



Analyse et Conception Objet de Logiciels

Chapitre 5

Patrons de conception



Patrons de conception

Objectifs

- Présenter quelques solutions conceptuelles (et logicielles) éprouvées et réutilisables en pratique
- Recueillir l'expérience et l'expertise des programmeurs

Intro

- Patrons de conception : introduits en 1977 par Christopher Alexander, architecte en bâtiment. Catalogue de problèmes classiques en construction (bâtiments, villes), avec des solutions.
- Patrons de conception orientés objet : reprennent cette idée de fournir un catalogue de solutions classiques à des problèmes de conception objet.



Notion de patron (« design pattern »)

- Description d'un problème récurrent, dans un certain contexte, accompagné d'une description des différents éléments d'une solution à ce problème.
- Solution suffisamment générale et flexible à un problème qu'on rencontre souvent.
- Éléments d'un patron de conception :
 - le nom (reconnaître le patron et indiquer son utilisation) ;
 - le problème, qui doit décrire l'objectif du patron ;
 - le contexte, qui décrit les circonstances d'utilisation du patron ;
 - la solution, qui décrit le schéma de conception résolvant le problème ;
 - les conséquences, qui décrivent les avantages et inconvénients de la solution proposée.



Notion de patron

Types de patrons

■ **Patrons d'analyse**

- Aider à la communication entre les utilisateurs et les informaticiens d'une part et à l'évaluation de la complexité d'autre part.

■ **Patrons de conception**

- Patrons de création, qui concernent la création d'objets ;
- Patrons « structurels », qui concernent la structure des objets et les relations entre ces objets ;
- Patrons comportementaux, relatifs au comportement des objets.

■ **Patrons de programmation (mise en œuvre)**

- Solutions spécifiques à un langage de programmation
(Ex : simulation de l'héritage multiple en Java).



Les patrons du GoF

- « Gang of Four » : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides
- Référence :
 - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (trad. Jean-Marie Lasvergères), *Design Patterns - Catalogue de modèles de conceptions réutilisables*, 1999.
- Les patrons de création : définissent comment faire l'instanciation et la configuration des classes et des objets.
- Les patrons structuraux : définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).
- Les patrons comportementaux : définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.



Les patrons du GoF

Création

- * **Fabrique abstraite** (Abstract Factory)
- * Monteur (Builder)
- * **Fabrique** (Factory Method)
- * Prototype (Prototype)
- * **Singleton** (Singleton)

Structure

- * **Adaptateur** (Adapter)
- * Pont (Bridge)
- * **Objet composite** (Composite)
- * **Décorateur** (Decorator)
- * Façade (Facade)
- * Poids-mouche (Flyweight)
- * Proxy (Proxy)

Comportement

- * Chaîne de responsabilité (Chain of responsibility)
- * **Commande** (Command)
- * **Interpréteur** (Interpreter)
- * Itérateur (Iterator)
- * Médiateur (Mediator)
- * Memento (Memento)
- * **Observateur** (Observer)
- * **État** (State)
- * **Stratégie** (Strategy)
- * Patron de méthode (Template Method)
- * **Visiteur** (Visitor)



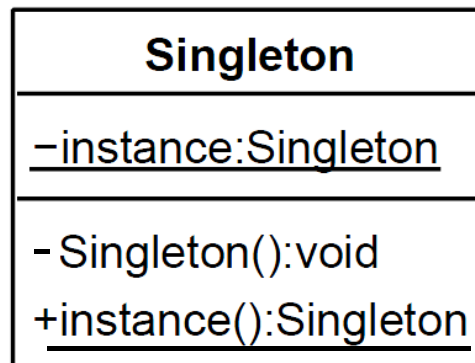
Les patrons du GoF (Singleton)

But

- L'objectif de ce patron est d'assurer qu'une classe a une instance unique, et d'en donner un accès global .

Motivation

- Lorsque la classe correspond à un objet unique dans le monde réel.





Les patrons du GoF (Singleton)

```
class Singleton {  
    // Attribut privé contenant l'instance unique de  
    Singleton.  
    final private static Singleton instance =  
        new Singleton() ;  
  
    // Méthode qui renvoie l'instance unique de Singleton.  
    static Singleton instance() {  
        return instance ;  
    }  
  
    /* On déclare le constructeur privé afin d'interdire  
    l'instanciation de cette classe depuis une autre  
    classe. */  
    private Singleton() { }  
}
```

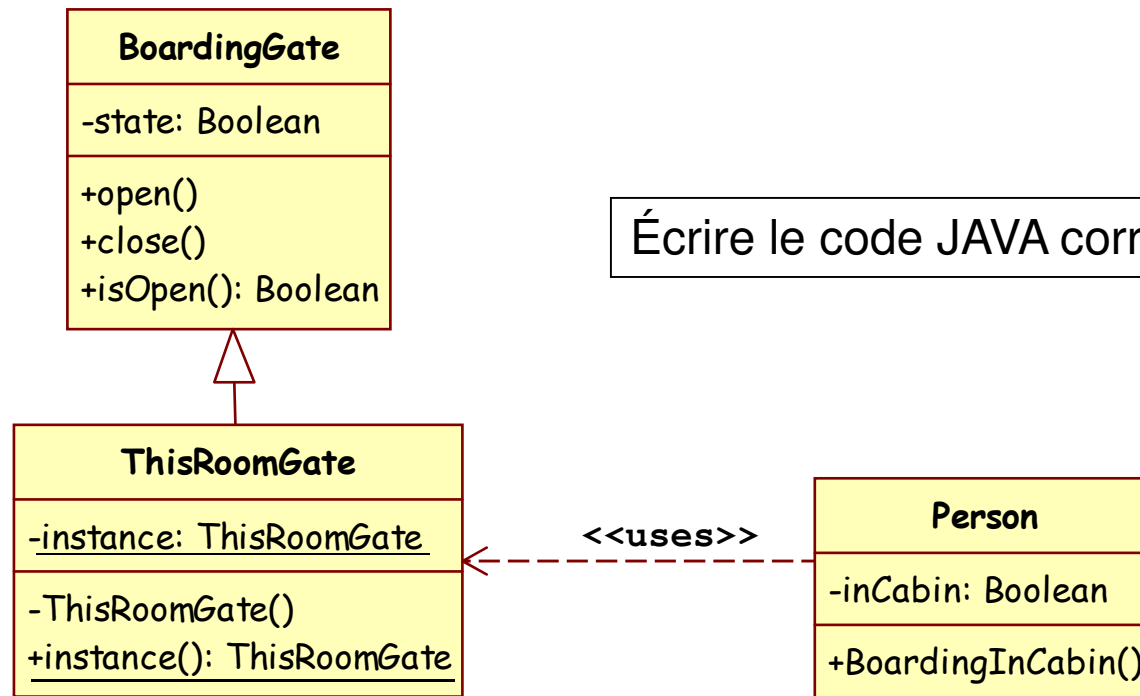

Les patrons du GoF (Singleton)

But

- L'objectif de ce patron est d'assurer qu'une classe a une instance unique, et d'en donner un accès global .

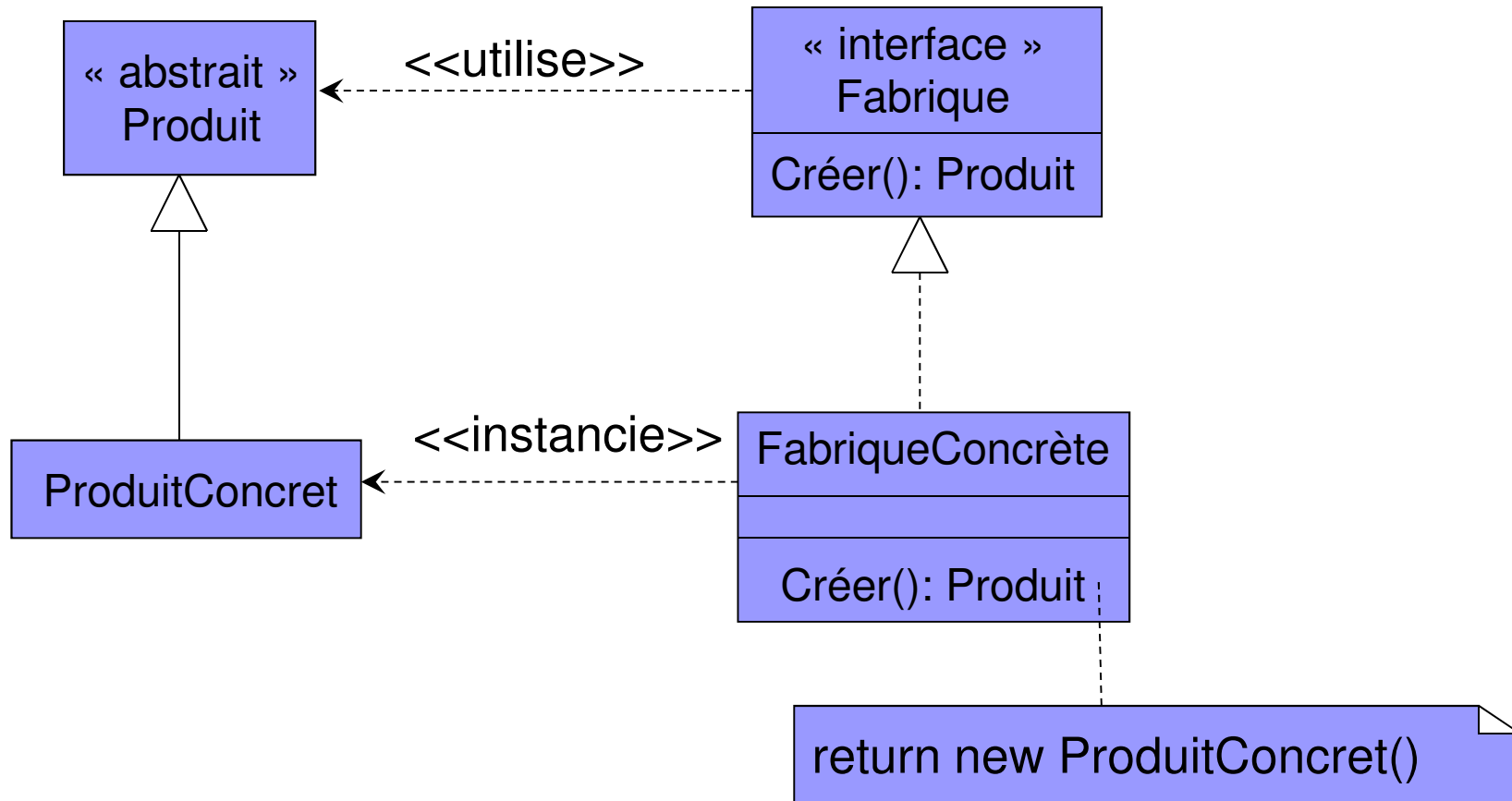
Application

- Une salle d'embarquement dispose d'une seule porte d'embarquement.



Écrire le code JAVA correspondant

Les patrons du GoF (Méthode Fabrique)



- Objectif : permettre la création d'objets sans que l'on sache la classe exacte de ces objets



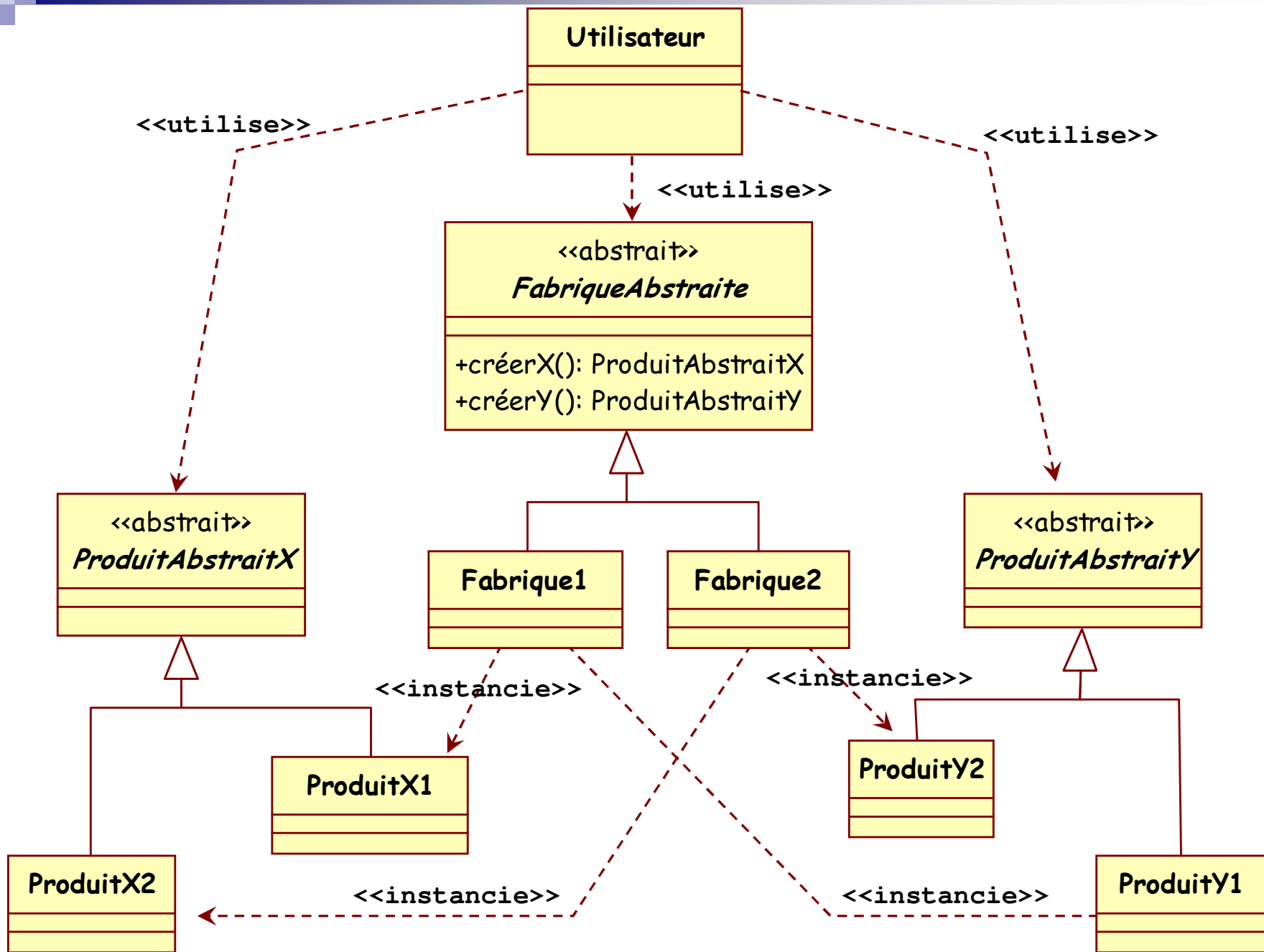
Les patrons du GoF (Fabrique abstraite)

But

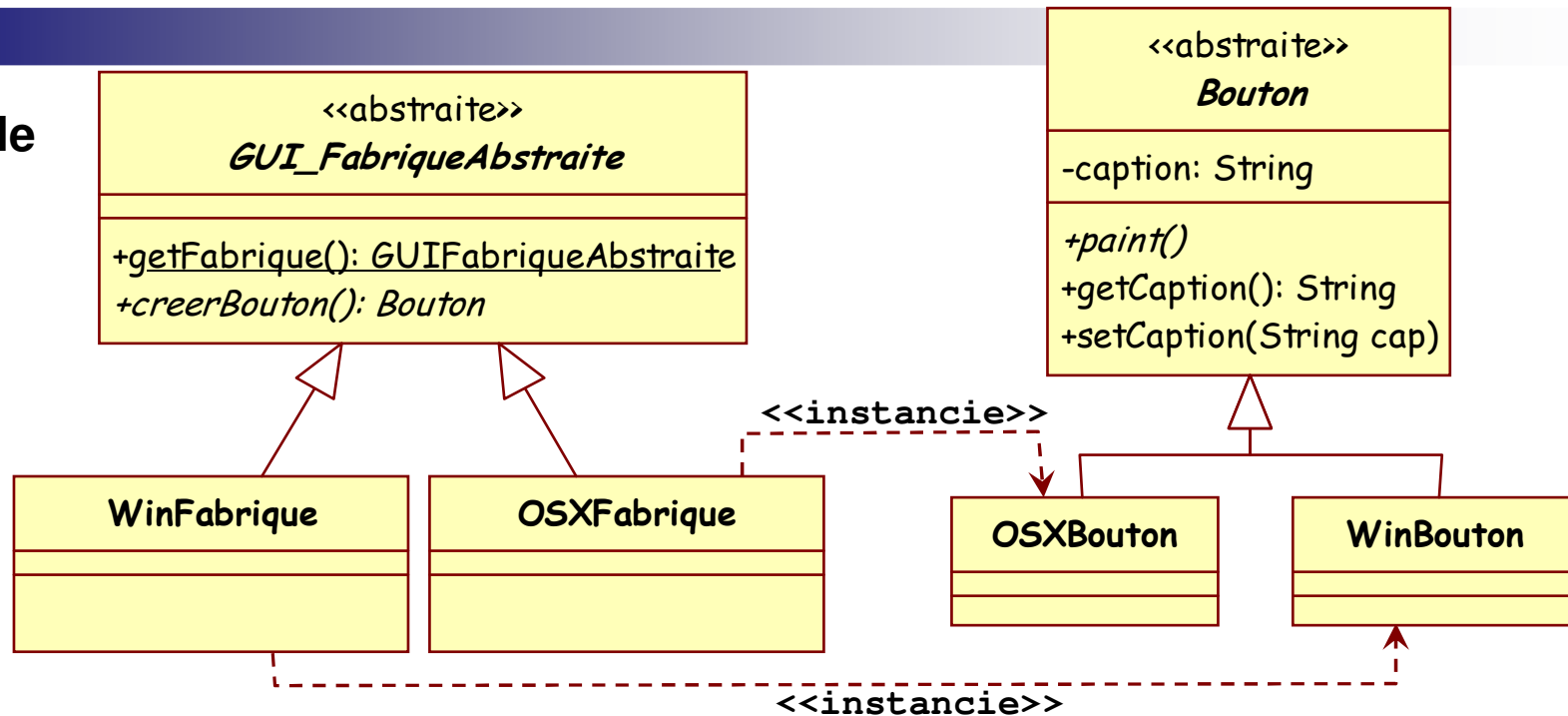
- Créer une famille d'objets qui dépendent les uns des autres, sans que l'utilisateur de cette famille d'objets ne connaisse la classe exacte de chaque objet.

Principe

- L'utilisateur a accès uniquement à différentes classes abstraites (une par produit), par exemple **ProduitAbstraitX** et **ProduitAbstraitY** et à une classe qui permet de créer des instances de ces produits : **Fabrique1** ou **Fabrique2**
- Ces instances sont créées à l'aide des méthodes **creerX** et **creerY**.
- Si l'utilisateur utilise la classe **Fabrique1** pour créer les objets, alors il obtient des instances de **ProduitX1** et **ProduitY1**. S'il utilise la classe **Fabrique2**, alors il obtient des instances de **ProduitX2** et **ProduitY2**.



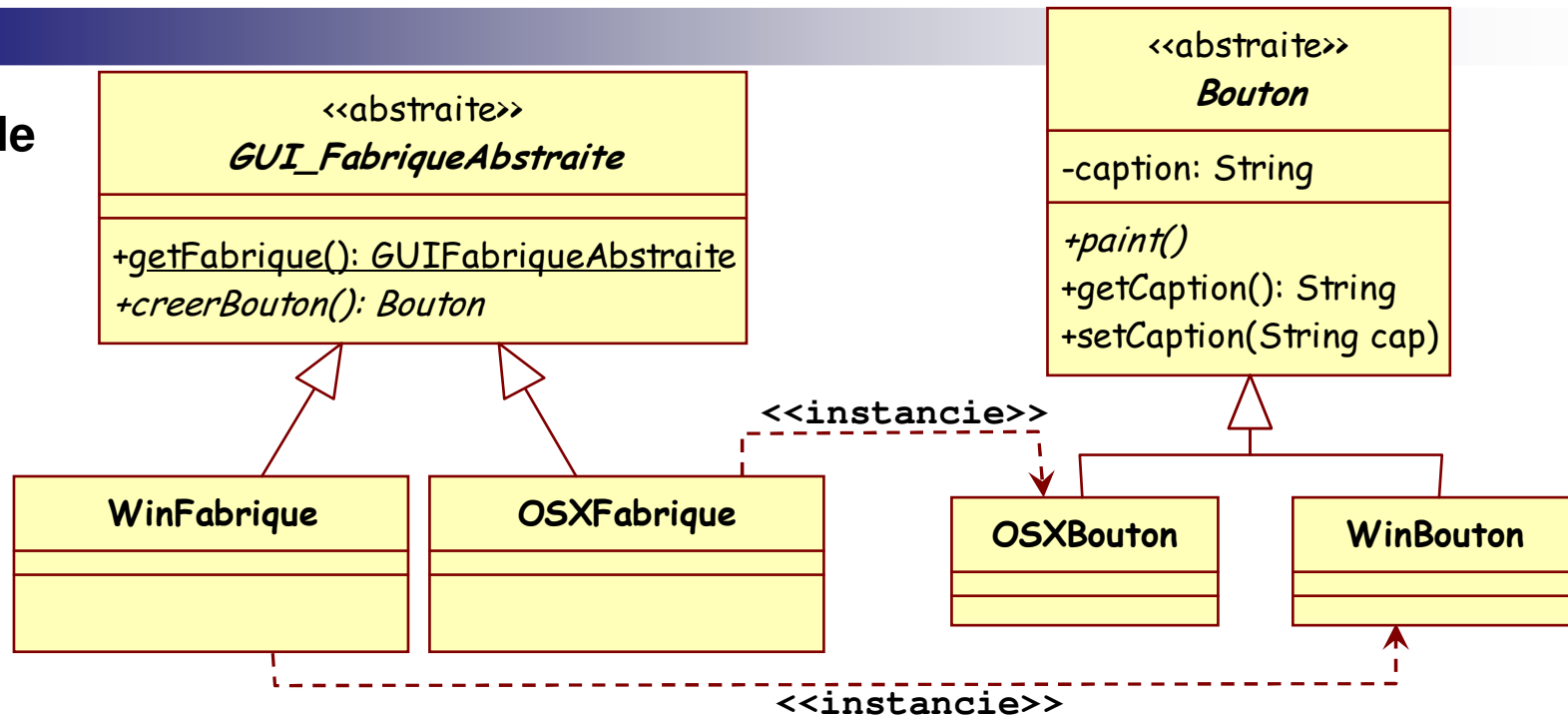
Exemple



```

abstract class GUI_FabriqueAbstraite {
    public static GUI_FabriqueAbstraite getFabrique() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys==0) { return(new WinFabrique()); }
        else { return(new OSXFabrique()); }
    }
    public abstract Bouton creerBouton();
}
  
```

Exemple

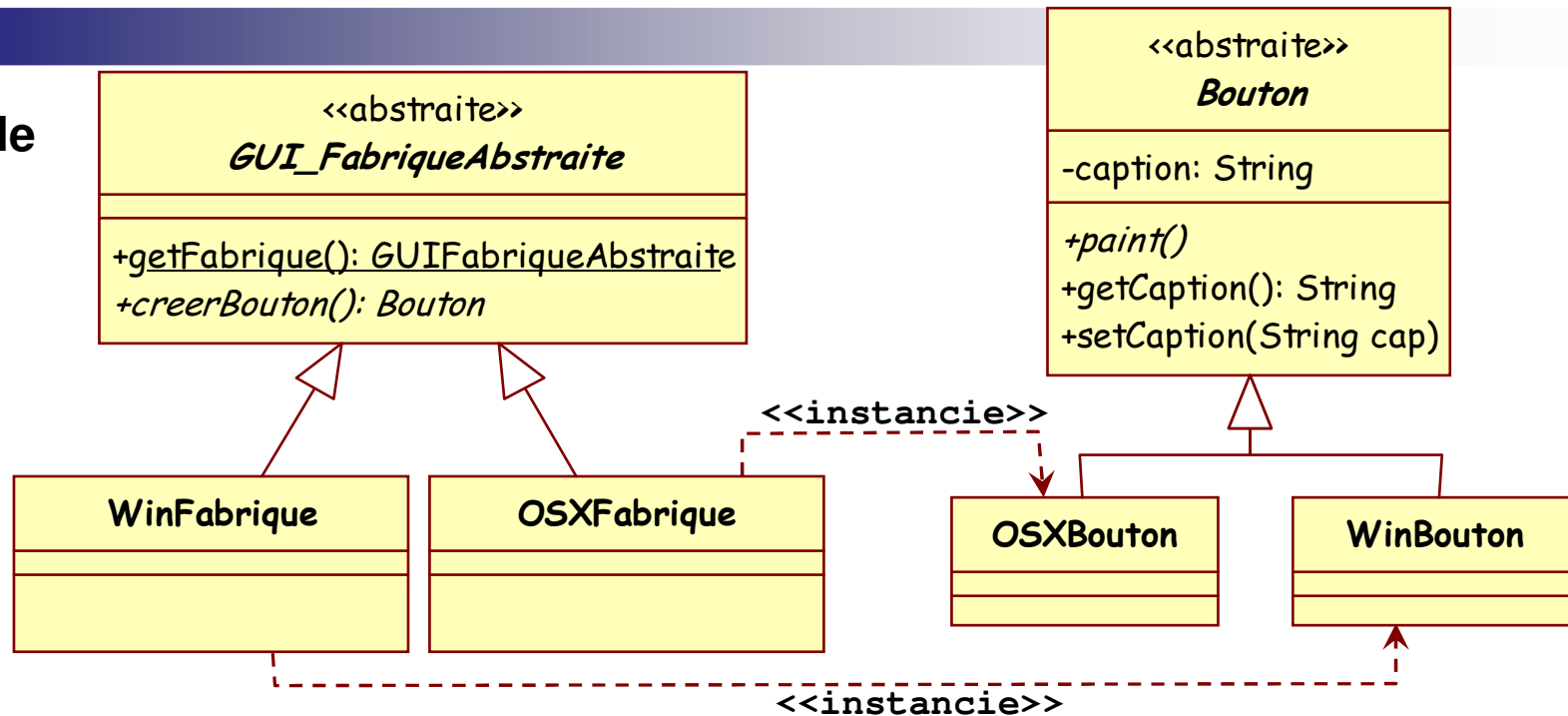


```

class WinFabrique extends GUI_FabriqueAbstraite {
    public Bouton creerBouton() {
        return(new WinBouton());
    }
}

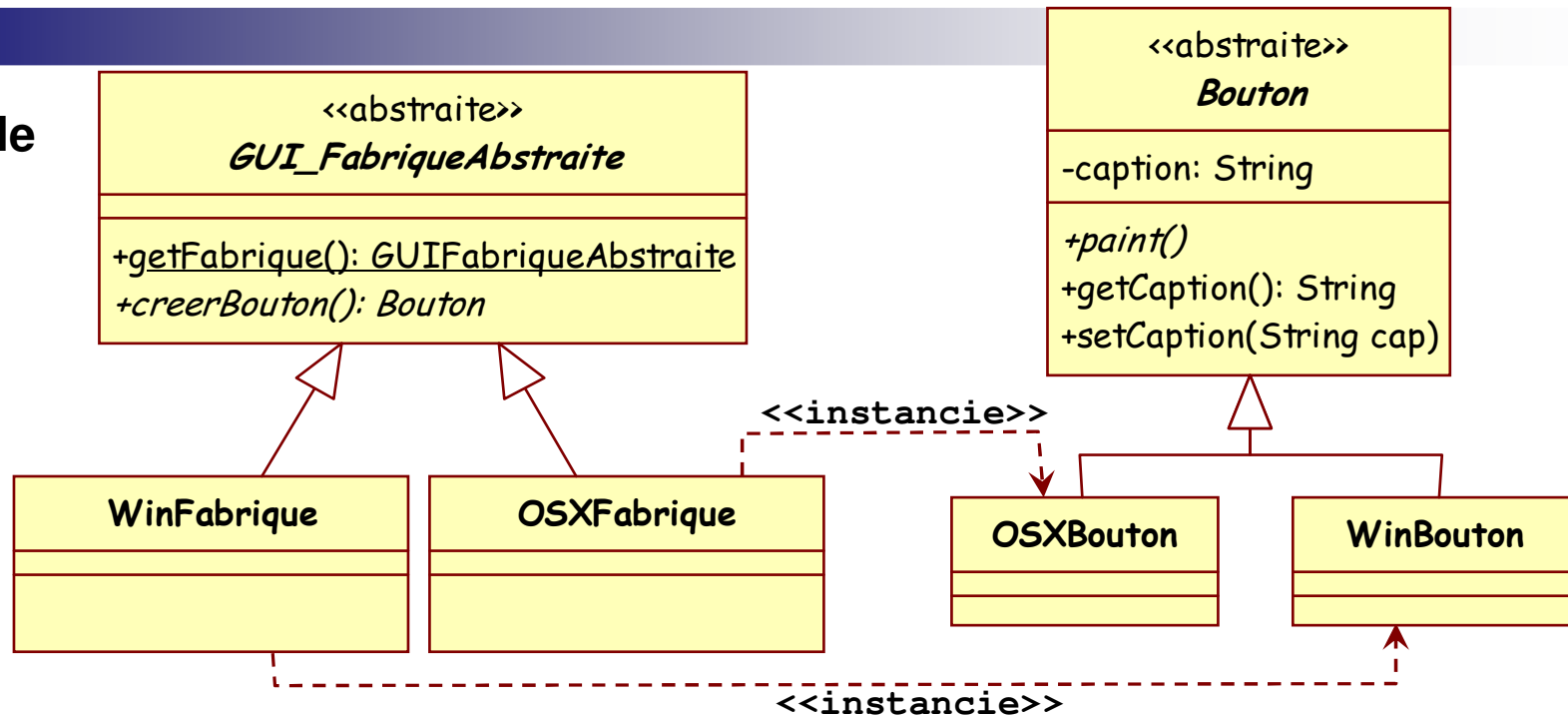
class OSXFabrique extends GUI_FabriqueAbstraite {
    public Bouton creerBouton() {
        return(new OSXBouton());
    }
}
  
```

Exemple



```
public abstract class Bouton {
    private String caption;
    public abstract void paint();
    public String getCaption() {
        return caption;
    }
    public String setCaption(String cap) {
        caption = cap;
    }
}
```

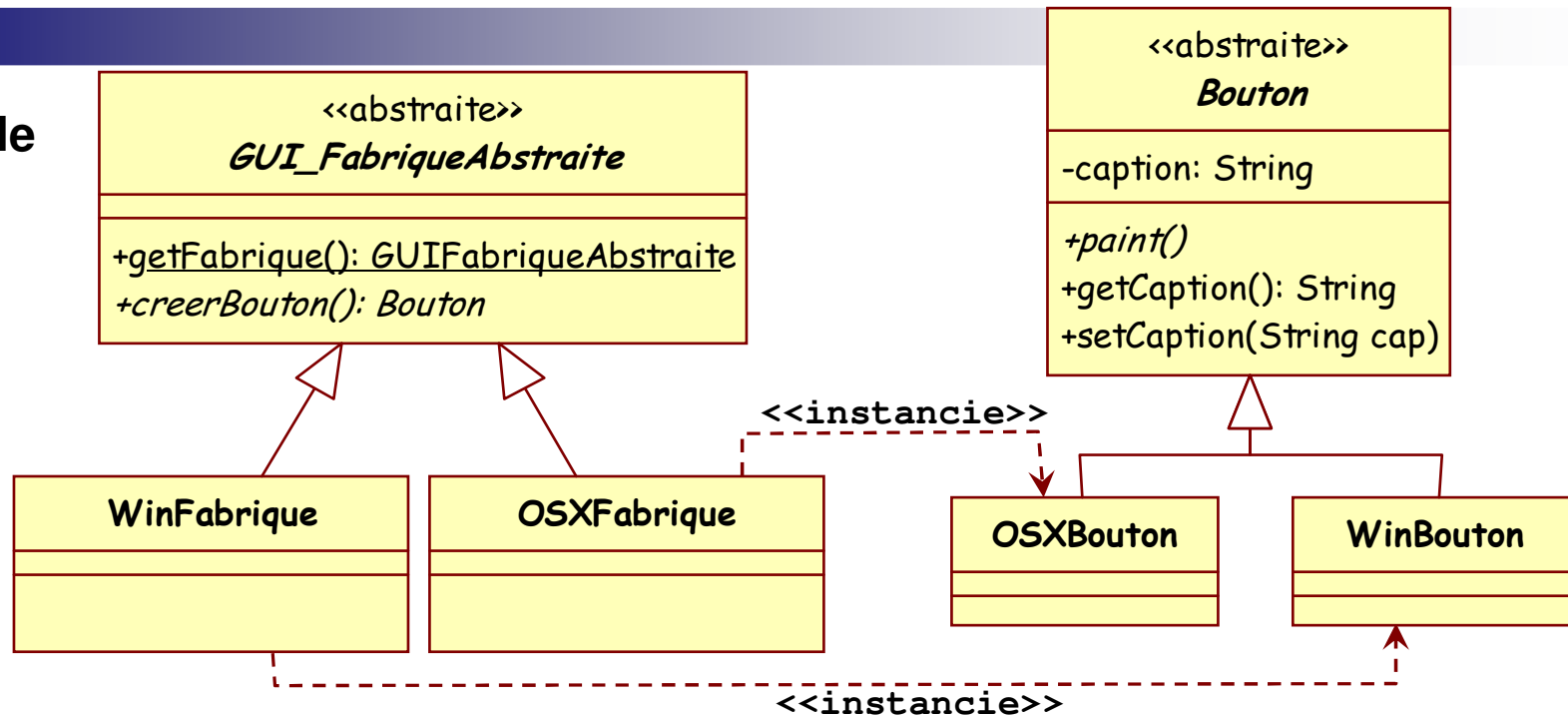
Exemple



```
class WinBouton extends Bouton {
    public void paint() {
        System.out.println("WinBouton: " + getCaption());
    }
}

class OSXBouton extends Bouton {
    public void paint() {
        System.out.println("OSXBouton: " + getCaption());
    }
}
```


Exemple



```

public class Application {
    public static void main(String[] args) {
        GUI_FabriqueAbstraite Fab = GUI_FabriqueAbstraite.getFabrique();
        Bouton bb = Fab.creerBouton();
        bb.setCaption("Play"); bb.paint();
    }
}
  
```

output :

WinBouton: Play

ou

OSXBouton: Play



Les patrons du GoF (Fabrique abstraite)

Avantages

- Facilite l'utilisation cohérente des différents produits : si le client utilise toujours la même classe, par exemple **Fabrique1**, pour créer des objets, il est sûr de toujours utiliser des produits de la même famille.
- L'utilisateur peut facilement changer de famille de produits, puisqu'il suffit de changer l'instanciation de la fabrique.
- On peut assez facilement ajouter une nouvelle famille de produits, en ajoutant une nouvelle sous-classe pour chaque produit abstrait.

Remarques

- On peut implémenter les sous-classes de **FabriqueAbstraite** en utilisant le patron **Singleton**.



Les patrons du GoF (Fabrique abstraite)

Exercice

On désire implémenter un moteur de jeu générique dans lequel deux éléments peuvent varier : les adversaires (on prévoit des obstacles **Puzzle** et **NastyVillain**) et les joueurs (on prévoit initialement deux types de joueurs **Kitty** et **KungFuGuy**). Cependant les divers jeux que l'on veut créer visent des publics différents : **Kitty** vs. **Puzzle** d'un côté, et **KungFuGuy** vs. **NastyVillain** de l'autre : pas question de faire jouer une instance de **Kitty** contre une de **NastyVillain**.

- Instancier le patron de la fabrique abstraite pour permettre la création et le lancement de différents jeux à partir du moteur générique.

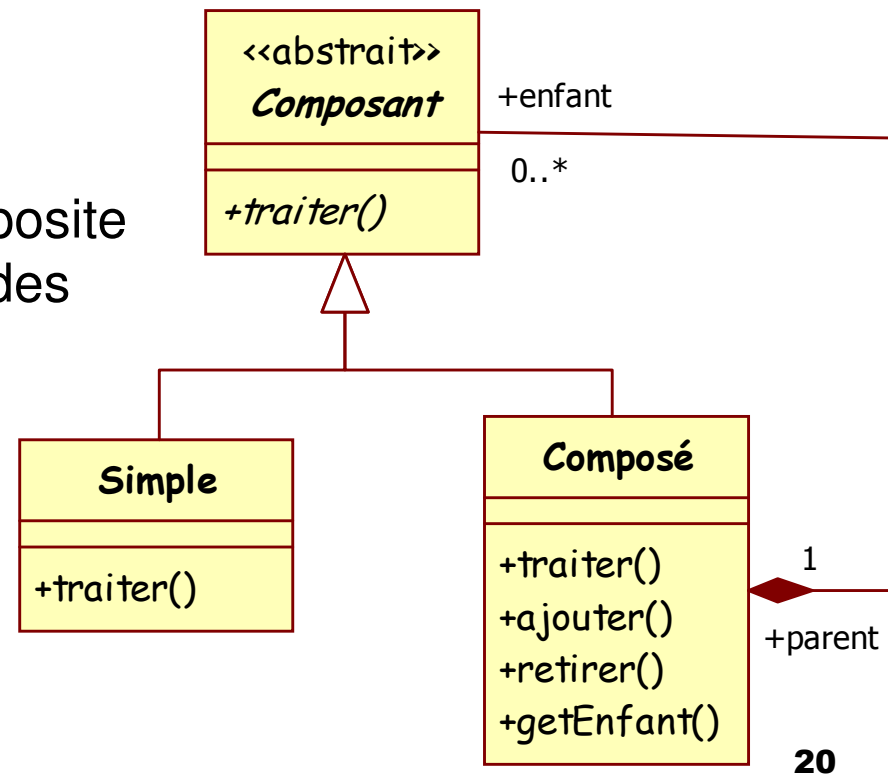
Les patrons du GoF (Objet composite)

But

- Créer des objets simples ou composés, avec des méthodes de traitement uniformes, pour lesquelles le client n'a pas à savoir s'il applique un certain traitement à un objet simple ou composé.

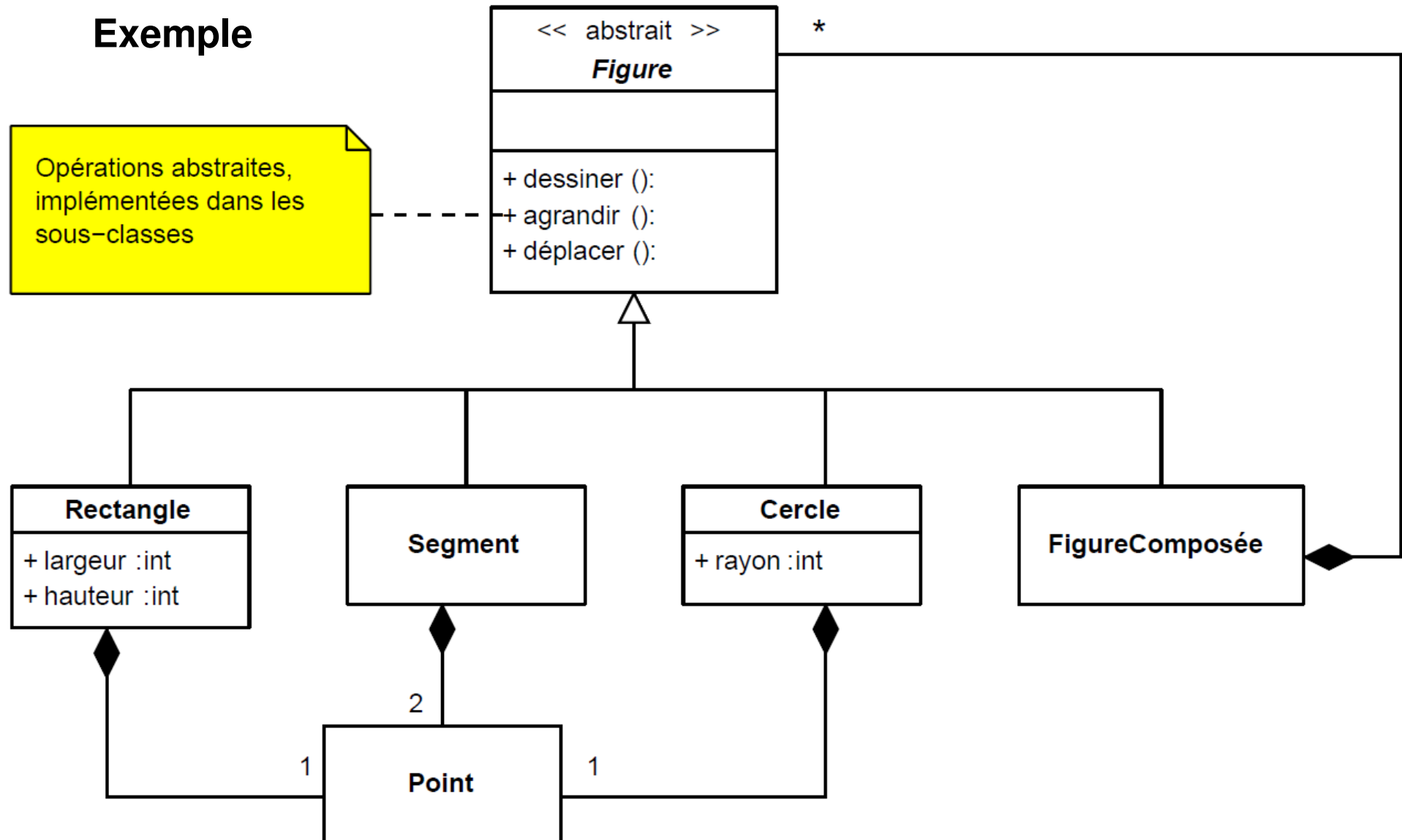
Solution

- On utilise une classe abstraite Composite contenant une (ou plusieurs) méthodes abstraites de traitement



Les patrons du GoF (Objet composite)

Exemple

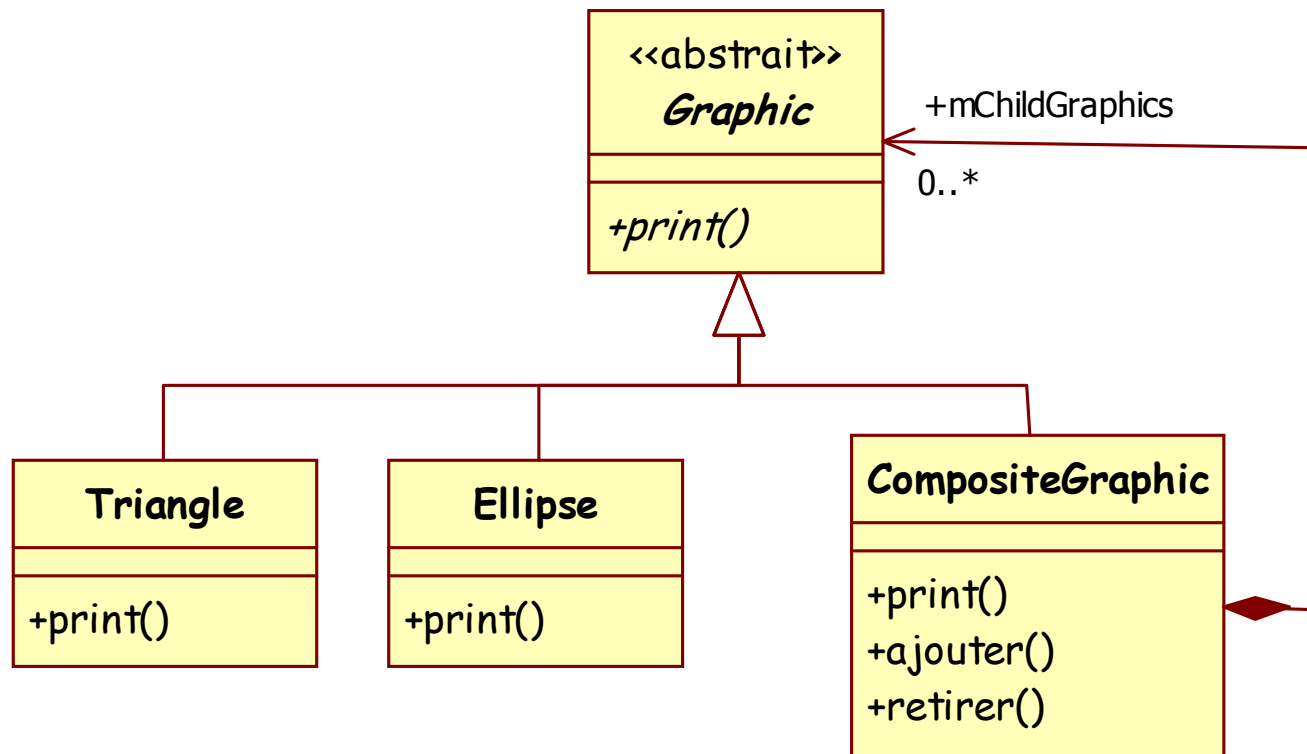


Les patrons du GoF (Objet composite)

Exercice

Proposer une implémentation en JAVA du patron composite.

L'appel de print() devrait afficher tous les éléments du graphiques





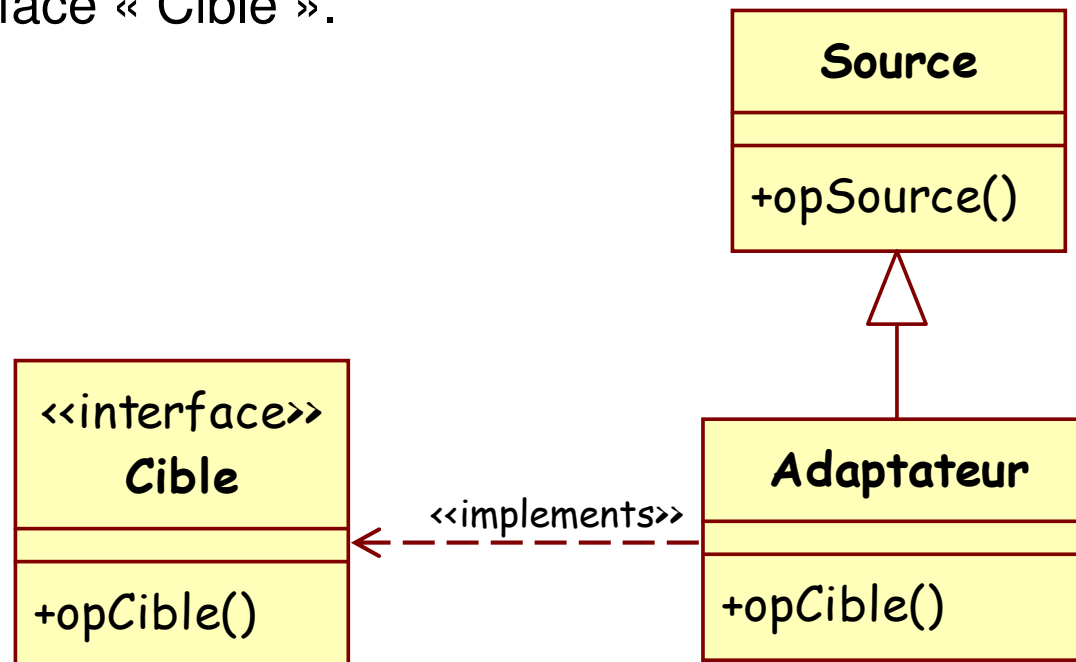
Les patrons du GoF (Adaptateur)

- L'adaptateur est un patron structurel.
- On utilise un adaptateur lorsque, pour implémenter une interface «cible», on souhaite réutiliser une classe «source» qui ne respecte pas exactement cette interface.
- La solution qui consiste à modifier la classe source n'est pas satisfaisante lorsque cette classe est réutilisée à d'autres endroits.
- **Motivation** : on veut intégrer une classe qu'on ne veut/peut pas modifier.

Les patrons du GoF (Adaptateur)

Solution par héritage

- Réaliser une sous-classe **Adaptateur** de « Source » qui implémente l'interface « Cible ».

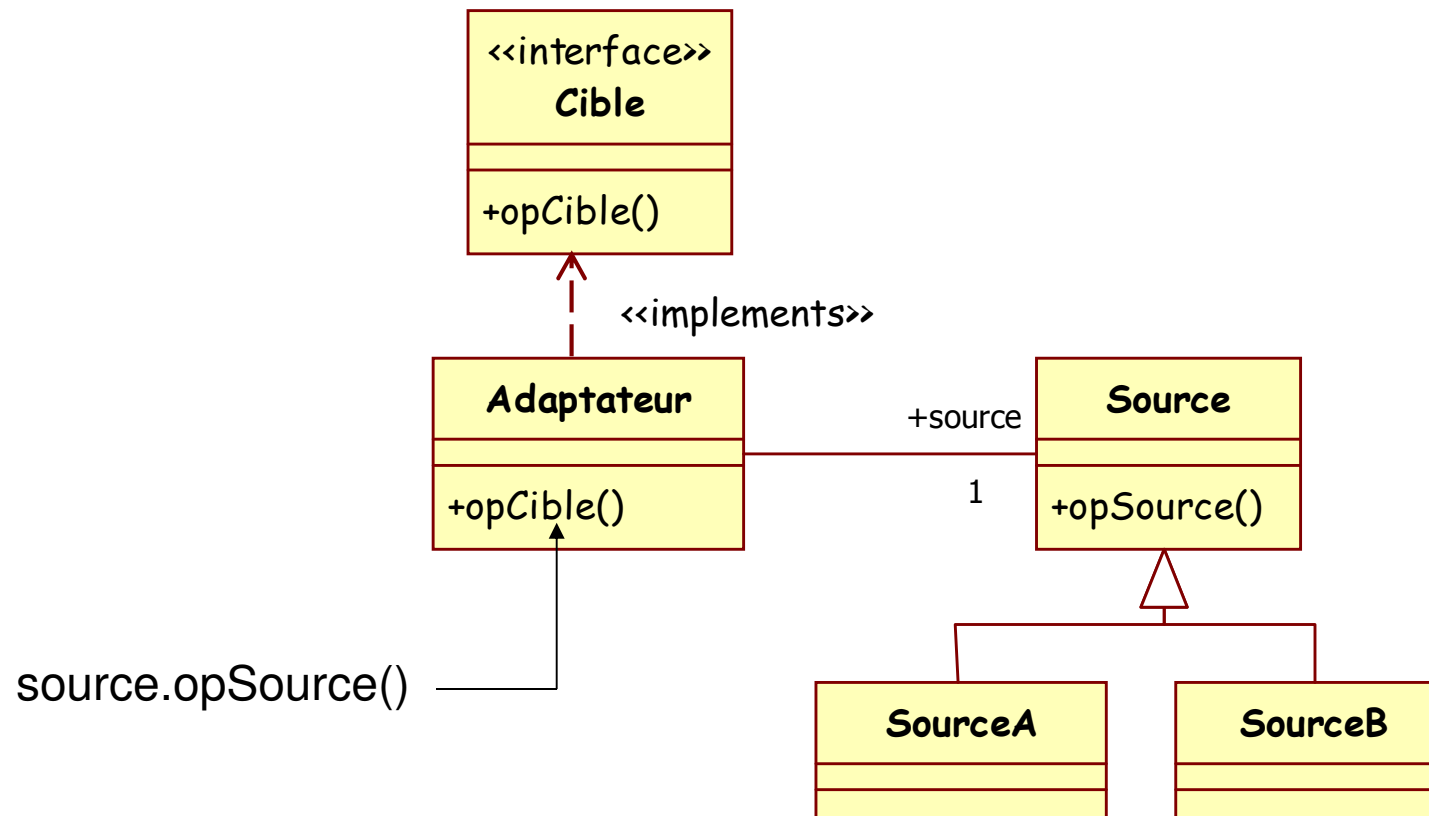


- Inconvénients :
 - on ne peut pas adapter des sous-classes de « Source » ;
 - on peut redéfinir des méthodes de « Source ».

Les patrons du GoF (Adaptateur)

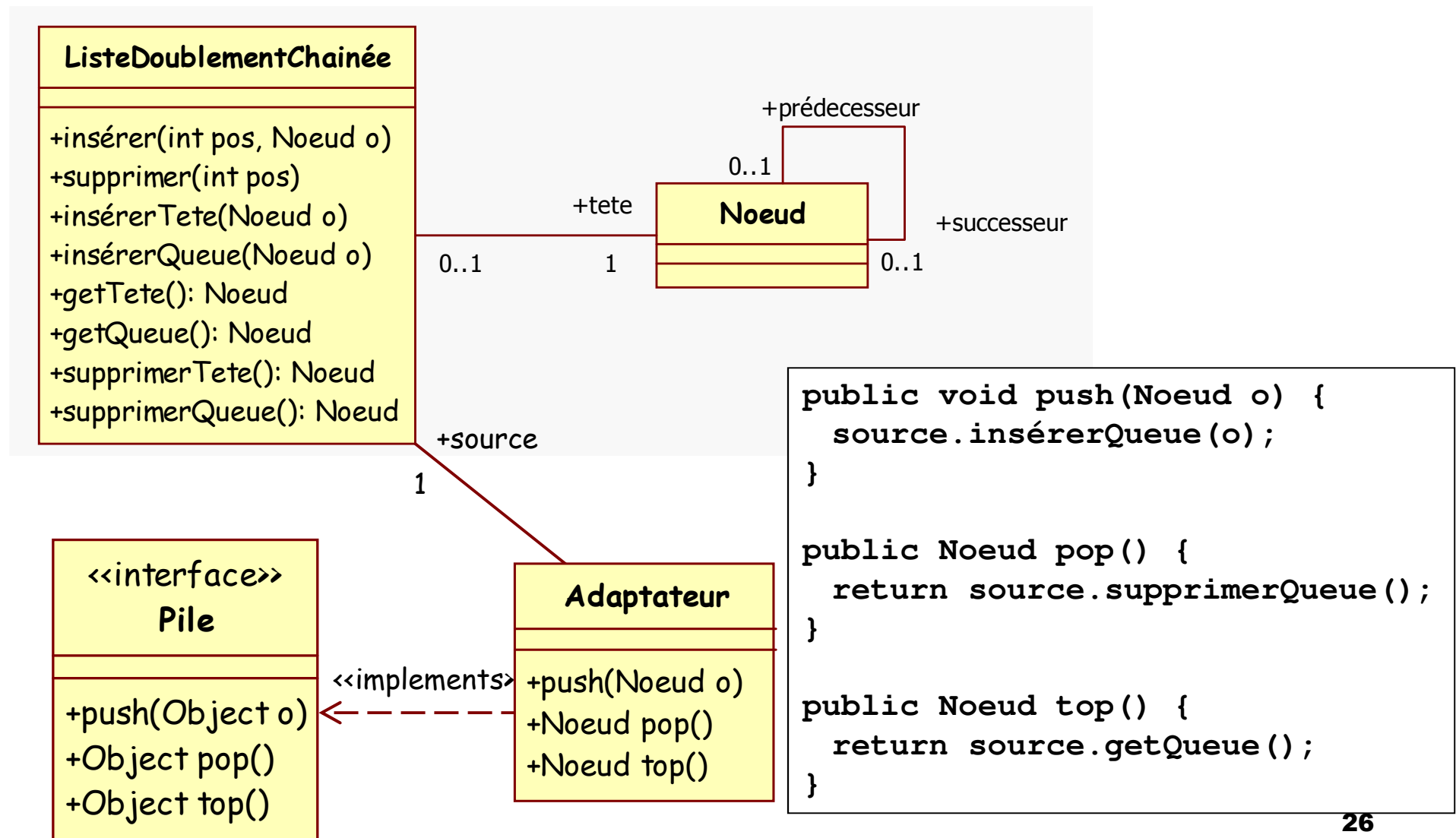
Solution par délégation

- Créer un lien entre l'adaptateur et la source
- l'adaptateur délègue le travail à effectuer à la classe source. La source joue le rôle du délégué.



Les patrons du GoF (Adaptateur)

Exemple (adaptateur par délégation)

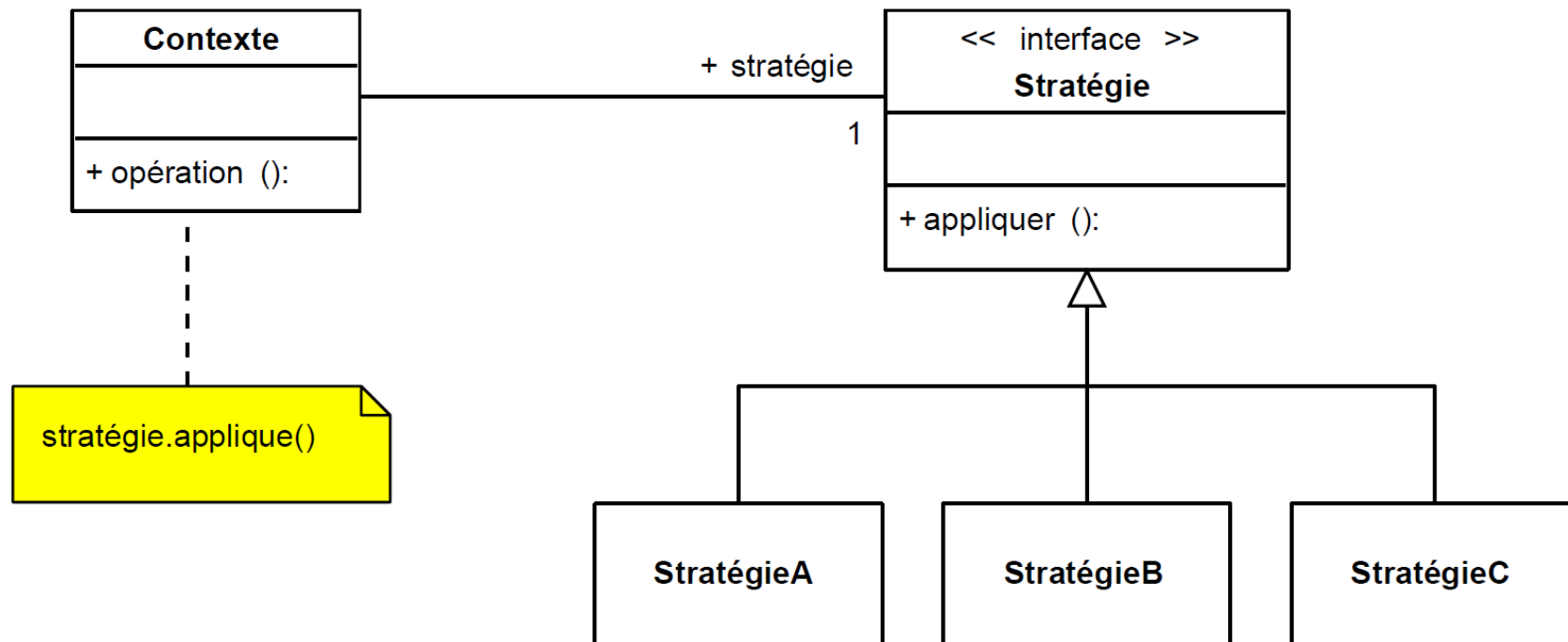


Les patrons du GoF (Stratégie)

- Définir une famille d'algorithmes encapsulés dans des objets, afin que ces algorithmes soient interchangeables dynamiquement.

Motivation

- Pour résoudre un problème, il existe souvent plusieurs algorithmes ; dans certains cas il peut être utile de choisir à l'exécution quel algorithme utiliser, par exemple selon des critères de temps de calcul ou de place mémoire.





Les patrons du GoF (Stratégie)

Avantages

- En découplant les algorithmes des données, la classe **Contexte** peut avoir des sous-classes indépendantes des stratégies ;
- On peut changer de stratégie dynamiquement.

Inconvénients

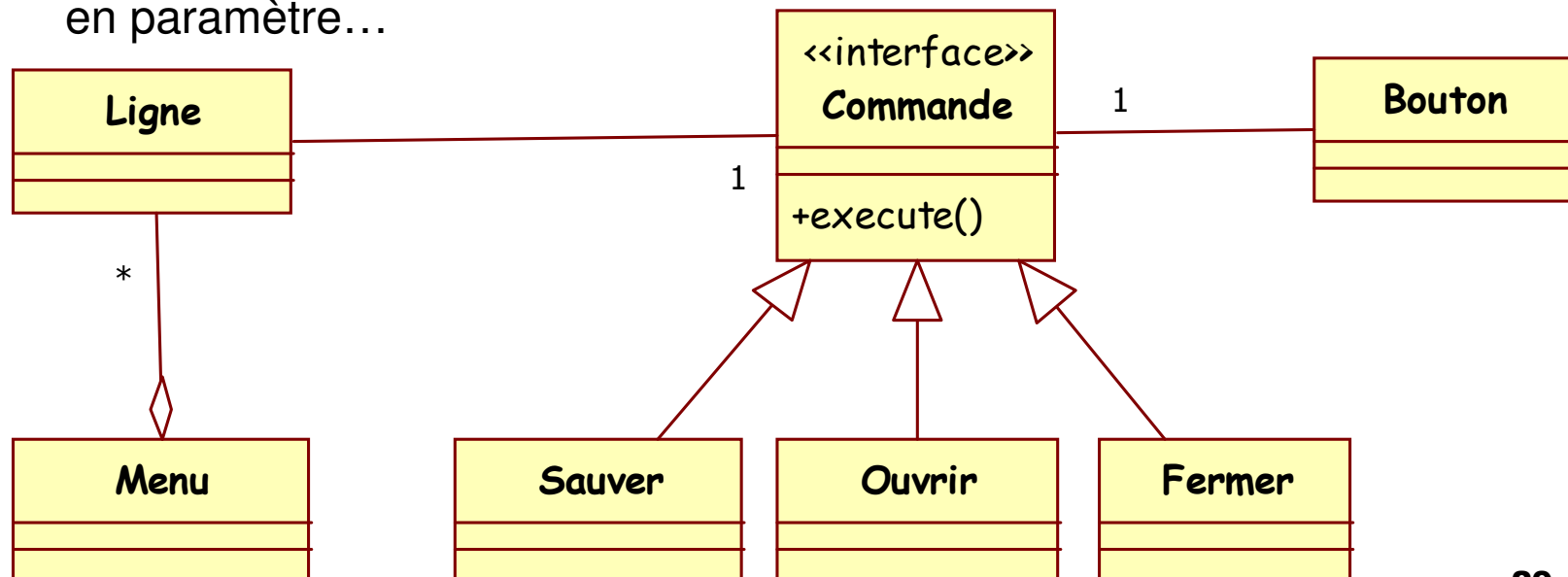
- Surcoût en place mémoire, car il faut créer des objets **Strategie** ;
- Surcoût en temps d'exécution à cause de l'indirection **strategie.appliquer()**

Les patrons du GoF (Commande)

Objectif

Encapsuler des commandes dans des objets interface graphique :

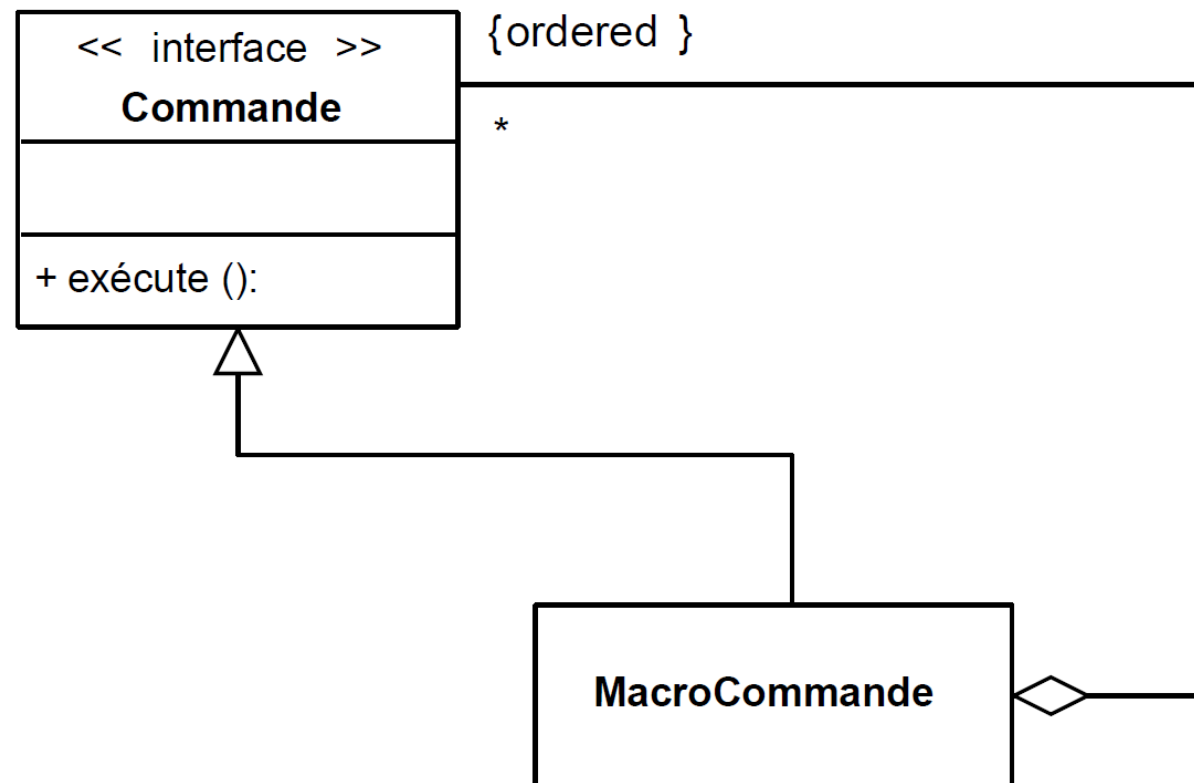
- ❑ associer des commandes à des boutons
- ❑ à des lignes dans des menus ;
- ❑ revenir en arrière d'une ou plusieurs commandes (annuler) ou repartir en avant (rétablir).
- ❑ on doit associer à des actions des objets que l'on peut stocker, passer en paramètre...



Les patrons du GoF (Commande)

Avantages

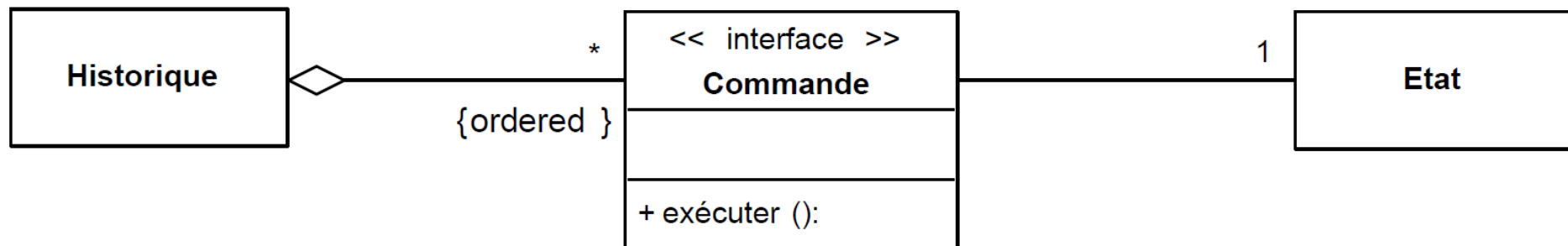
- on peut modifier une association Ligne--Commande à l'exécution, afin de paramétrer le logiciel à l'exécution ;
- on peut effectuer la même action par différents moyens (en passant par un menu, ou en appuyant sur un bouton... etc.) ;
- on peut définir des macro commandes composées de séquences de commandes



Les patrons du GoF (Commande)

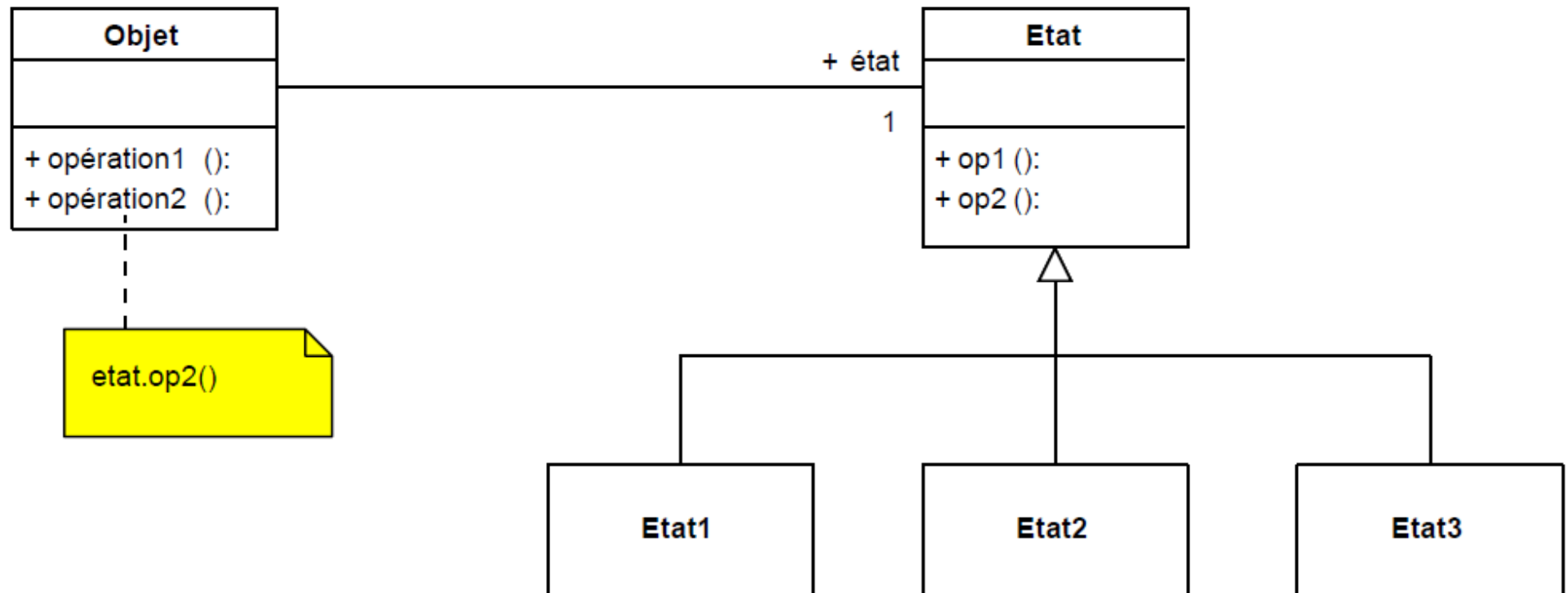
Avantages

- on peut revenir en arrière (annuler, rétablir). Cela nécessite de stocker l'historique des commandes, et un état interne associé à chaque commande effectuée



Patron Etat

- Permet de réaliser des objets dont le comportement change lorsque leur état interne est modifié.



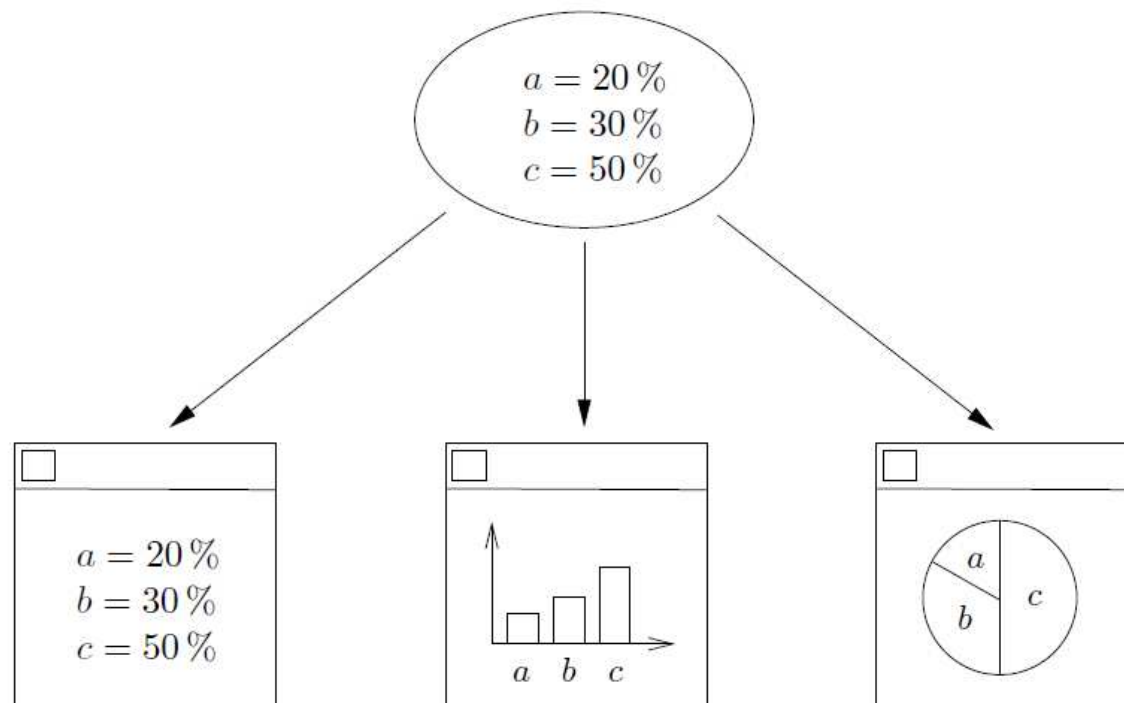


Patrons Stratégie/Commande/Etat

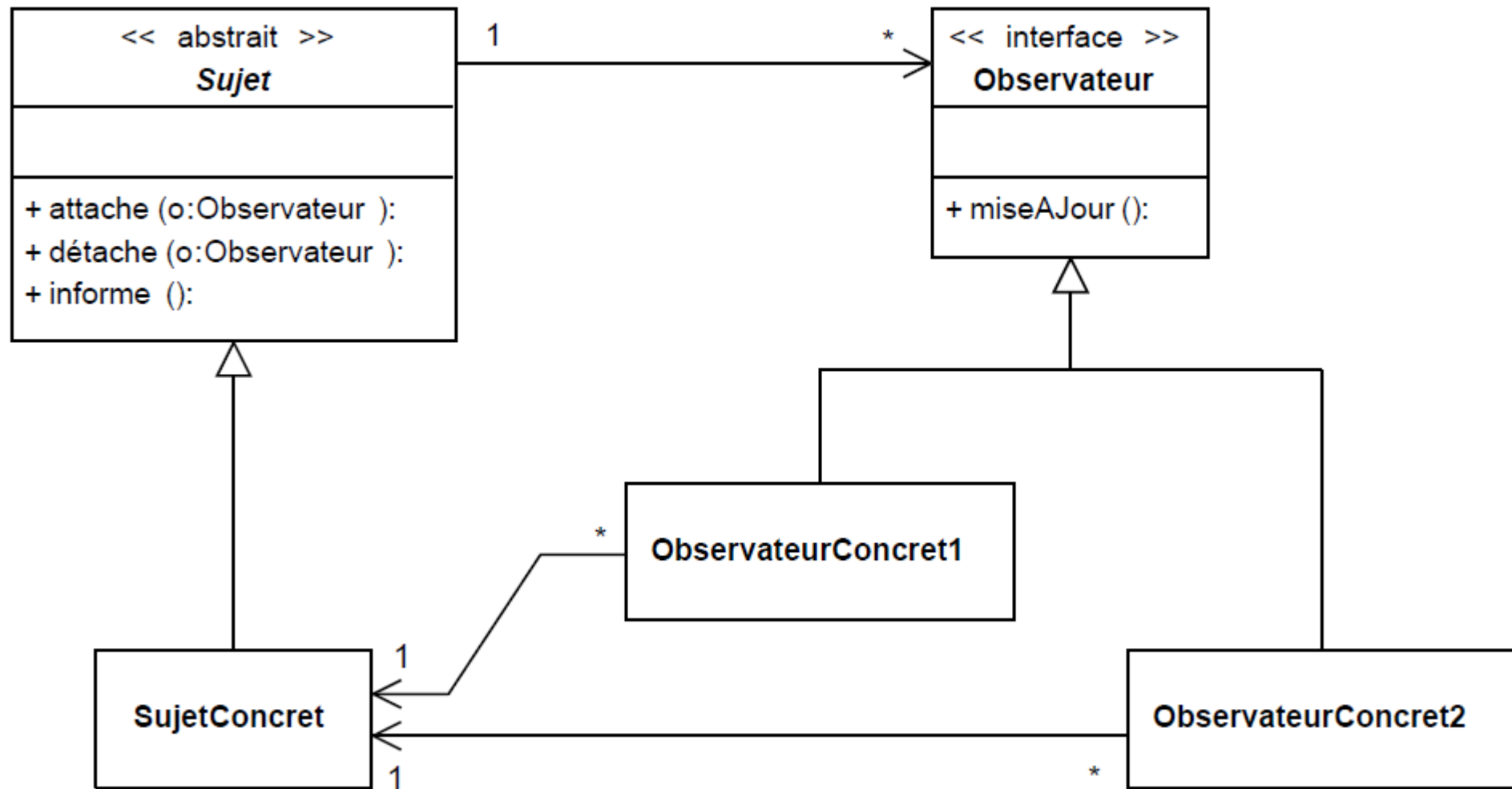
- Stratégie : le but est d'implémenter une même opération de plusieurs façons différentes. On peut considérer qu'on a une seule spécification de l'opération.
- Commande : on peut vouloir associer une même commande à différents objets, et définir des séquences de commandes (macros) ou des historiques de commandes.
- Etat : plusieurs opérations peuvent être associées à un état.

Patron Observateur

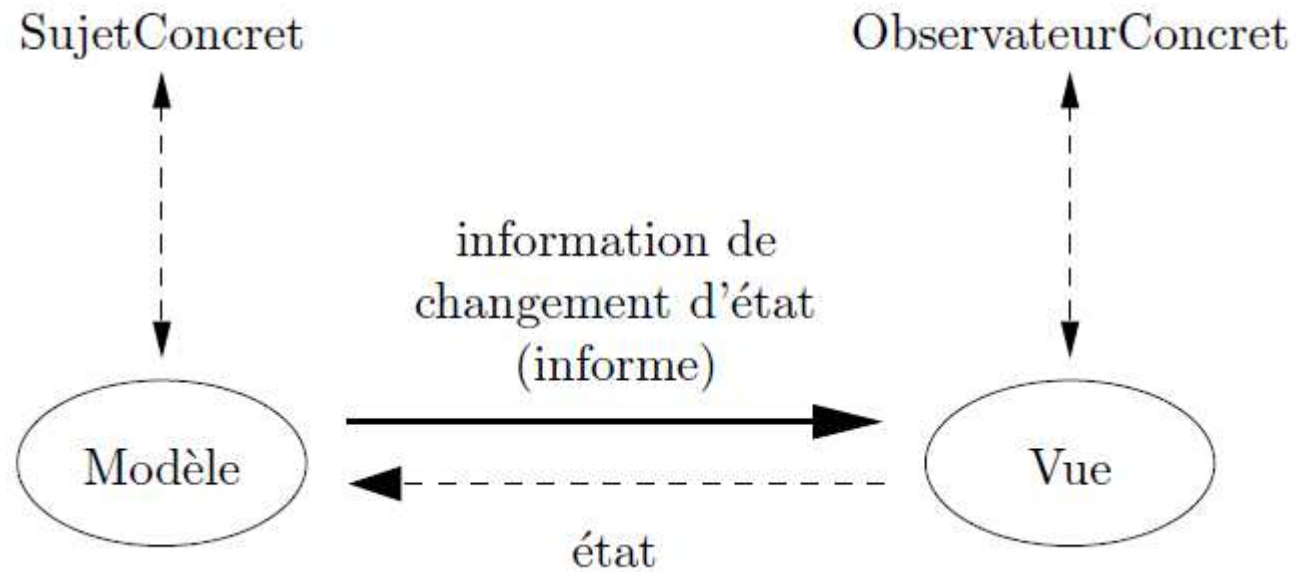
- L'Observateur est un patron comportemental
- Permet de définir une dépendance entre un objet (appelé sujet) et un ensemble d'objets (appelés observateurs)
- Lorsque le sujet change d'état, tous les observateurs qui en dépendent soient informés et mis à jour automatiquement.



Patron observateur



Patron observateur





Patron interprète

- Patron de conception comportemental.

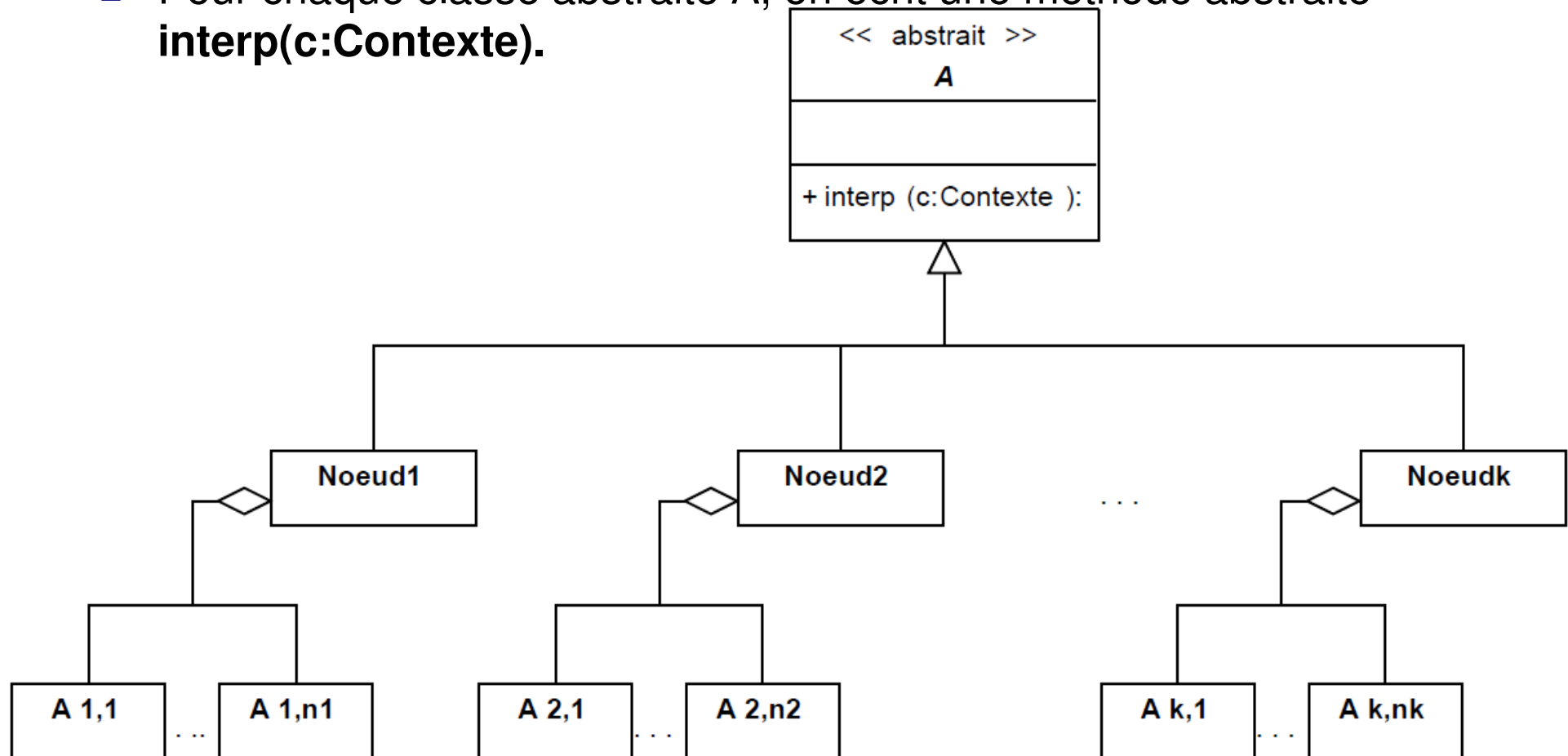
Objectif :

- Etant donné un langage, il permet de définir une représentation pour sa grammaire abstraite (autrement dit, une structure d'arbre abstrait), ainsi qu'un interprète utilisant cette représentation.
- On suppose qu'on a une grammaire abstraite, avec des règles de la forme suivante :

$$\begin{array}{lcl} A & \rightarrow & \text{Noeud}_1(A_{1,1}, \dots, A_{1,n_1}) \\ & & | \text{Noeud}_2(A_{2,1}, \dots, A_{2,n_2}) \\ & & | \dots \\ & & | \text{Noeud}_k(A_{k,1}, \dots, A_{k,n_k}) \end{array}$$

Patron interprète

- On associe à chaque non terminal une classe abstraite, et à chaque noeud une classe concrète qui hérite de cette classe abstraite.
- Pour chaque classe abstraite A , on écrit une méthode abstraite **interp(c:Contexte)**.





Patron interprète

Avantages :

- On peut facilement modifier et étendre la grammaire. Par exemple, on peut facilement ajouter de nouvelles expressions en définissant de nouvelles classes.
- L'implémentation de la grammaire est simple, et peut être réalisée automatiquement à l'aide d'outils de génération.

Inconvénients :

- lorsque la grammaire est complexe, on a une multiplication des classes.
- il devient alors délicat d'ajouter de nouvelles opérations, car celles-ci doivent être ajoutées dans toutes les classes de la hiérarchie.



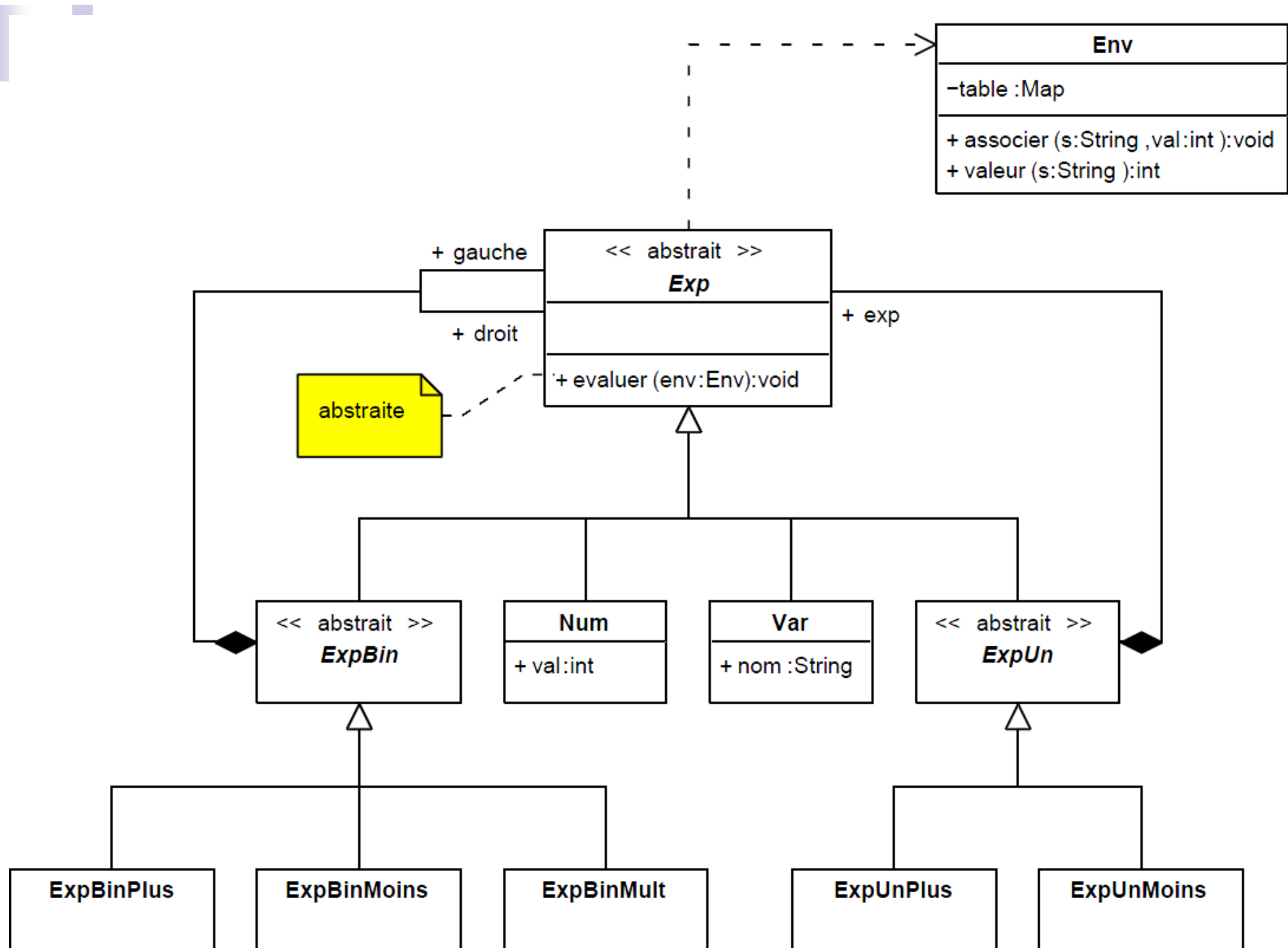
Exercice 21. Patron Interprète


$\text{Exp} \rightarrow \text{ExpBin} \mid \text{ExpUn} \mid \underline{\text{num}} \mid \underline{\text{var}}$

$\text{ExpBin} \rightarrow \text{Exp} + \text{Exp} \mid \text{Exp} - \text{Exp} \mid \text{Exp} * \text{Exp}$


$\text{ExpUn} \rightarrow + \text{Exp} \mid - \text{Exp}$

2 * - a






```
class Env {  
    private Map<String, Integer> table ;  
    public void associe(String s, int i) {  
        table.put(s, i) ;  
    }  
    public int valeur(String s) {  
        if (table.containsKey(s)) {  
            return table.get(s) ;  
        } else {  
            throw new ValeurIndefinie() ;  
        }  
    }  
}
```



```
abstract class Exp {  
    abstract public int eval(Env env) ;  
}
```

```
abstract class ExpBin extends Exp {  
    Exp gauche ;  
    Exp droit ;  
    public ExpBin(Exp g, Exp d) {  
        gauche = g ;  
        droit = d ;  
    }  
}
```

```
class ExpBinPlus extends ExpBin {  
    public ExpBinPlus(Exp g, Exp d) {  
        super(g, d) ;  
    }  
    public int eval(Env env) {  
        return gauche.eval(env) + droit.eval(env) ;  
    }  
}
```



```
class Num extends Exp {  
    int val ;  
    public Num(int val) {  
        this.val = val ;  
    }  
    public int eval(Env env) {  
        return val ;  
    }  
}
```

```
class Var extends Exp {  
    String nom ;  
    public Var(String nom) {  
        this.nom = nom ;  
    }  
    public int eval(Env env) {  
        return env.valeur(nom) ;  
    }  
}
```



Test

```
class TestExp {  
    static void ok(boolean b) {  
        if (b) {  
            System.out.print("ok ") ;  
        } else {  
            System.out.print("erreur ") ;  
        }  
    }  
  
    public static void main(String[] args) {  
        Env env = new Env() ;  
        env.associe("a", 1) ;  
        env.associe("b", 2) ;  
        Exp exp = new ExpBinMult(new Num(2), new ExpUnMoins(new Var("a"))) ;  
        ok(exp.eval(env) == -2) ;  
        System.out.println("") ;  
    }  
}
```

2	*	-	a
---	---	---	---



Patron visiteur

- Patron de conception comportemental.

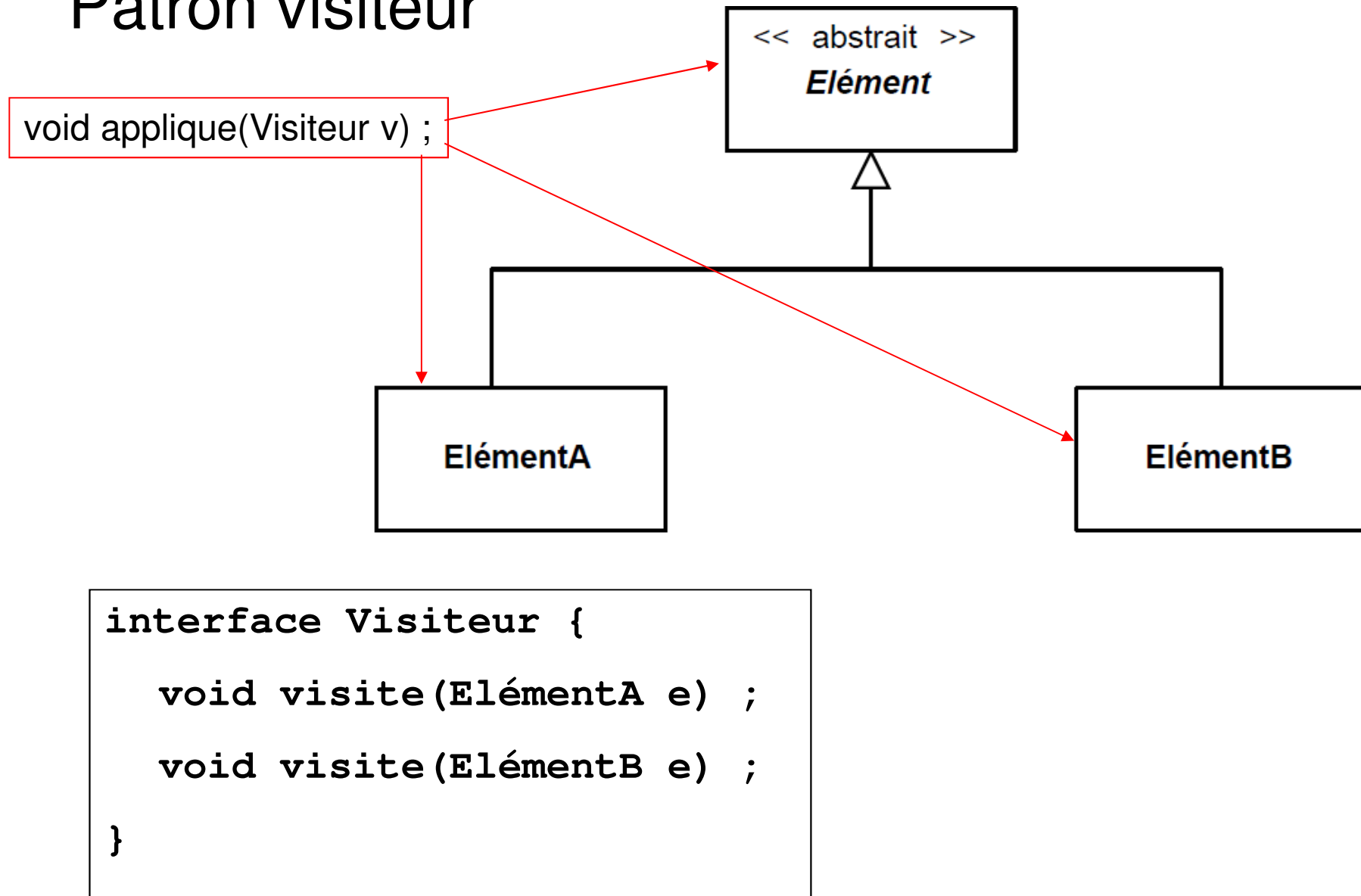
Objectif :

- représenter une opération définie en fonction de la structure d'un objet. Le Visiteur permet alors de définir de nouvelles opérations sans modifier les classes qui définissent la structure des objets auxquelles elles s'appliquent.

Principe

- On considère une hiérarchie de classes définissant la structure d'objets. On suppose qu'on a une classe abstraite **Elément**, racine de cette hiérarchie
- On définit une interface **Visiteur**, qui contient une méthode **void visite(ElémentX e)** pour chaque sous-classe concrète **ElémentX** de **Elément**.

Patron visiteur






```
abstract class Élément {  
    abstract void applique(Visiteur v) ;  
}
```

```
class ÉlémentA extends Élément {  
    void applique(Visiteur v) {  
        v.visite(this) ; // Appel de la méthode visite(ElémentA)  
    }  
}
```

```
class ÉlémentB extends Élément {  
    void applique(Visiteur v) {  
        v.visite(this) ; // Appel de la méthode visite(ElémentB)  
    }  
}
```


- 
- On appelle « visiteur » un objet d'une classe qui implémente l'interface **Visiteur**.
 - Un visiteur définit une opération s'appliquant sur tout objet de type **Elément**.
 - La méthode **applique(Visiteur v)** représente l'application de l'opération définie par le visiteur à l'objet.

```
class Opération implements Visiteur {  
    public void visite(ElémentA e) {  
        // Ce que l'opération doit faire sur un objet de type ElémentA  
    }  
    public void visite(ElémentB e) {  
        // Ce que l'opération doit faire sur un objet de type ElémentB  
    }  
    // Les attributs de cette classe peuvent servir de paramètres  
    // d'entrée ou de résultat pour l'opération.  
}
```



Patron Visiteur

Un appel de l'opération se fait de la façon suivante :

```
Elément e = new ElémentA() ; // Un élément  
Visiteur v = new Opération() ; // Une opération  
e.applique(v) ; // L'opération v est appliquée sur l'élément e
```



Ce qu'on fait dans une programmation non objet

```
class Opération {  
    static void op(Elément e) {  
        if (e instanceof ElémentA) {  
            ElémentA aA = (ElémentA) e ;  
            // Ce que l'opération doit faire sur ElémentA  
            ...  
        } else if (e instanceof ElémentB) {  
            ElémentB eB = (ElémentB) e ;  
            // Ce que l'opération doit faire sur ElémentB  
            ...  
        }  
    }  
}
```



Patron visiteur

Avantages

- Il oblige à traiter tous les cas, et c'est vérifié à la compilation ;
- il s'agit d'une programmation « purement objet » ;
- le patron évite l'utilisation de **instanceof**, de devoir déclarer une variable initialisée à l'aide d'une conversion, et évite également ainsi le risque d'erreur de conversion à l'exécution ;
- il évite d'effectuer plusieurs tests pour trouver le code à exécuter (en particulier lorsque **Elément** a de nombreuses sous-classes) ;
- il permet l'ajout de nouvelles opérations sans modifier la hiérarchie de classes, et l'ajout de nouvelles classes (en modifiant l'interface).



Patron visiteur

Inconvénients

- Ce patron est lourd à mettre en oeuvre : il faut prévoir une méthode **applique** par classe ;
- le traitement de méthodes comportant des paramètres et un résultat est également lourd ;
- on a un surcoût à chaque indirection **applique** → **visite** ;
- le code est peu lisible lorsqu'on ne connaît pas le patron.



Variante générique du patron Visiteur

- Interface **Visiteur** générique, paramétré par le type de retour **T** de la méthode **visite**

```
interface Visiteur<T> {  
    T visite(ElémentA e) ;  
    T visite(ElémentB e) ;  
}
```

- 
- Méthodes `applique` génériques, paramétrée par le type de retour `T`

```
abstract class Element {  
    abstract <T> T applique(Visiteur<T> v) ;  
}
```

```
class ElementA extends Element {  
    <T> T applique(Visiteur<T> v) {  
        return v.visite(this) ;  
    }  
}
```



Avantages de cette variante générique

- Permet d'avoir des types de retour différents suivant les visiteurs utilisés
- Evite d'utiliser un attribut supplémentaire pour coder le type de retour d'une méthode **visite**.