

Méthodes pour l'informatisation - compléments

Conception et architecture

Christine Plumejeaud

Doctorante Informatique UJF

CNAM - centre de Grenoble

Je remercie [Catherine Oriat \(LIG\)](#)

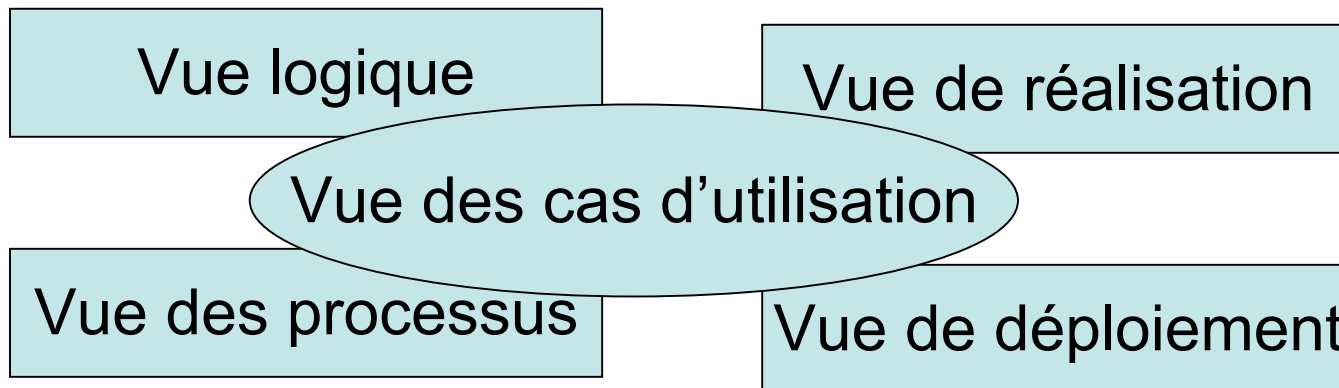
pour son livret « Analyse, Conception et Validation de logiciels »
qui inspire ce chapitre.

Plan

1. Architecture en couches
2. Architecture MVC
3. Utilisation de patrons de conception

Architecture logicielle

Elle décrit la structure générale du logiciel en constituants de haut niveau, ainsi que l'interaction entre ces éléments



Le modèle des 4+1 vue de Philippe Kruchten

Les vues

1. **La vue logique** : décrit l'organisation du système en sous-systèmes, couches, paquetages, classes et interfaces. --> diagramme de paquetage et classe
2. **La vue de réalisation** : concerne l'organisation des différents fichiers (exécutable, code source, documentation) --> diagramme de composants
3. **La vue des processus** : représente la décomposition du travail en différents flos d'exécution : processus, fil d'exécution. Importante dans le cas d'environnement multi-tâche.
4. **La vue de déploiement** : décrit les différentes ressources matérielles et l'implantation du logiciel sur ces ressources (liens réseau, performances du système, tolérance aux pannes) --> diagramme de déploiement
5. **La vue des cas d'utilisation** : sert à motiver et justifier les différents choix architecturaux --> diagramme de cas d'utilisation

Architecture en couche

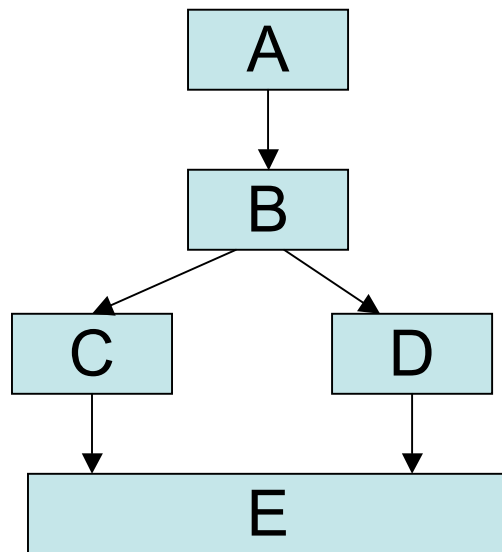
Le logiciel peut être organisé en **couche** : une couche regroupe un ensemble de classes et propose un ensemble cohérent de services à travers une interface.

Les couches sont ordonnées : les couches de plus haut niveau peuvent accéder à des couches de plus bas niveau mais pas l'inverse.

Dans une architecture **étanche**, une couche de niveau n n'accède que à la couche de niveau $n-1$

Architecture en couche

Exemple



A : couche de présentation

B : couche « application », réalisant la médiation entre l'interface graphique et les couches C et D.

C et D : 2 couches « domaine », (« métier ») contenant les principales classes du domaine d'applications, conçues pour être indépendante de l'interface graphique et de l'infrastructure

E : couche infrastructure contenant les services techniques bas niveau

Avantage d'une architecture en couche

1. Maintenance

- Le système est plus facilement modifiable. Une modification de couche n'affecte pas les couches inférieures. Une modification de couche qui ne change pas d'interface n'affecte pas les couches supérieures

2. Réutilisation

- Des éléments de chaque couche peuvent être réutilisés. Par exemple, les couches « métier » peuvent être communes à plusieurs applications

3. Portabilité

- On confine les éléments qui dépendent du système aux basses couches. Suivant le système d'exploitation, on change les couches basses, mais les couches hautes restent identiques

Inconvénients d'une architecture en couche

Si on a un grand nombre de couches, et si en plus ces couches sont étanches, l'appel de fonction de bas niveau est **moins efficace**, puisqu'il faut traverser toutes les couches pour parvenir à ces fonctions.

Il faut donc trouver **un compromis** entre une bonne encapsulation et une bonne efficacité.

Architecture Modèle-Vue-Contrôleur

Fréquemment utilisée pour les IHM. Introduite par SmallTalk en 1980, elle très largement répandue aujourd'hui : Struts, Spring, Zend, etc.

Le logiciel est découpé en 3 parties :

- Le modèle : les données métier.
- La vue : traitement des sorties.
- Le contrôleur : traitement des interactions (entrées).

Entrées --> Traitement --> Sorties

Contrôleur --> Modèle --> Vue

Architecture MVC

1. Modèle

- comporte les classes principales correspondant aux différentes fonctionnalités de l'application : données et traitements. Indépendamment des parties « vue » et « contrôleur », elle effectue des actions en réponse aux demandes de l'utilisateur (par l'intermédiaire de la partie contrôleur), et informe la partie vue des changements d'états du modèle pour sa mise à jour.

2. Vue

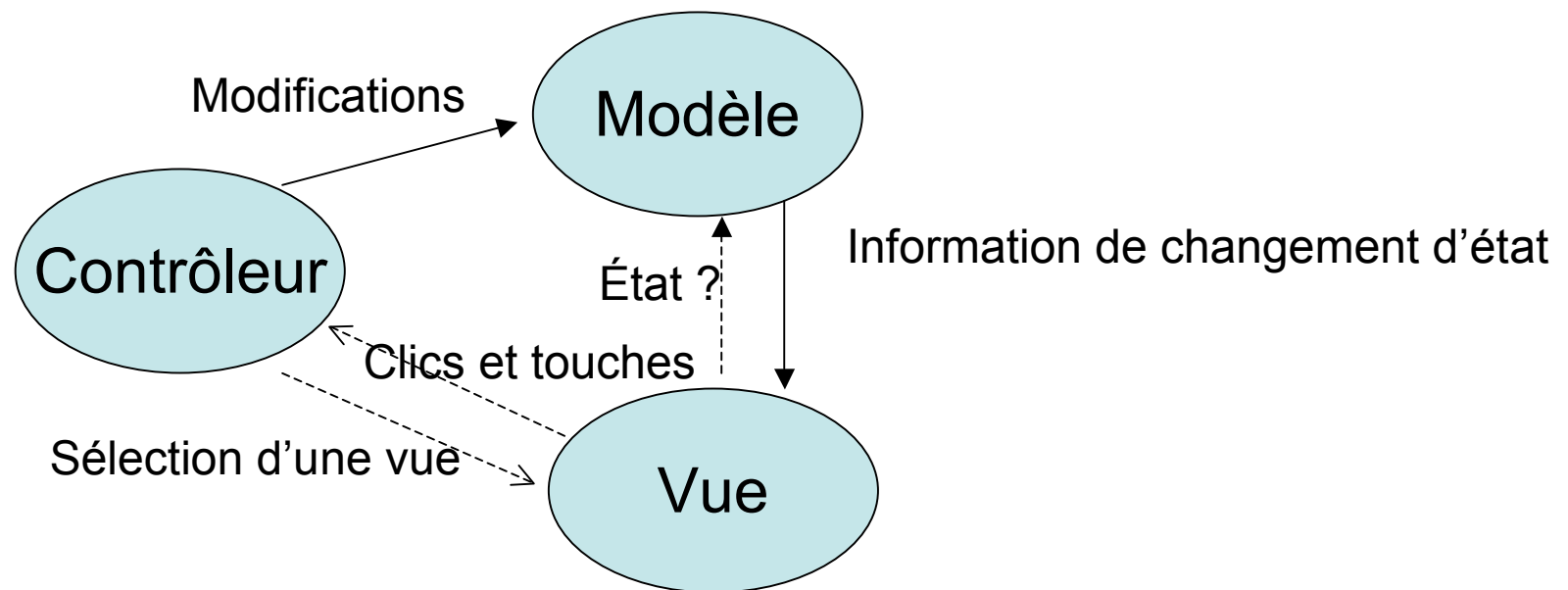
- Comporte les classes relatives à l'interface graphique (ce que l'utilisateur voit). Cette partie est passive : elle est informée des changements d'états du modèle et se met à jour lors de ces changements.

3. Contrôleur

- Récupère les actions de l'utilisateur (clics de souris, touches clavier) et associe ces événements à des actions qui modifient le modèle.

Interactions MVC

1. Le contrôleur, en fonction des événements qu'il reçoit de l'utilisateur, effectue des modifications du modèle.
2. Ces modifications du modèle sont ensuite transmises à l'interface graphique.



Avantages d'une architecture MVC

1. Vues multiples

- Gestion et affichage de plusieurs vues du même modèle possible et facile

2. Portabilité

- Le portage de l'interface sur d'autres plateformes est possible, sans avoir à modifier le noyau de l'application

3. Evolution

- L'amélioration de l'interface graphique (ajout de lignes dans un menu, ou de boutons) est plus facile. De plus, ces modifications peuvent se faire lors de l'exécution du logiciel. --> personnalisation des pages Web.

Patrons de conception

L'objectif des patrons de conception (« design patterns » en anglais) est de recueillir l'expérience et l'expertise des programmeurs, afin de la transmettre à d'autres programmeurs.

Les patrons de conception orientés objet reprennent cette idée de fournir un **catalogue de solutions** classiques à des problèmes de conception objet.

Un patron

Un patron de conception est la description d'un problème récurrent dans un certain contexte, accompagné d'une description des différents éléments d'une solution à ce problème.

- Le nom
- Le problème
- Le contexte
- La solution
- Les conséquences (avantages + inconvénients)

Les différentes sortes de patrons

1. Analyse et définition des besoins
 - Résoudre les problèmes de communications entre les utilisateurs et les informaticiens, évaluer la complexité
2. Analyse et conception
 - Définir une architecture adéquate (patrons architecturaux), résoudre des sous-problèmes, améliorer la communication entre développeurs (patrons de conception)
3. Mise en œuvre
 - Produire un code correct et facile à maintenir. On utilise des patrons de programmation ou *idiomes*

Les patrons de conception

1. Patrons de création : création d'objet
2. Patrons structurels : structure des objets et relations entre ces objets
3. Patrons comportementaux : relatifs au comportement des objets.

Singleton

1. But : s'assurer qu'une classe n'a qu'une instance unique et en donner un accès global.
2. Motivation : ce patron peut servir dans beaucoup de circonstances, en particulier lorsque la classe correspond à un objet unique dans le monde réel.
3. Exemple : système de fichiers, gestionnaire de fenêtres, système de comptabilité pour une entreprise, classe de configuration pour une application Web.

Singleton : code

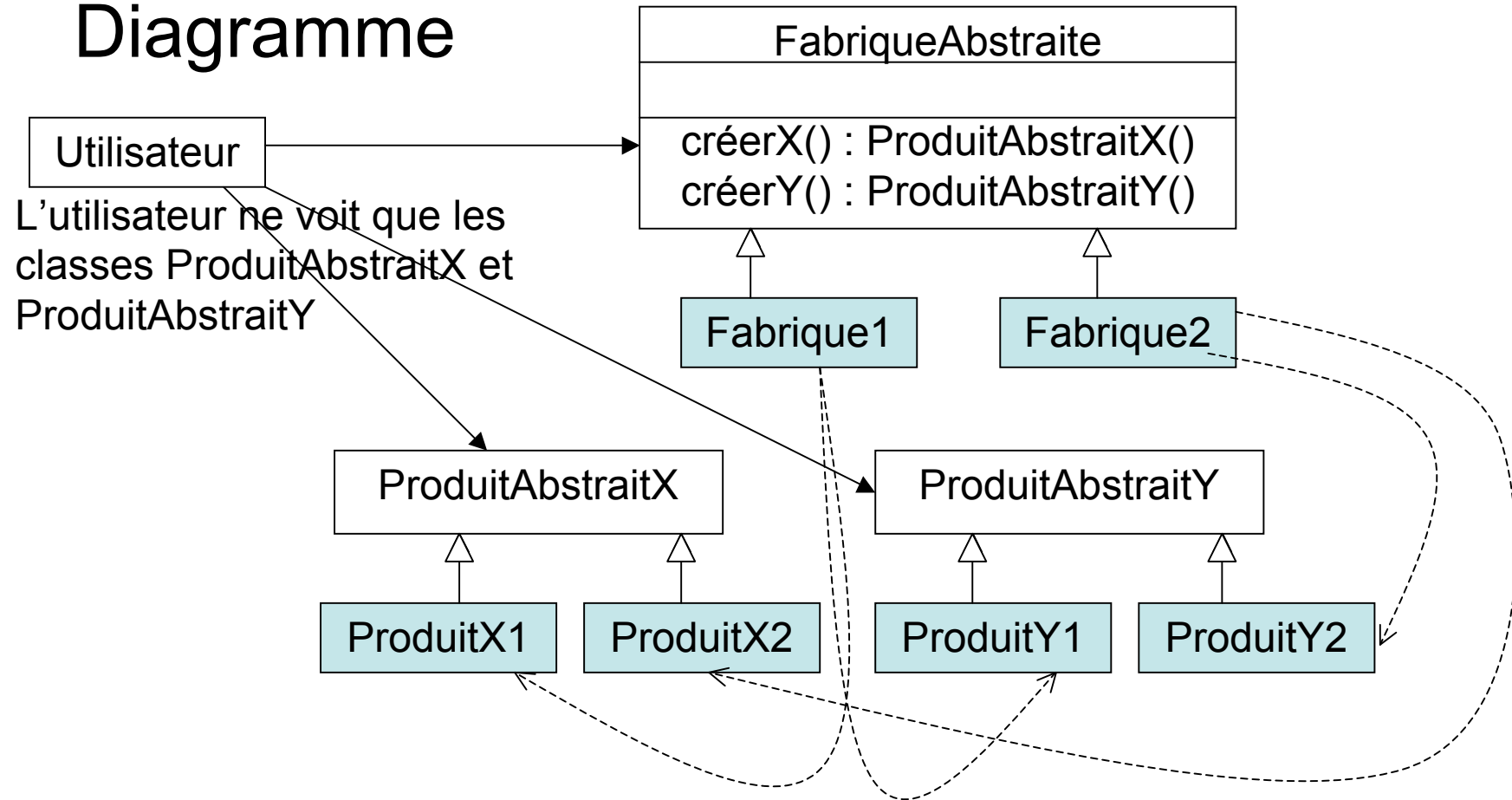
```
public class Singleton {  
  
    /* Attribut privé contenant l'instance unique Singleton */  
    final private static Singleton _myInstance = new Singleton();  
  
    /* Accesseur à l'instance unique de Singleton */  
    static Singleton instance() {  
        return _myInstance;  
    }  
  
    /* Constructeur privé */  
    private Singleton(){  
        //do the job  
    }  
  
}
```

Fabrique abstraite (patron de création)

1. 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
2. Principe : l'utilisateur accède uniquement à différentes classes abstraites (une par produit) *ProduitAbstraitX* et *ProduitAbstraitY* et à une classe qui permet de créer des instances de ces produits : *Fabrique1* ou *Fabrique2*. Si l'utilisateur utilise la *Fabrique1*, il obtient des instances de *ProduitX1* et *ProduitY1*. Sinon, avec la *Fabrique2*, il obtient les instances de *ProduitX2* et *ProduitY2*

Fabrique abstraite

Diagramme



Fabrique abstraite

1. Avantages

- Le patron facilite l'utilisation cohérente des différents produits : si le client utilise toujours la même fabrique, il est sûr d'avoir toujours des produits de la même famille.
- L'utilisateur peut facilement changer de famille de produits : il suffit de changer de fabrique.
- On peut facilement ajouter une nouvelle famille de produits en ajoutant une nouvelle sous-classe pour chaque produit abstrait.

2. Inconvénient :

- Il peut-être difficile d'ajouter de nouveaux produits puisqu'il faut modifier toutes les classes qui dérivent de `FabriqueAbstraite`

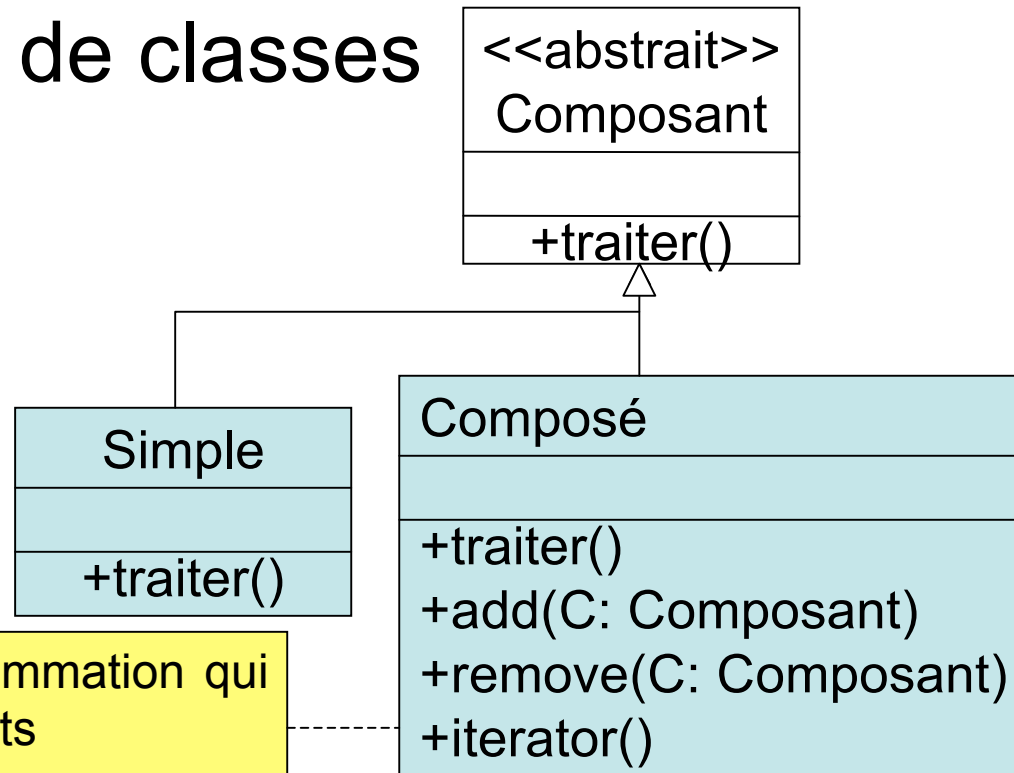
3. Rq : chaque Fabrique peut s'implémenter avec le patron Singleton

Objet composite (patron structurel)

1. 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.
2. Solution : Employer une classe abstraite Composite contenant une ou plusieurs méthodes abstraites de traitement.

Objet Composite

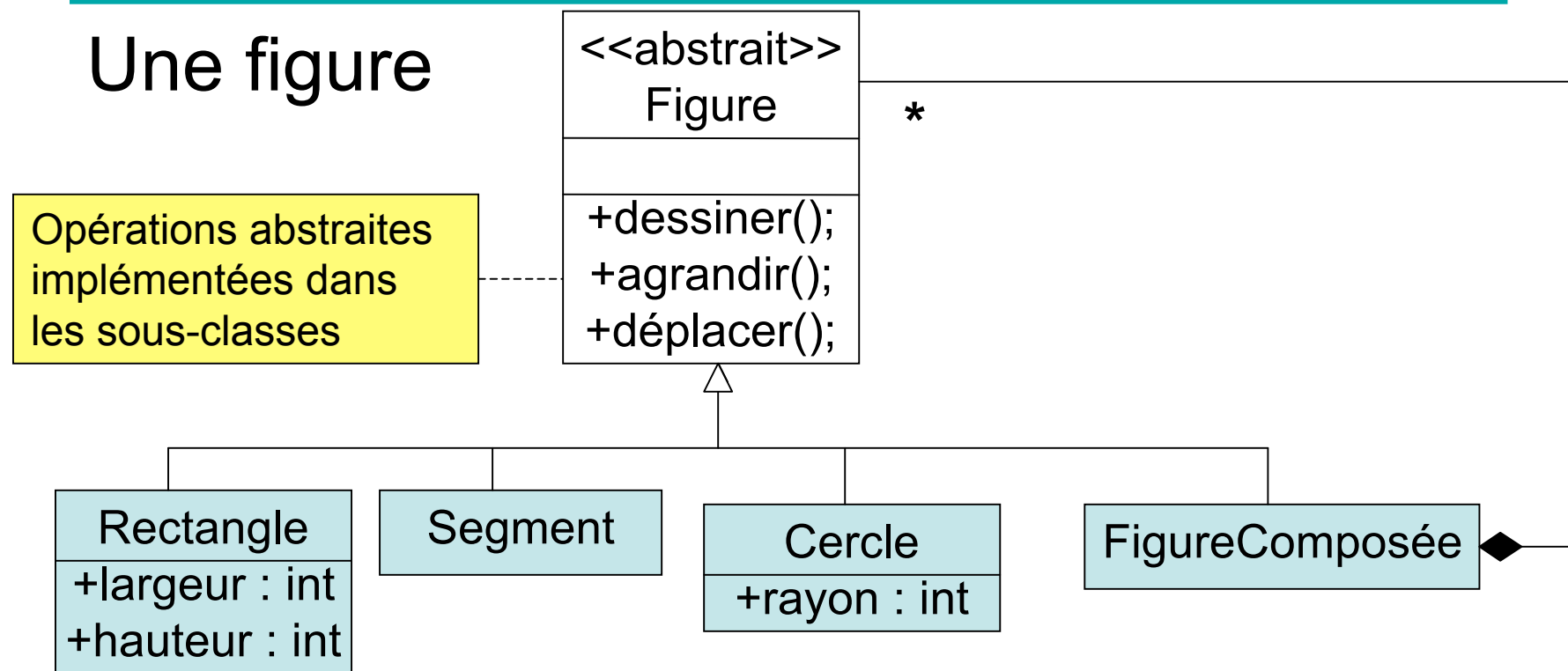
1. Diagramme de classes



Iterator est un idiome de programmation qui permet d'itérer sur les composants

```
Iterator iter = list.iterator();
while (iter.hasNext()) {
    Object o = iter.next();
    // do the job
}
```

Objet Composite : exemple



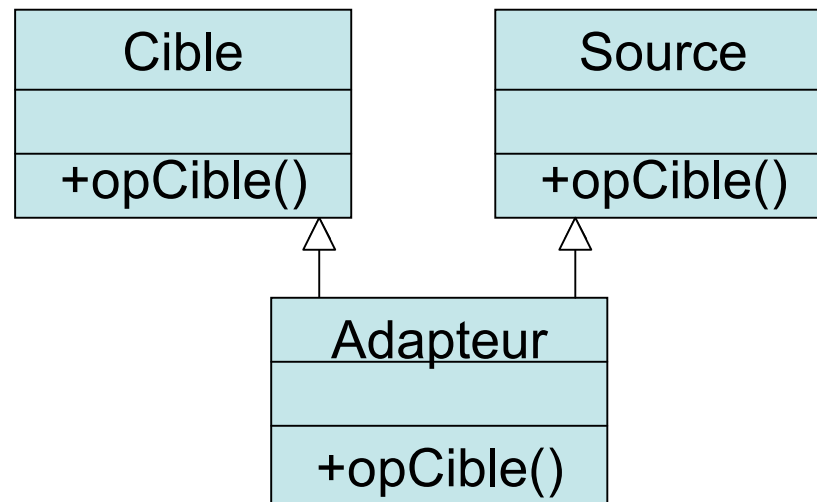
Les objets Cercle, Rectangle, Segment héritent de Figure (opérations : dessiner, agrandir, déplacer) et appartiennent à FigureComposée

Adaptateur (patron structurel)

1. But : implémenter une interface «Cible » en utilisant une classe « Source » qui ne respecte pas cette interface mais est déjà utilisée ailleurs.
2. Solutions
 - Par héritage : réaliser une sous-classe de Source qui implémente la Cible.
 - Par délégation : créer un lien entre l'adaptateur et la source, et l'adaptateur doit déléguer le travail à effectuer à la classe source. La source joue le rôle du délégué.

Adaptateur par héritage

1. Diagramme

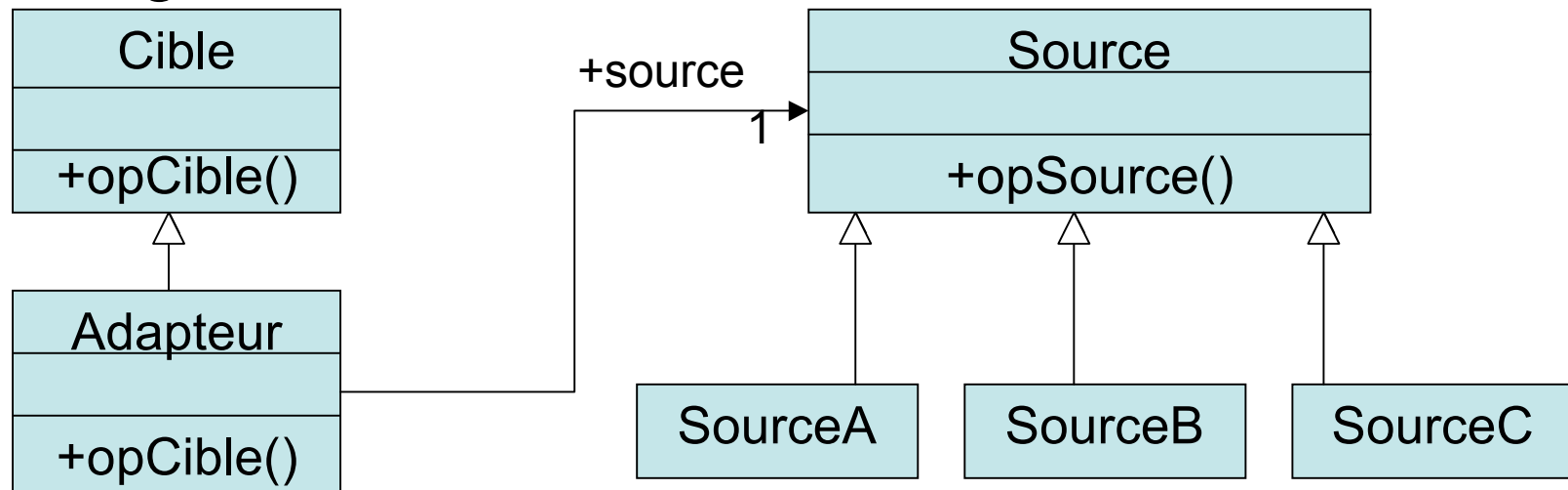


2. Inconvénient

- on ne peut pas adapter les sous-classes de **Source**, et on peut redéfinir les méthodes de **Source**.

Adaptateur par délégation

1. Diagramme :



2. Avantages

- On peut adapter les sous-classes de **Source**: **SourceA**, **SourceB**, **SourceC**
- Cette solution empêche la redéfinition des méthodes de **Source**

Patrons stratégie, commande et état

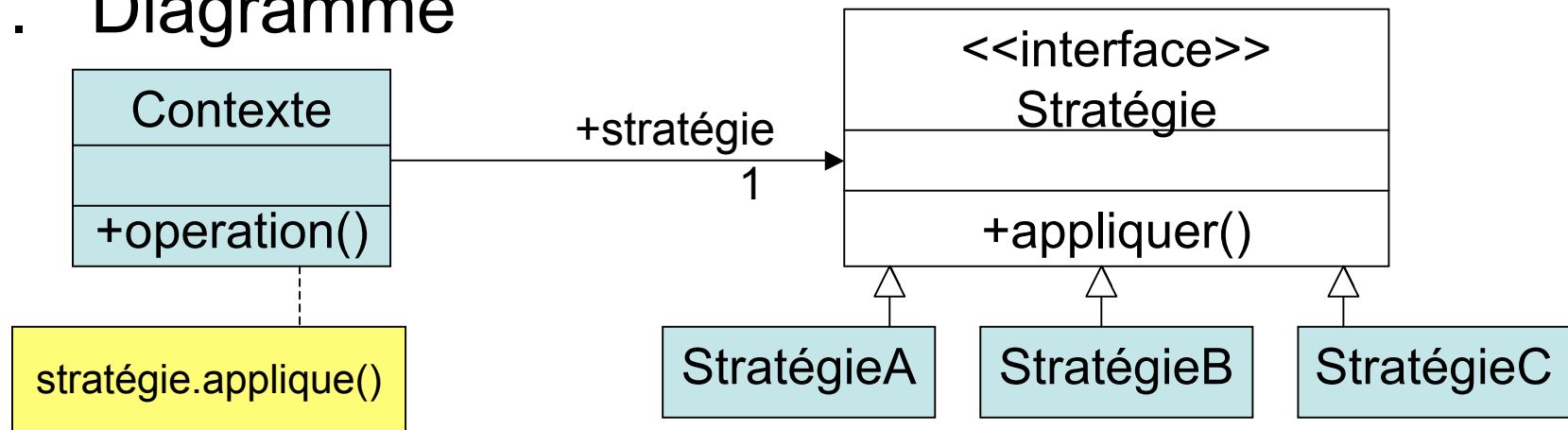
1. Ces trois patrons comportementaux sont assez similaires et consistent à :
 - Associer à certains traitements des objets
 - Définir une hiérarchie de classes pour effectuer ces traitements de façon uniforme
 - Découpler les données et les traitements : la hiérarchie de classe des traitements est indépendante de celle des objets qui vont utiliser ces opérations.

Patron Stratégie

1. But : définir une famille d'algorithmes encapsulés dans des objets, afin que ces algorithmes soient interchangeables dynamiquement.
2. Motivation : pour résoudre un problème il existe souvent plusieurs algorithmes ; dans certains cas, il peut être utile de à l'exécution quel algorithme utiliser, par exemple, selon des critères de temps de calcul ou de place mémoire

Patron Stratégie

1. Diagramme



2. Avantages

- La classe Contexte peut avoir des sous-classes indépendantes des Stratégies
- On peut changer de stratégie dynamiquement

3. Inconvénients :

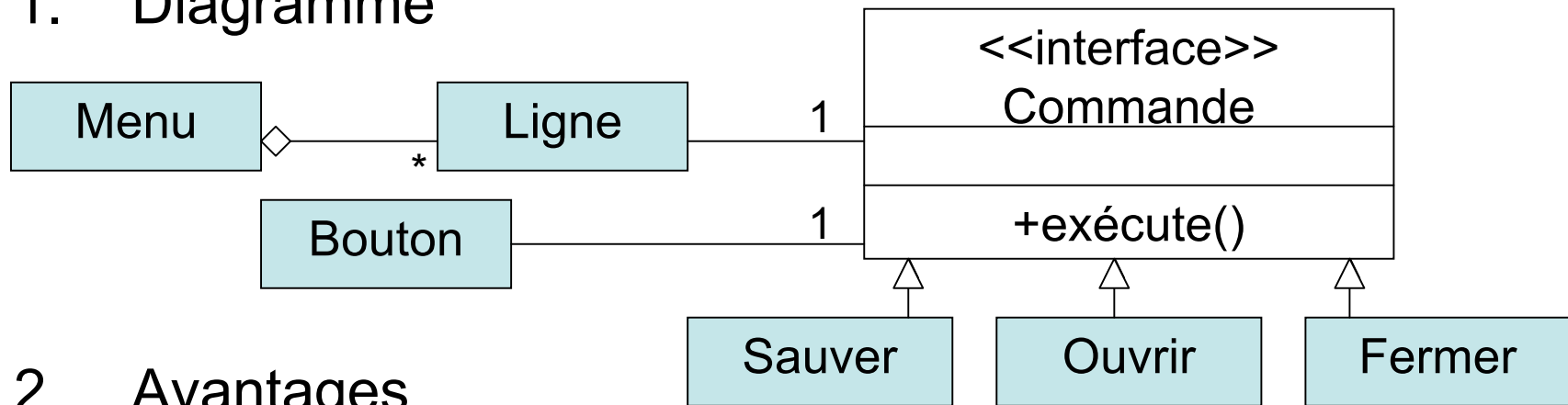
- Surcroît en place mémoire (* nombre d'objet *Stratégie*)
- Un surcroît en temps d'exécution à cause de l'indirection

Patron Commande

1. But : encapsuler les commandes dans des objets
2. Motivation : dans des applications avec IHM, on peut associer des commandes à des boutons ou à des lignes dans des menus ; revenir en arrière d'une ou plusieurs commandes (« annuler ») ou repartir en avant (« rétablir»). Pour cela, on doit associer à des actions des objets que l'on peut stocker, passer en paramètres.

Patron Commande

1. Diagramme

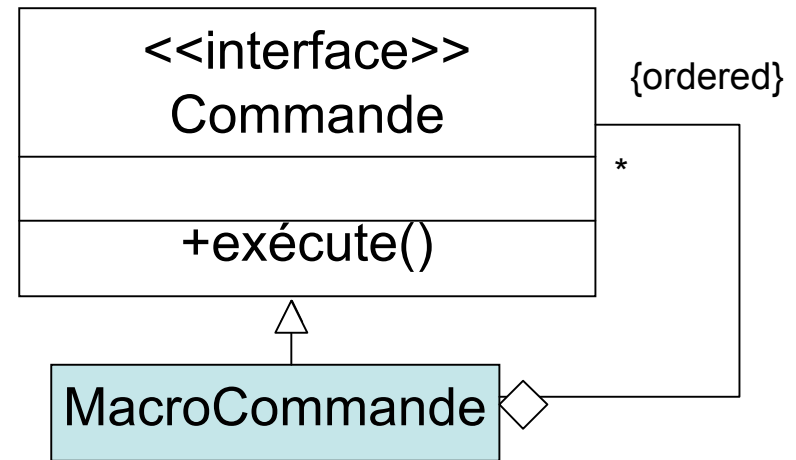


2. Avantages

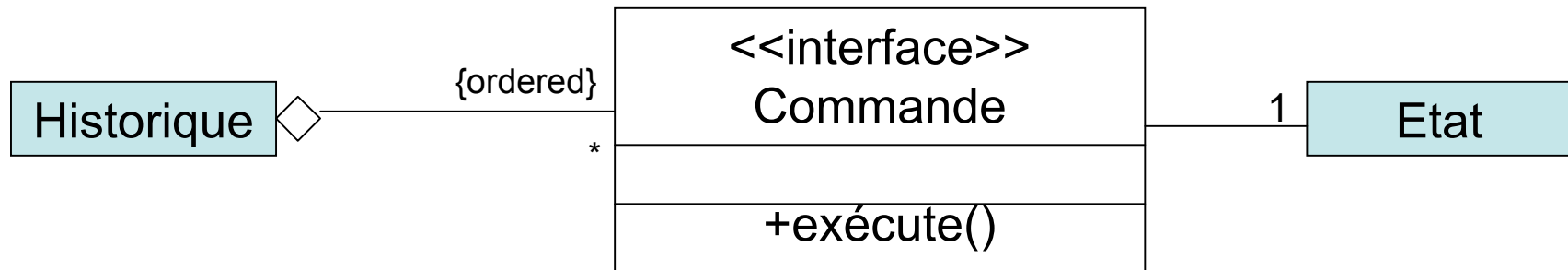
- On peut modifier une association Ligne-Commande à l'exécution afin de paramétrer le logiciel à l'exécution
- On peut effectuer la même action par différents moyens
- On peut définir des macros-commandes, composées de séquences de commandes
- On peut revenir en arrière (annuler, rétablir) : cela nécessite de stocker l'historique des commandes, et un état interne à chaque commande effectuée.

Patron commande

1. Macro commande

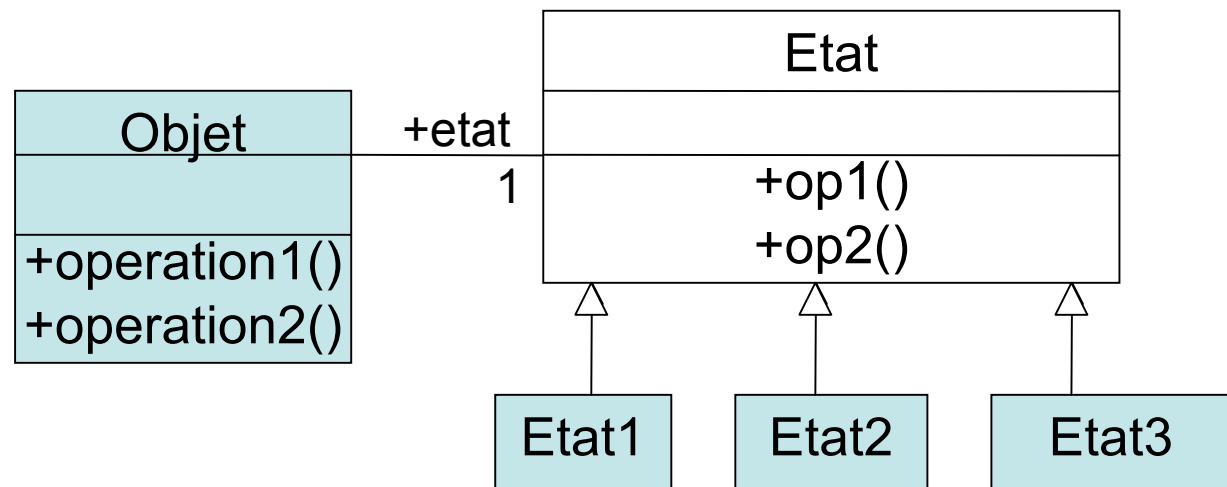


2. Historique



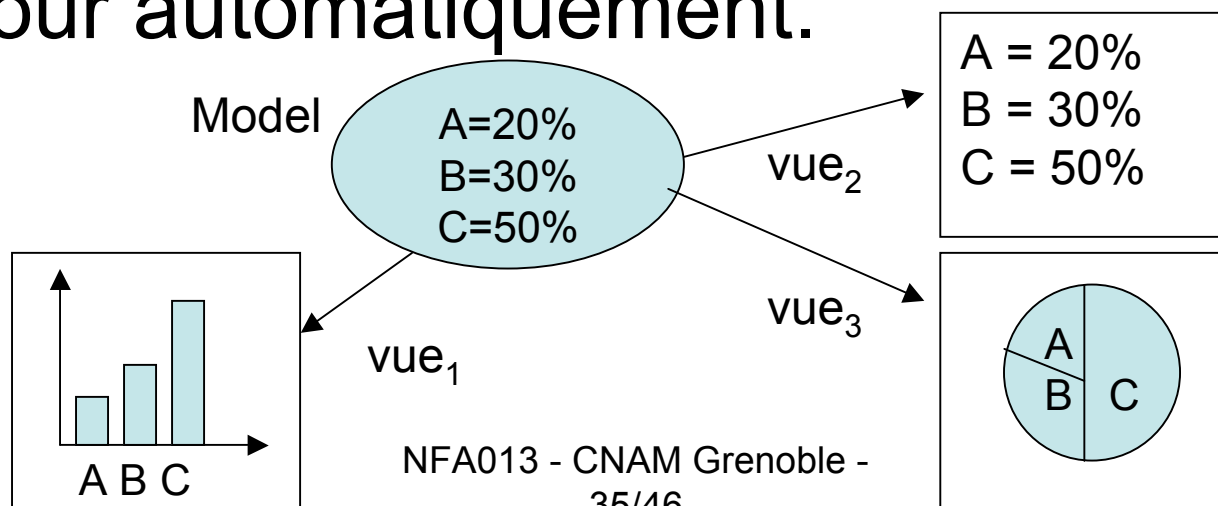
Patron Etat

Le patron Etat permet de réaliser des objets dont le comportement change lorsque leur état interne est modifié.



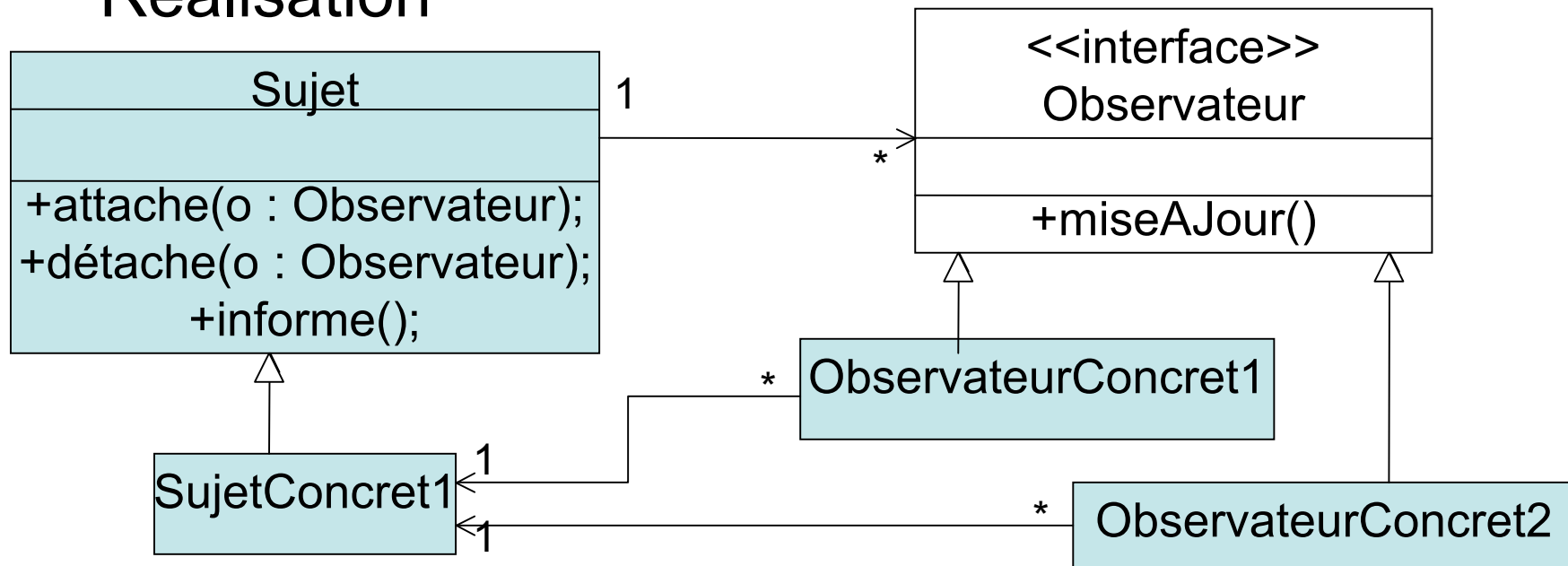
Patron observateur (patron comportemental)

1. But : définir une dépendance entre un objet (le *sujet*) et un ensemble d'objets (les *observateurs*) de sorte que lorsque le sujet change d'état, tous ses observateurs soient informés et mis à jour automatiquement.



Observateur

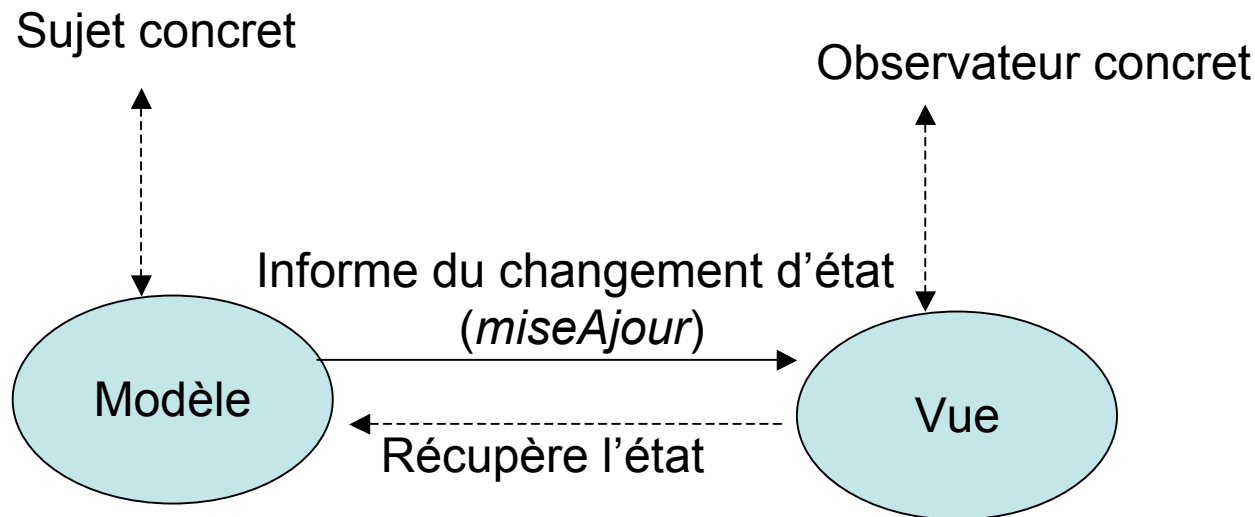
Réalisation



- une classe abstraite *Sujet*, dont hérite toutes les classes pouvant servir de sujet.
- une interface *Observateur*, qui contient une méthode *miseAJour()*
- Un sujet est en relation avec un ensemble d'observateurs (qu'on peut ajouter ou enlever de cet ensemble via *attacher()* et *détacher()*).
- Lorsque le sujet change d'état, il utilise *miseAJour()* qui notifie tous les observateurs du changement d'état.

Observateur et MVC

Le patron observateur est un moyen d'implémenter en partie une architecture MVC. Les classes qui héritent de *Sujet* sont des classes du **modèle**, et les classes qui implémentent *Observateur* sont des classes de la **vue**.



Interprète (patron comportemental)

1. But : Il permet de définir une représentation abstraite de la grammaire d'un langage; le langage est ensuite interprété en fonction du contexte associé à la grammaire.
2. Applications : pour les opérations de calcul dirigées par la syntaxe, en particulier en compilation
 - Evaluation (pour écrire un interprète au sens propre)
 - Vérification de type
 - Génération de code
 - Etc.

Interprète

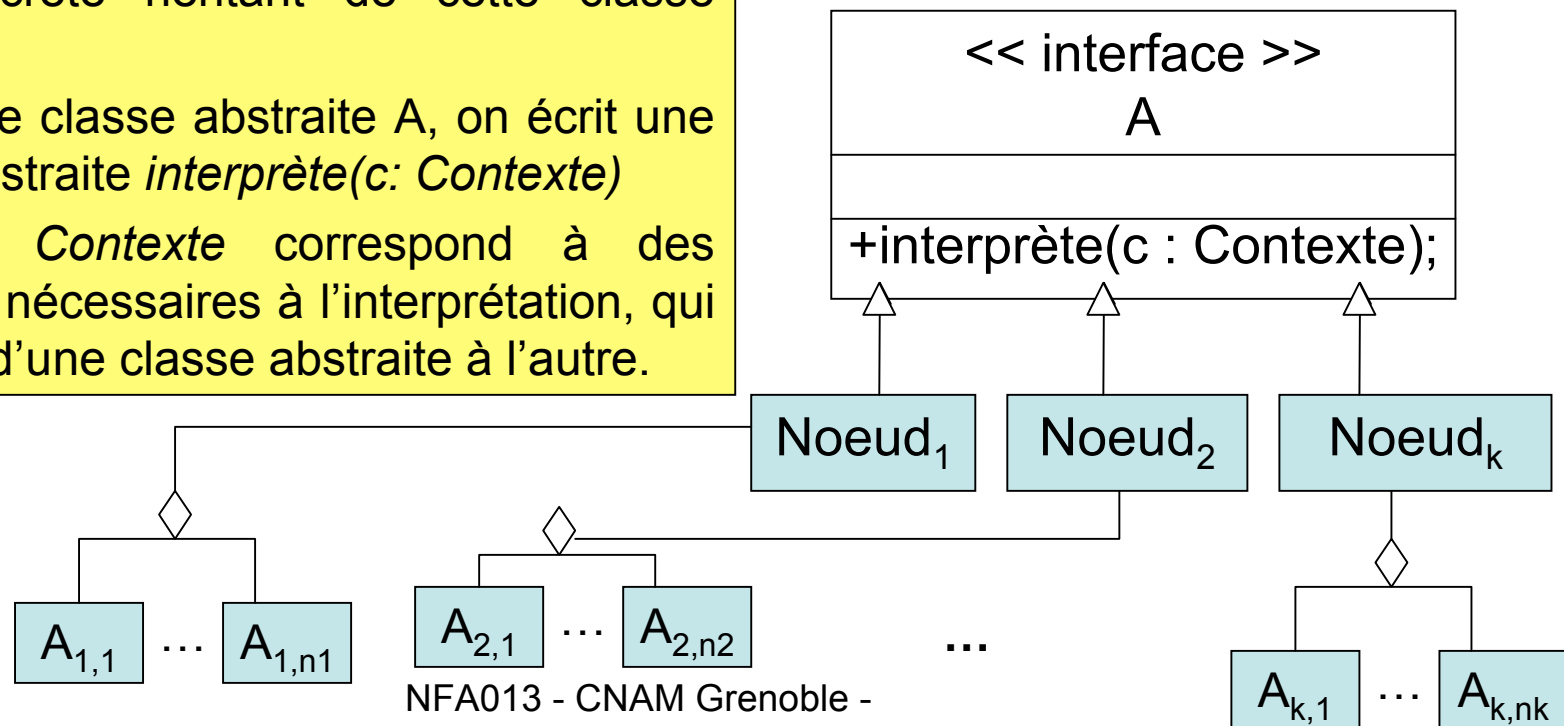
Soit la grammaire :

$$A \rightarrow \text{Noeud}_1(A_{1,1}, \dots, A_{1,n1}) \mid \text{Noeud}_2(A_{2,1}, \dots, A_{2,n2}) \mid \dots \mid \text{Noeud}_k(A_{k,1}, \dots, A_{k,nk})$$

On associe à chaque non terminal une classe abstraite, et à chaque nœud une classe concrète héritant de cette classe abstraite.

Pour chaque classe abstraite A , on écrit une méthode abstraite *interprète*(c : Contexte)

La classe *Contexte* correspond à des paramètres nécessaires à l'interprétation, qui peut varier d'une classe abstraite à l'autre.



Interprète

1. 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.

2. Inconvénients

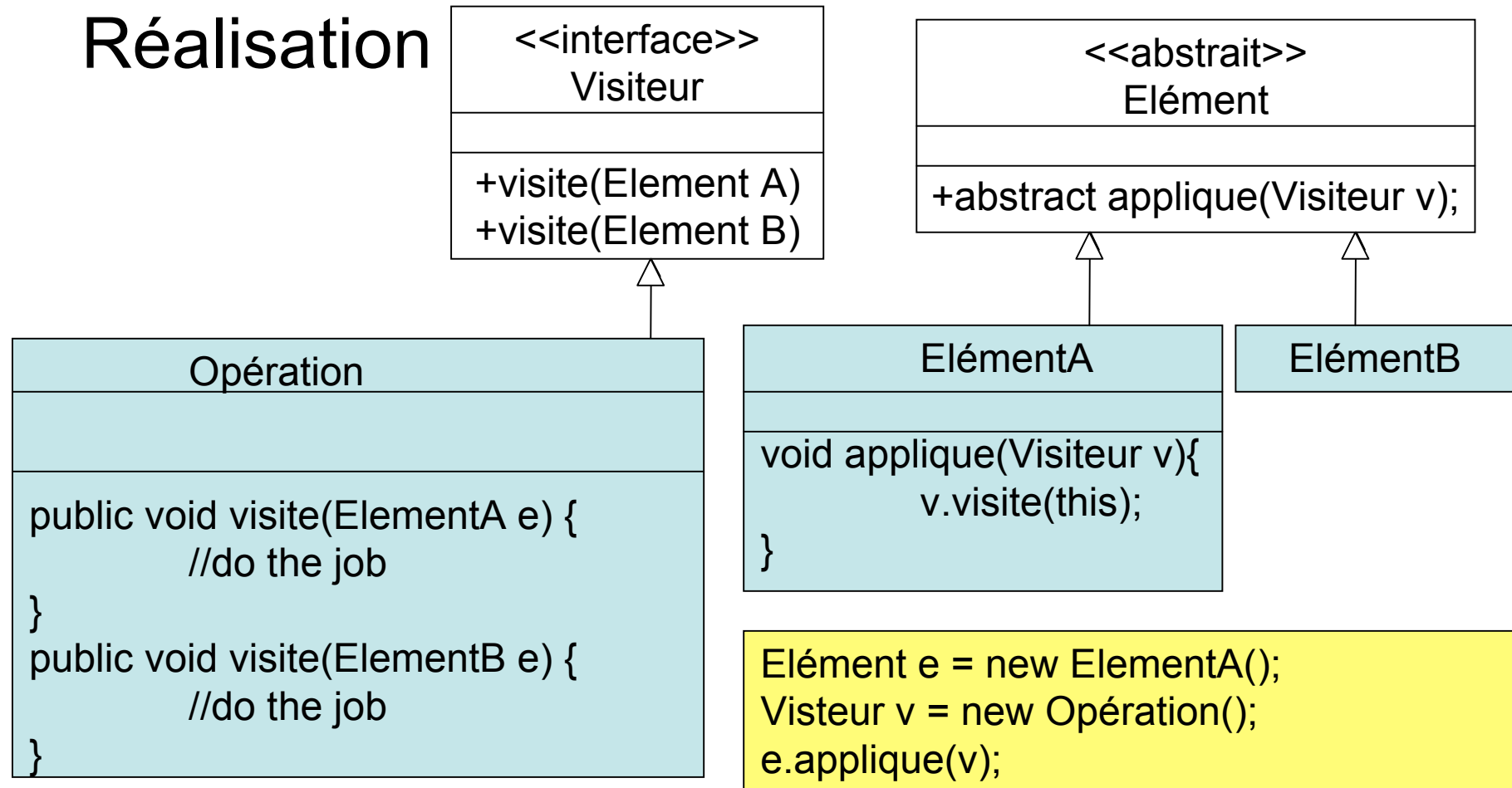
- Multiplication des classes lorsque la grammaire est complexe
- L'ajout d'opérations est délicat, car celles-ci doivent être ajoutées dans toutes les classes de la hiérarchie.

Visiteur (patron comportemental)

1. But : représenter une opération définie en fonction de la structure d'un objet. Il permet de définir de nouvelles opérations sans modifier les classes qui définissent la structure des objets auxquelles elles s'appliquent.
2. Intérêt :
 - Favorise la réutilisation (on récupère facilement, par exemple, une structure d'arbre abstrait)
 - Code parallèle de plusieurs équipes : une travaille avec la structure, l'autre implémente les opérations
 - Partage de structure d'objets entre plusieurs applications

Visiteur

Réalisation



Visiteur

Sans le patron Visiteur :

- répartir tout le code de l'opération dans les différentes classes de la hiérarchie. Cela correspond à une application du patron Interprète.
- Ecrire une classe Opération qui teste les différents cas à l'aide de *instanceof* (programmation non objet) et convertit ensuite au bon type (*casting*)

```
Class Opération {  
    static void operation(Element e){  
        if (e instanceof ElémentA) {  
            ElémentA a = (ElémentA) e;  
            //do the job  
        } else if (e instanceof ElémentB) {  
            ElémentB b = (ElémentB) e;  
            //do the job  
        }  
    }  
}
```

Teste le type

Convertit au bon type

USAGE

```
Elément e = new ElémentA();  
Opération.operation(e);
```

Visiteur

1. Avantages

- Il oblige à traiter tous les cas (vérifié à la compilation)
- Programmation « purement » objet
- Il évite d'effectuer n tests pour trouver le code à exécuter
- Ajout de nouvelles opérations sans modifications de la hiérarchie

2. Inconvénients

- Lourd à mettre en œuvre : prévoir une méthode *applique* par classe
- Le traitement de méthodes comportant des paramètres **et** un résultat (non *void*) est lourd.
- Surcoût à chaque indirection applique --> visite
- Le code est peu lisible lorsqu'on ne connaît pas le patron.

Références

1. Sites Web

- <http://tahe.developpez.com/>
- <http://laurent-audibert.developpez.com/>
- <http://uml.free.fr/index-cours.html>

2. Outils UML

BOUML : <http://bouml.free.fr/index.html>

JUDE : <http://jude.change-vision.com/jude-web/index.html>

Comparatif (outils commerciaux UML pour base de données):

<http://christian-soutou.developpez.com/tutoriel/uml/bdd/>

Bibliographie

– UML :

- G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language User Guide*, Addison-Wesley, 1999
- M. Blaha, J. Rumbaugh. *Modélisation et conception orientées objet avec UML 2*, Pearson Education, 1995.

– Schéma de conception :

- Gamma E., Helm R., Johnson R., Vlissides J., *Design patterns. Catalogue de modèles de conception réutilisables*, Vuibert 1999.
Traduction de Jean-Marie Lasvergères.

– Java :

- Bloch J., *Java Efficace, guide de programmation*, Vuibert, 2002.
Traduction d'Alexis Moussine-Pouchkine.
- Flanagan D., *Java in a Nutshell, 4^{ème} édition, manuel de référence*, O'Reilly, 2002, traduction de Alexandre Gachet.

– RUP :

- Pierre-Yves Cloux : RUP, XP, architectures et outils. Industrialiser le processus de développement