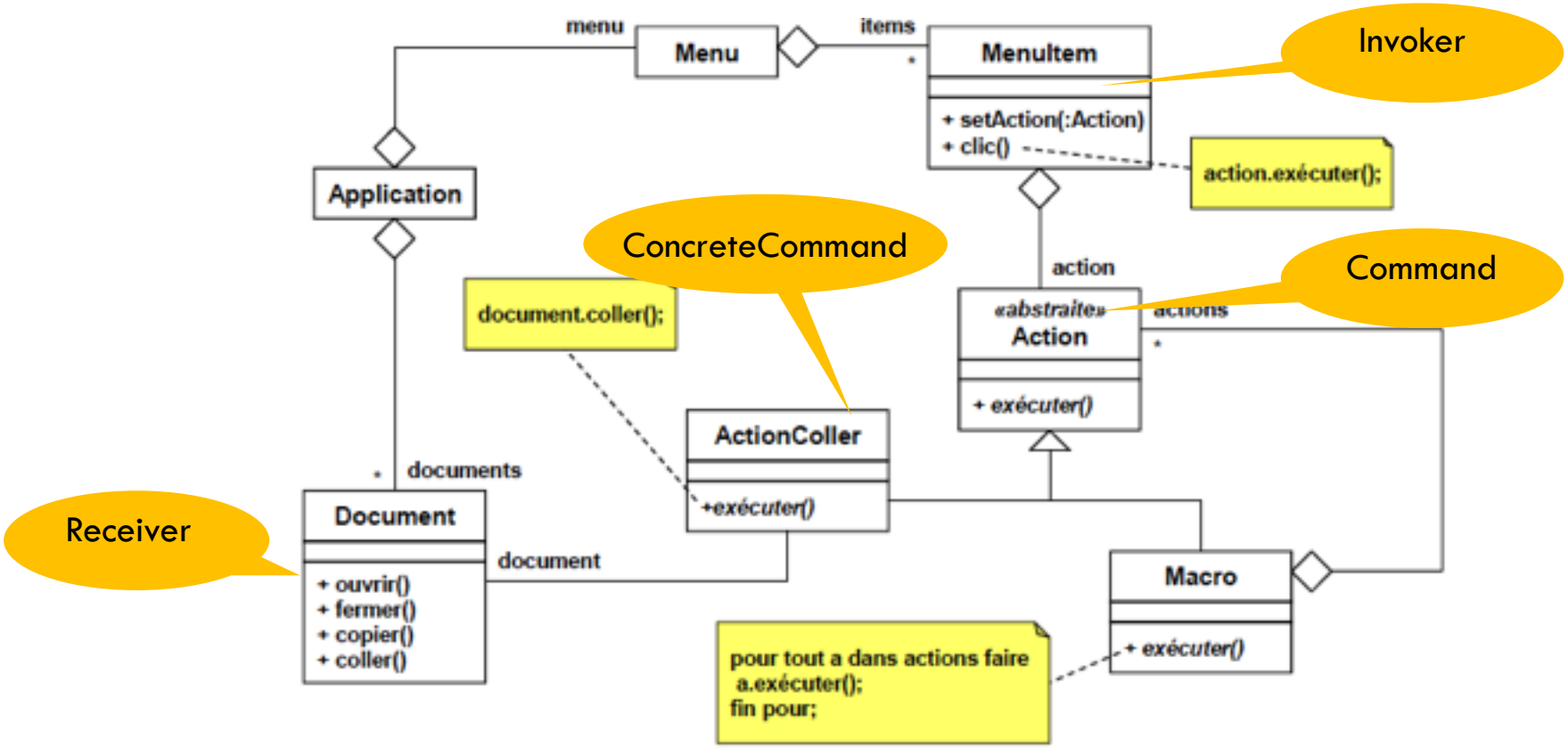
Motivation

- Un toolkit de génération de menu doit être paramétrable pour que l'application générée puisse implémenter les actions associées à chaque composant de l'IG:
  - Associé une classe Action comportant une méthode exécuter qui implémentera le code pour chaque application générée



Motivation

- Un toolkit de génération de menu doit être paramétrable pour que l'application générée puisse implémenter les actions associées à chaque composant de l'IG:
  - Associé une classe Action comportant une méthode exécuter qui implémentera le code pour chaque application générée



Pattern Command

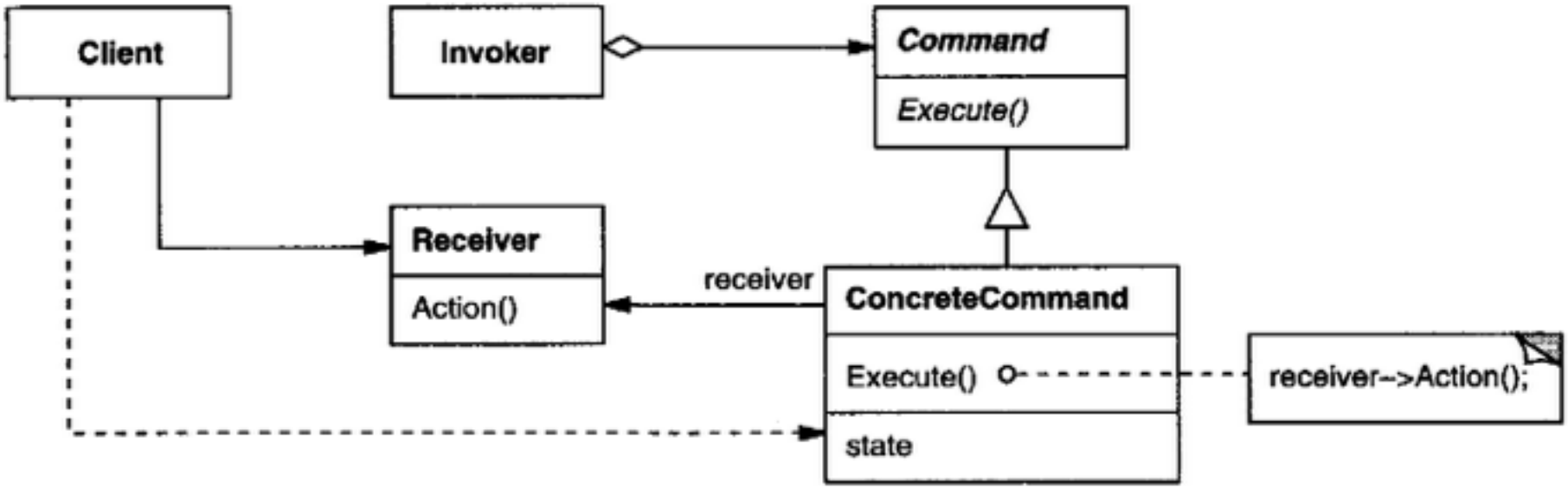
- Intention
  - Encapsuler une requête (ou un traitement) comme un objet
  - Permet de paramétrer le client avec différentes requêtes, gérer ou traiter des files de requêtes,
  - Permettre l'annulation de requêtes.
- Synonymes
  - Action, Transaction
- Utilisation connus
  - Les Menus des GUI, Undo
- Patrons associés
  - Composite, Memento, Prototype

Pattern Command

- **Problème (Quand l'utiliser)**
  - Paramétrer les objets par des actions à faire. Command est une alternative orientée objet aux fonctions de callback de la programmation procédurale (une fonction qui s'inscrit quelque part pour être appelé à un stade ultérieur, ex. pointeurs de fonctions en C++)
  - Manipuler les requêtes, les exécuter à différents moments
  - Défaire des requêtes (undo): nécessite que l'interface implémente la méthode Annuler qui permet d'inverser l'effet de l'appel précédent de « execute ». Les commandes exécutées sont enregistrés dans une liste d'historique
  - Mémoriser les modifications : permet le rétablissement du système en cas de crash.
  - Structurer le système en opérations de haut niveau (transaction): le patron commande offre un moyen de modéliser les transactions. Les commandes ont une interface commune, permettant d'appeler toutes les transactions de la même façon, ce qui permet également d'ajouter facilement de nouvelles transactions.

Pattern Command

- Structure



Pattern Command

- **Participants**
  - **Command**
    - déclare une interface pour exécuter une opération
  - **ConcreteCommand**
    - définit un lien entre l'objet *Receiver* et l'action
    - implémente execute() en invoquant la méthode correspondante de l'objet *Receiver*
  - **Client**
    - crée l'objet *ConcreteCommand* et positionne son *Receiver*
  - **Invoker**
    - demande à l'objet *Command* de traiter la requête
  - **Receiver**
    - sait comment effectuer la ou les opérations associées à la requête

Pattern Command

- **Conséquences**
  - Découpler les objets qui invoquent une action de ceux qui l'exécutent
  - Les Commandes sont encapsulées dans des objets qui sont manipulées comme n'importe quel autre objet
  - Il est possible d'assembler des commandes en une commande composée (Ex. MacroCommande) généralement en utilisant le patron Composite.
  - Il est facilite d'ajouter de nouvelles commandes sans changer les classes existantes.

Pattern Command

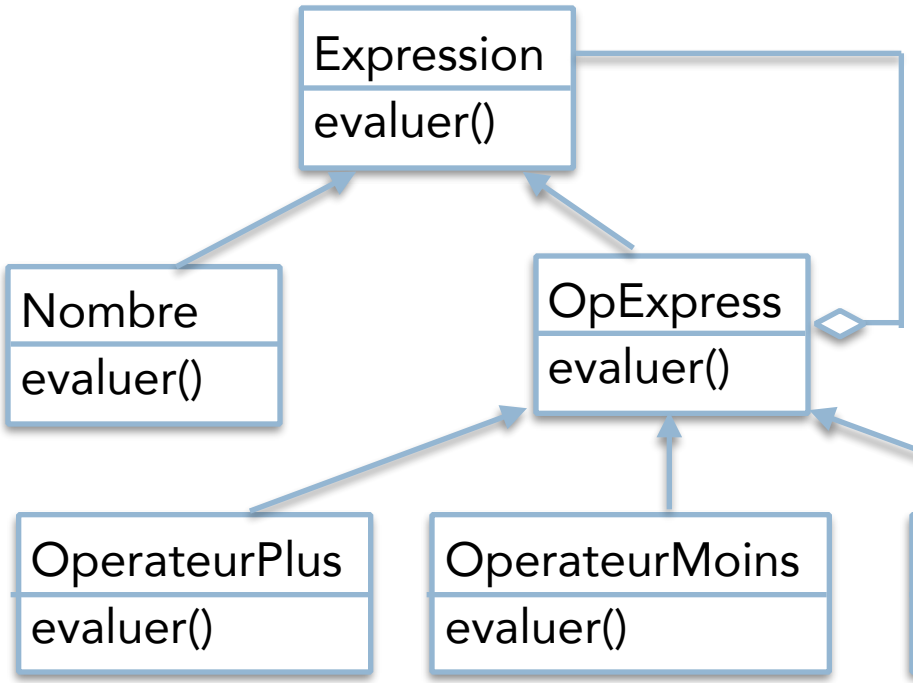
- **Implémentation**
  - Pour supporter les undo et redo, il faut définir les opérations correspondantes et la classe ConcreteCommand doit stocker, pour cela, certaines informations
  - Un historique des commandes doit être conservé pour réaliser les undo à plusieurs niveaux

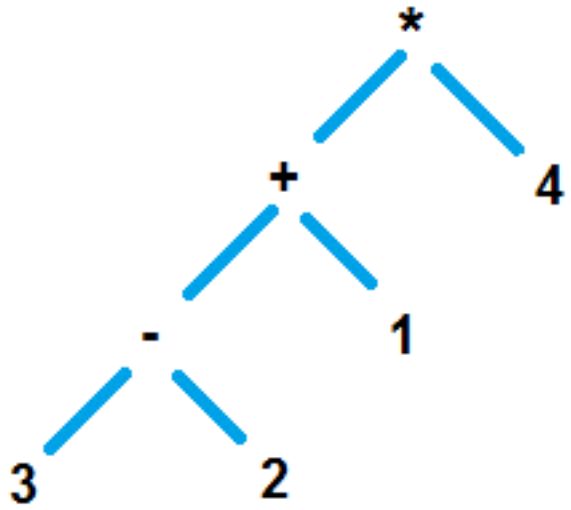
PATRONS DE COMPORTEMENT

Interpreter

Motivation

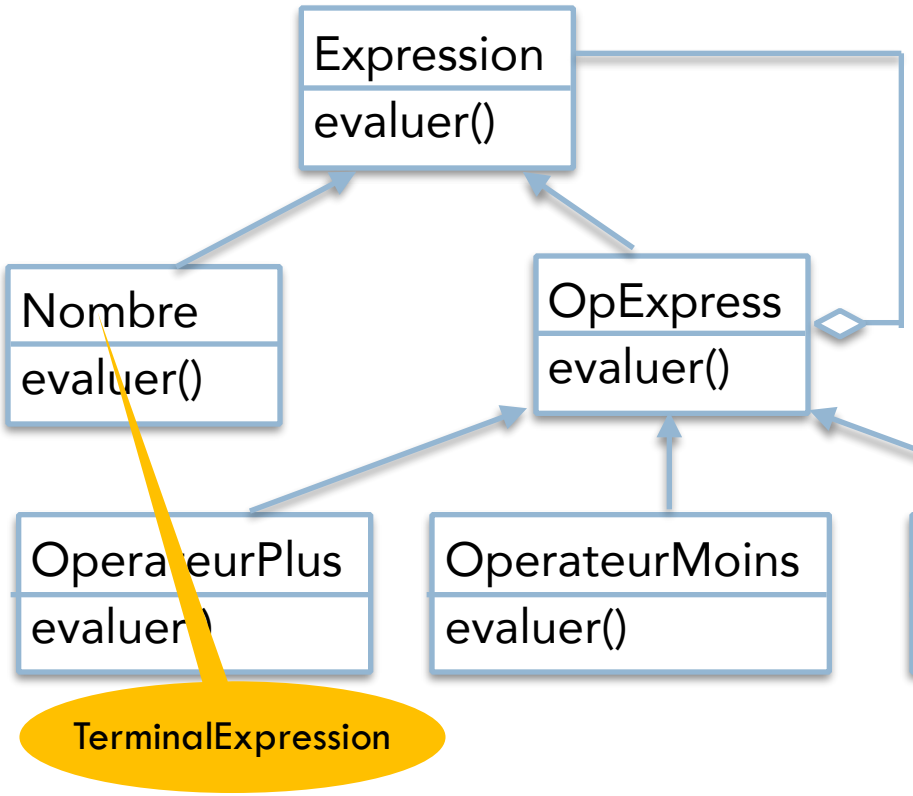
- Évaluer une expression arithmétique en notation polonaise post-fixée.
  - Exemple: (3 – 2 + 1) * 4 est équivalente à 4 3 2 – 1 + *

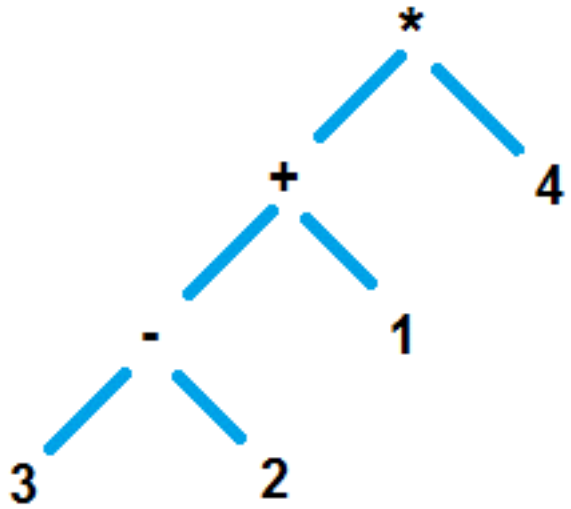




Motivation

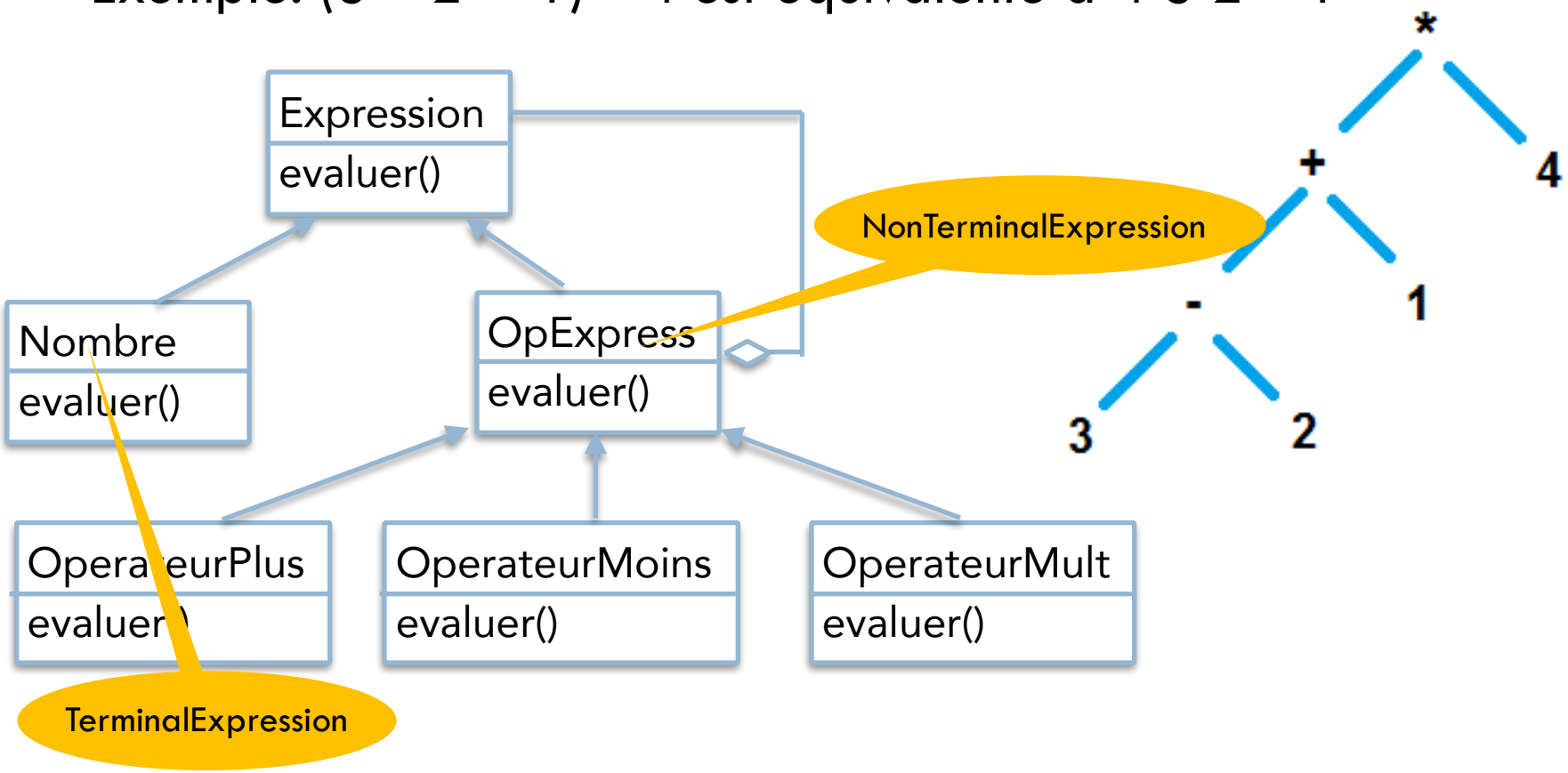
- Évaluer une expression arithmétique en notation polonaise post-fixée.
  - Exemple: (3 – 2 + 1) * 4 est équivalente à 4 3 2 – 1 + *





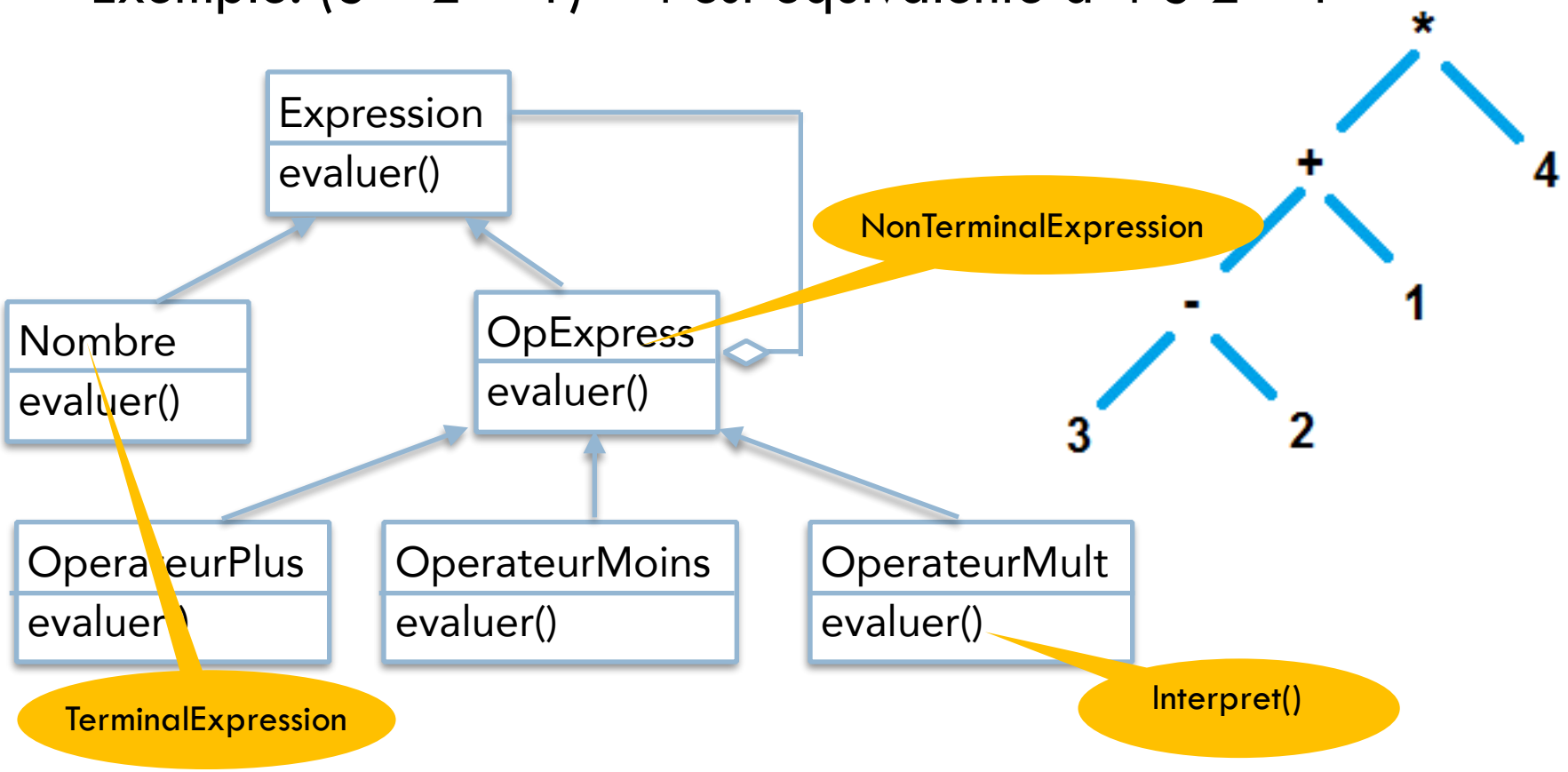
Motivation

- Évaluer une expression arithmétique en notation polonaise post-fixée.
  - Exemple: (3 – 2 + 1) * 4 est équivalente à 4 3 2 – 1 + *



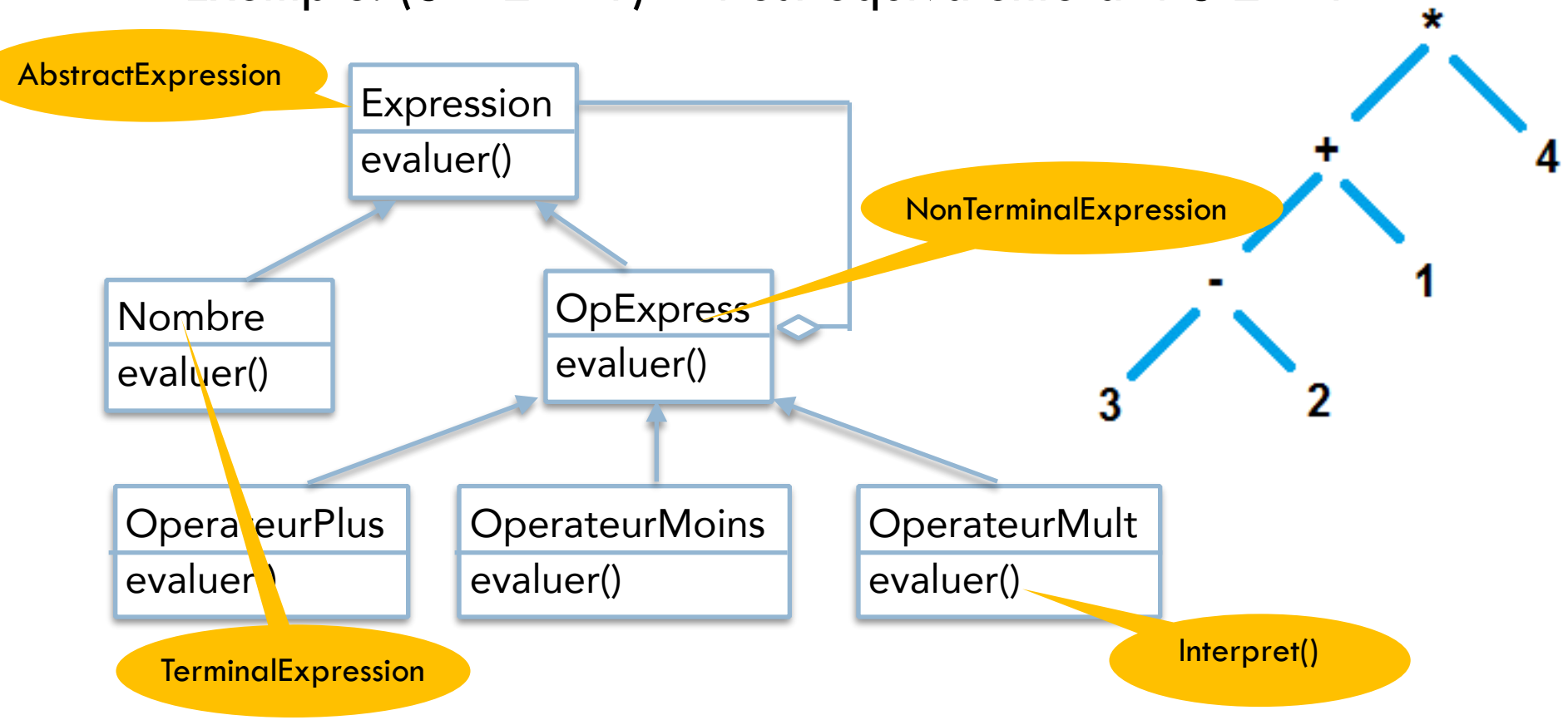
Motivation

- Évaluer une expression arithmétique en notation polonaise post-fixée.
  - Exemple: (3 – 2 + 1) * 4 est équivalente à 4 3 2 – 1 + *



Motivation

- Évaluer une expression arithmétique en notation polonaise post-fixée.
  - Exemple: (3 – 2 + 1) * 4 est équivalente à 4 3 2 – 1 + *



Pattern Interpreter

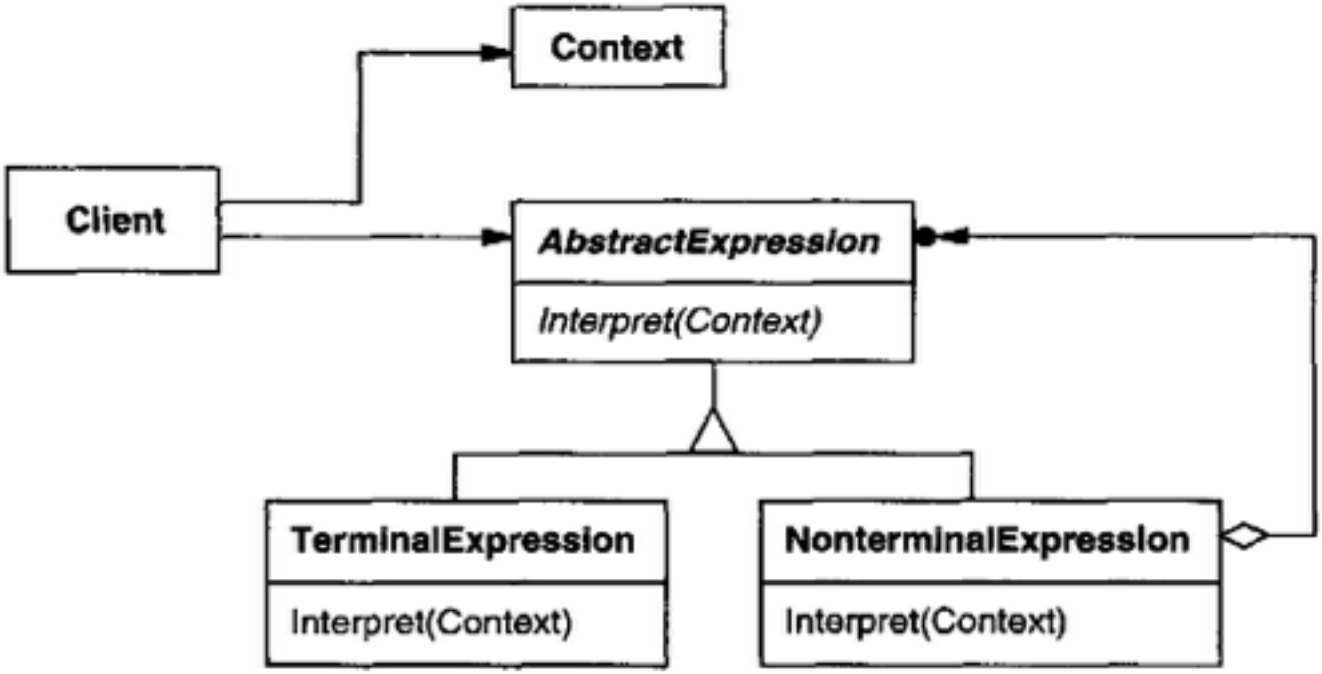
- Intention
  - Définit une représentation de la grammaire d'un langage ainsi qu'un interprète capable de la manipuler
- Synonymes
  - Interprète, Little Language
- Utilisation connus
  - Les compilateurs des langages de POO
- Patrons associés
  - Composite, Flyweight, Iterator, Visitor

Pattern Interpreter

- **Problème (Quand l'utiliser)**
  - Utiliser le pattern Interpreter lorsqu'il y a un langage à interpréter, et il est possible de représenter des déclarations dans des arbres de syntaxe abstraite.
    - *La grammaire du langage doit être simple:* sinon ça risque de devenir ingérable (utiliser des parsers (analyseurs))
    - *L'efficacité n'est pas un souci majeur :* Les interprètes les plus efficaces ne sont pas généralement mis en œuvre directement en arbres d'analyse, mais passent d'abord par une phase de transformation. Par exemple, les expressions régulières sont souvent transformés en machines d'état. Cette phase peut être implémentée par le pattern Interpreter (ce qui le rend toujours applicable).

Pattern Interpreter

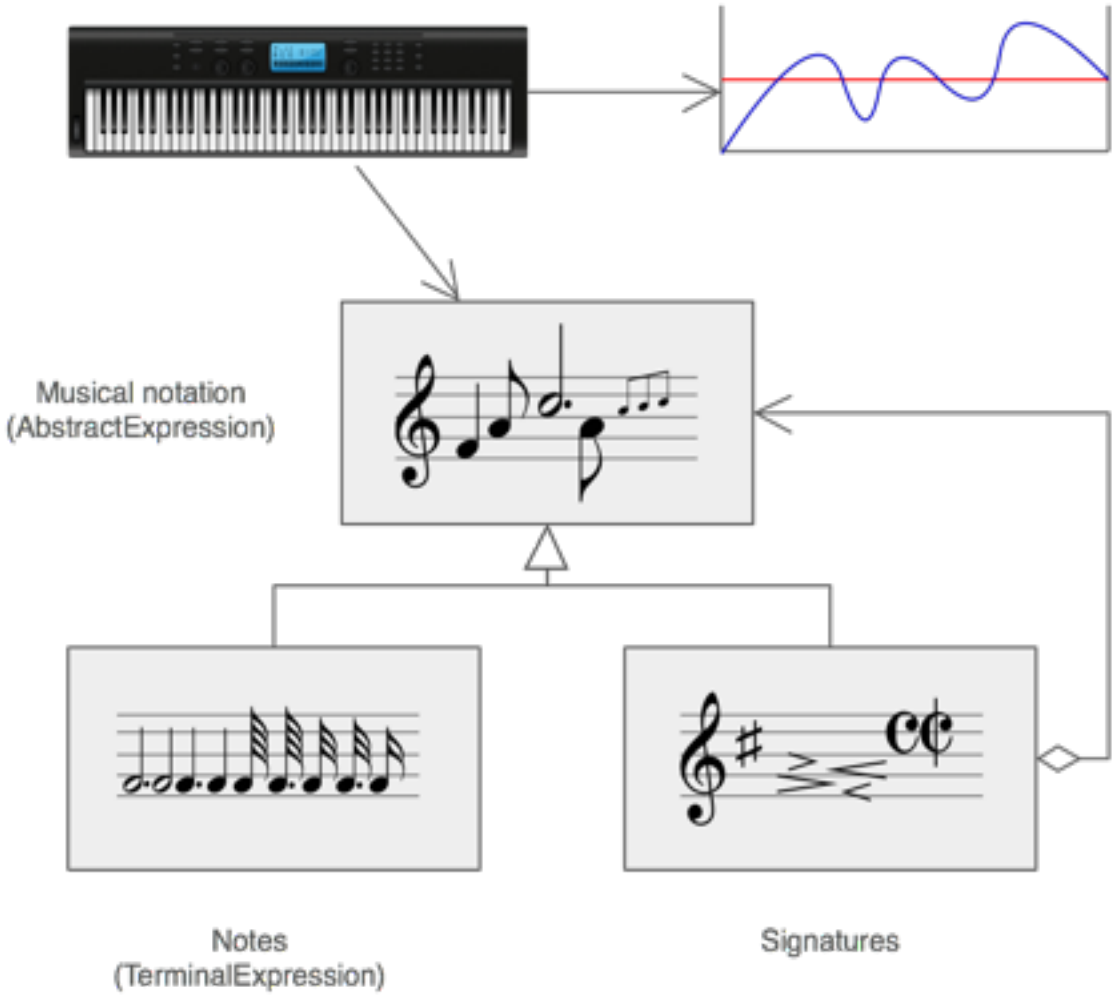
- Structure



Pattern Interpreter

- **Participants**
  - AbstractExpression
    - Déclare une méthode *Interpret* abstraite qui est commune à tous les nœuds de l'arbre de syntaxe abstraite.
  - TerminalExpression
    - Implémente une méthode *Interpret* associé aux symboles terminaux dans la grammaire.
    - Une instance est nécessaire pour chaque symbole terminal dans une phrase.
  - NonterminalExpression
    - Une telle classe est nécessaire pour chaque règle $R := R_1 R_2 \ldots R_n$ dans la grammaire.
    - Maintient des variables d'instance de type AbstractExpression pour chacun des symboles R_i.
    - Implémente une méthode *Interpret* pour les symboles non terminaux dans la grammaire. *Interpret* est généralement appelée récursivement sur les variables représentant $R_1 \ldots R_n$
  - Context
    - Contient des informations globales à l'interprète.
  - Client
    - Construit (ou est donnée) un arbre de syntaxe abstraite représentant une phrase dans le langage que la grammaire définit. L'arbre de syntaxe abstraite est construit à partir des instances des classes *NonterminalExpression* et *TerminalExpression*.
    - Appelle la méthode *Interpret*

Exemple


Musical notation
(AbstractExpression)
Notes
(TerminalExpression)
Signatures

Pattern Interpreter

- **Conséquences**
  - Il est facile d'implémenter la grammaire, la modifier ou l'étendre
  - Les grammaires complexes sont difficiles à maintenir (utiliser des générateurs de compilateur)
  - Permet l'ajout de nouvelles façons d'interpréter les expressions. Par exemple, vous pouvez traiter les méthodes « afficher » ou « contrôler le types » d'une expression en définissant une nouvelle méthode sur les classes d'expression. Si vous continuez à créer de nouvelles façons d'interpréter une expression, il est possible d'utiliser le patron Visitor pour éviter de changer les classes de la grammaire.

Pattern Interpreter

- **Implémentation**
  - *Création de l'arbre de syntaxe abstraite*. Le pattern Interpreter n'explique pas comment le créer.
  - *Définir la méthode Interpréter*. Il n'ai pas obligatoire de définir la méthode *Interpréter* dans les classes d'expression. S'il est fréquent de créer un nouvel *interprète*, alors il est préférable d'utiliser le patron Visiteur.
    - Par exemple, une grammaire d'un langage de programmation aura de nombreuses opérations sur les arbres de syntaxe abstraite, comme le contrôle de type, l'optimisation, la génération de code, et ainsi de suite. Il sera préférable d'utiliser un visiteur afin d'éviter de définir ces opérations sur toutes les classes de grammaire.
  - *Partager les symboles terminaux avec le patron Flyweight*. Les grammars dont les phrases contiennent plusieurs occurrences d'un symbole terminal peuvent bénéficier du partage d'un seul exemplaire de ce symbole.

PATRONS DE COMPORTEMENT

Memento

Motivation

- Supposons un jeu avec plusieurs niveaux de difficulté croissante,
- Plus on avance dans les niveaux plus les chances de rencontrer une situation qui mettra fin au jeu augmentent.
- Les joueurs veulent une commande de sauvegarde qui leur permet de mémoriser leur progression pour reprendre le jeu au dernier niveau

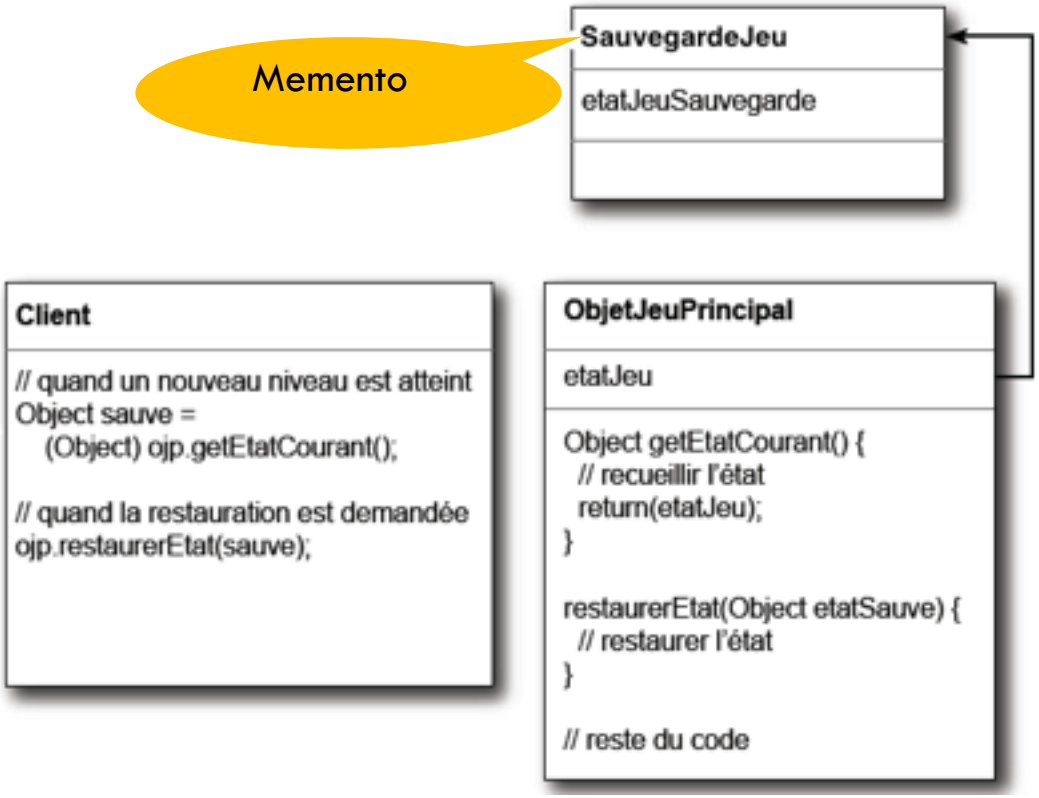
terminé avec succès.

Motivation

- Supposons un jeu avec plusieurs niveaux de difficulté croissante,
- Plus on avance dans les niveaux plus les chances de rencontrer une situation qui mettra fin au jeu augmentent.
- Les joueurs veulent une commande de sauvegarde qui leur permet de mémoriser leur progression pour reprendre le jeu au dernier niveau terminé avec succès.

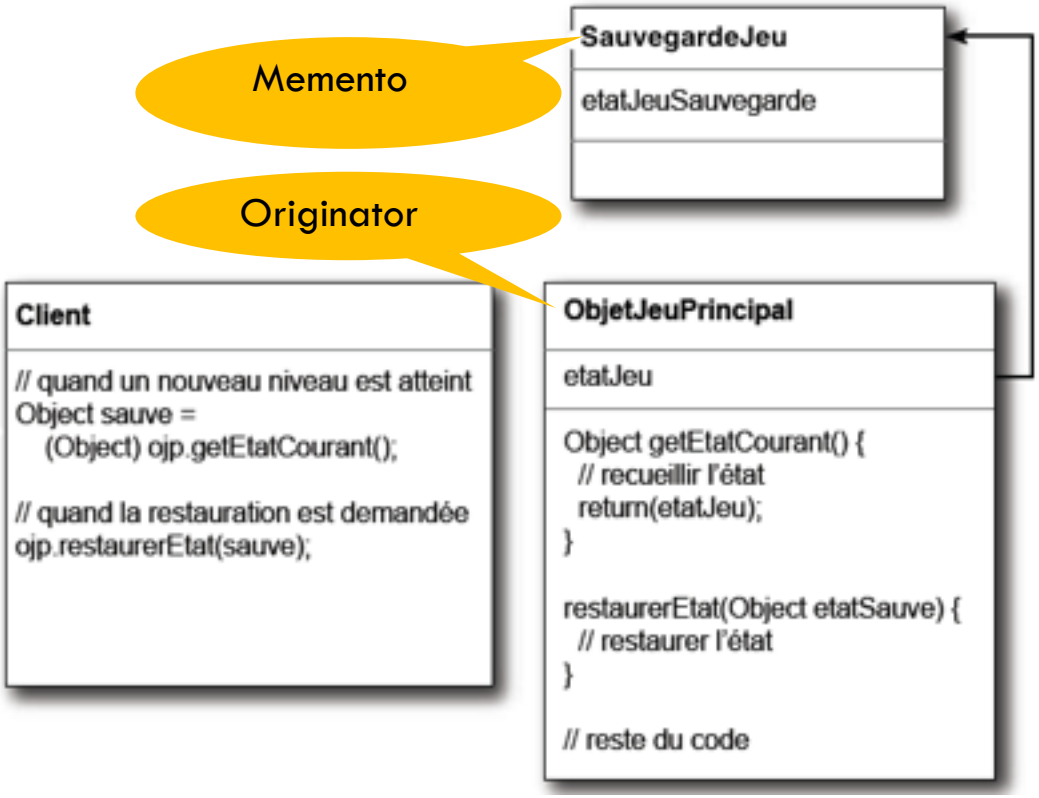
Motivation

- Supposons un jeu avec plusieurs niveaux de difficulté croissante,
- Plus on avance dans les niveaux plus les chances de rencontrer une situation qui mettra fin au jeu augmentent.
- Les joueurs veulent une commande de sauvegarde qui leur permet de mémoriser leur progression pour reprendre le jeu au dernier niveau terminé avec succès.



Motivation

- Supposons un jeu avec plusieurs niveaux de difficulté croissante,
- Plus on avance dans les niveaux plus les chances de rencontrer une situation qui mettra fin au jeu augmentent.
- Les joueurs veulent une commande de sauvegarde qui leur permet de mémoriser leur progression pour reprendre le jeu au dernier niveau terminé avec succès.



Pattern Memento

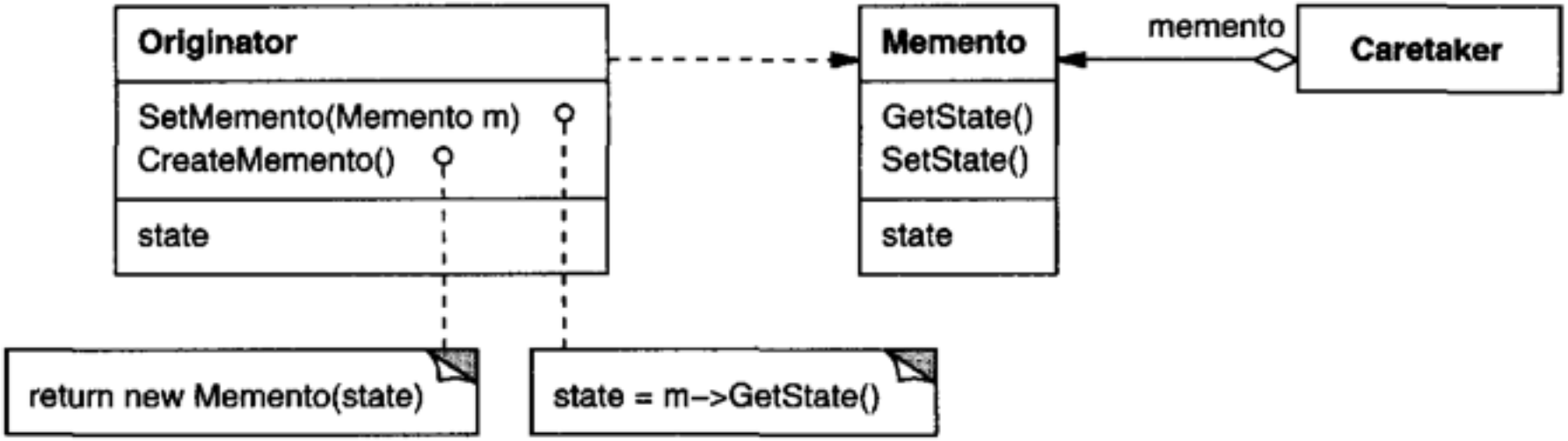
- Intention
  - Transmettre à l'extérieur d'un objet son état interne sans violation de l'encapsulation dans le but de restaurer ultérieurement sont état.
  - Mécanisme de sauvegarde et de restauration d'objets
- Synonymes
  - Token (Jeton), Souvenir, Snapshot
- Utilisation connus
  - Mécanisme d'annulation «undo» parfois complexe
- Patrons associés
  - Command, Itération

Pattern Memento

- **Problème (Quand l'utiliser)**
  - un instantané de tout ou partie d'un objet doit être mémorisé, et
  - l'utilisation d'une interface directe pour atteindre l'état conduirait à rompre l'encapsulation

Pattern Memento

- Structure



Pattern Memento

- **Participants**
  - Memento
    - stocke l'état interne de l'objet Originator. Le Memento peut stocker peu ou beaucoup d'information de l'état interne de l'Originator(créateur) selon la nécessité déterminé par l'Originator(créateur).
    - protège contre les accès par des objets autres que l'Originator. Memento possède effectivement deux interfaces. Caretaker voit une interface étroite : il peut uniquement passer Memento à d'autres objets. Originator, en revanche, voit une interface large, lui permettant d'accéder à toutes les données nécessaires pour restaurer son état précédent. Idéalement, seul l'Originator qui a créé le Memento serait autorisé à accéder à son état interne.
  - Originator (Créateur)
    - crée un Memento contenant un aperçu de son état interne actuel.
    - utilise le memento pour restaurer son état interne.
  - Caretaker (Gardien)
    - Est responsable de stocker Memento.
    - ne fonctionne jamais ou examine le contenu d'un memento.

Exemple


```
class Créateur {//Originator
Object state;
Object save() {
return new Memento(state);
}
void restore(Object o) {
Memento m = (Memento)o;
this.state = m.state;
}

static class Memento {
Object state;
Memento(Object s) { state = s; }
}
}

class Gardien { //Caretaker
ArrayList<Object> states = new ArrayList<Object>();
void addMemento(Object o) { states.add(o); }
}
```

Pattern Memento

- **Conséquences**
  - Préserver l'encapsulation.
  - Simplifier l'Originator
  - Utiliser Memento peut être coûteux
  - Définition d'interfaces étroites et larges difficile dans certaines langages.
  - Des coûts cachés pour preserver Memento dans Caretaker.

Pattern Memento

- **Implémentation**
  - Prise en charge de deux interfaces large et étroite dans le langage de programmation
  - Stockage des changements progressifs (à la place de l'état entier)