

Patrons comportementaux

- Les patrons comportementaux aident à l'implémentation.
- Comment partager une fonction « presque » identique
- Comment faire un mécanisme d'undo ?
- Comment répondre à F1 (aide) ?
- Comment maintenir ma représentation graphique à jour ?
- Comment implémenter efficacement un protocole à état ?
- Comment permettre de choisir l'algorithme employé ?
- On s'est débarrassé des switch, comme supprimer les instanceof ?

Patrons comportementaux

- Template Method: pas de redondance
- Command : un objet représentant une action
- Chain of responsibility : si je sais faire, je traite, sinon, je transmets à mon chef
- Observer : être prévenu des changements d'un objet
- State : quand le traitement dépend de l'état
- Strategy : plusieurs choix d'algorithme
- Iterator : accéder simplement aux éléments d'une collection
- Visitor : mort aux instanceof !

Template method : quand

- Un des plus simples !
- À la limite du DP et du simple principe de POO
- Deux composants ont des similarités significatives, mais sans partage d'interface ou d'implémentation ?
 - Les changements vont devoir être dupliqués !

Template method: comment

- Définir le squelette d'un algorithme tout en déléguant certaines étapes aux sous-classes.
- Les sous-classes peuvent redéfinir certaines étapes de l'algorithme sans en changer la structure

Template method: exemple

```
class SortUp { ///// Shell sort /////
    Public void sort( int v[], int n ) {
        for (int g = n/2; g > 0; g /= 2)
            for (int i = g; i < n; i++)
                for (int j = i-g; j >= 0; j -= g)
                    if (v[j] > v[j+g]) doSwap(v[j],v[j+g]);
    }
    Private void doSwap(int& a,int& b) {...}
}

class SortDown {
    Public void sort( int v[], int n ) {
        for (int g = n/2; g > 0; g /= 2)
            for (int i = g; i < n; i++)
                for (int j = i-g; j >= 0; j -= g)
                    if (v[j] < v[j+g]) doSwap(v[j],v[j+g]);
    }
    Private void doSwap(int& a,int& b) {...}
};

... main( ...) {
    Int[] t = new int[size] { ... }

    SortDown downObj;
    downObj.sort( array);
}
```

```
class AbstractSort { ///// Shell sort /////
    Public void sort( int v[], int n ) {
        for (int g = n/2; g > 0; g /= 2)
            for (int i = g; i < n; i++)
                for (int j = i-g; j >= 0; j -= g)
                    if (needSwap( v[j], v[j+g] ))
                        doSwap(v[j], v[j+g]);
    }
    Private int needSwap(int,int) = 0;
    void doSwap(int& a,int& b) {... }
}

class SortUp extends AbstractSort {
    int needSwap(int a, int b) { return (a > b);}
}

class SortDown extends AbstractSort {
    int needSwap(int a, int b) {return (a < b); }
}

... main( ...) {
    Int[] t = new int[size] { ... }

    AbstractSort downObj = new SortDown();
    downObj.sort( array);
}
```

Template method: exemple

```
abstract class Generalization {
    // 1. Squelette d'un algo dans une "template method"
    public void findSolution() {
        stepOne();
        stepTwo();
        stepThr();
        stepFor();
    }
    // 2. implémentation par défaut pour certaines étapes
    protected void stepOne()
        { System.out.println( "Generalization.stepOne" ); }
    protected void stepFor()
        { System.out.println( "Generalization.stepFor" ); }
    // 3. certaines étapes doivent être impl. dans les classes dérivées
    // Simple "placeholder", "hook"
    abstract protected void stepTwo();
    abstract protected void stepThr();
}
```

```
abstract class Specialization extends Generalization {
    // 4. les classes dérivées peuvent impl. certaines étapes
    // + 1. Squelette d'un algo dans une "template method"
    protected void stepThr() {
        step3_1();
        step3_2();
        step3_3();
    }
    // 2. implémentation par défaut pour certaines étapes
    protected void step3_1() { System.out.println( "Specialization.step3_1" ); }
    protected void step3_3() { System.out.println( "Specialization.step3_3" ); }
    // 3. certaines étapes doivent être impl. dans les classes dérivées
    abstract protected void step3_2();
}
```

```
class Realization extends Specialization {
    // 4. les classes dérivées peuvent impl. certaines étapes
    protected void stepTwo() { System.out.println( "Realization .stepTwo" ); }
    protected void step3_2() { System.out.println( "Realization .step3_2" ); }
    // 5. les classes dérivées peuvent redéfinir certaines étapes
    // 6. les classes dérivées peuvent redéfinir des étapes
    // tout en utilisant l'impl. de la classe de base
    protected void stepFor() {
        System.out.println( "Realization .stepFor" );
        super.stepFor();
    }
}
```

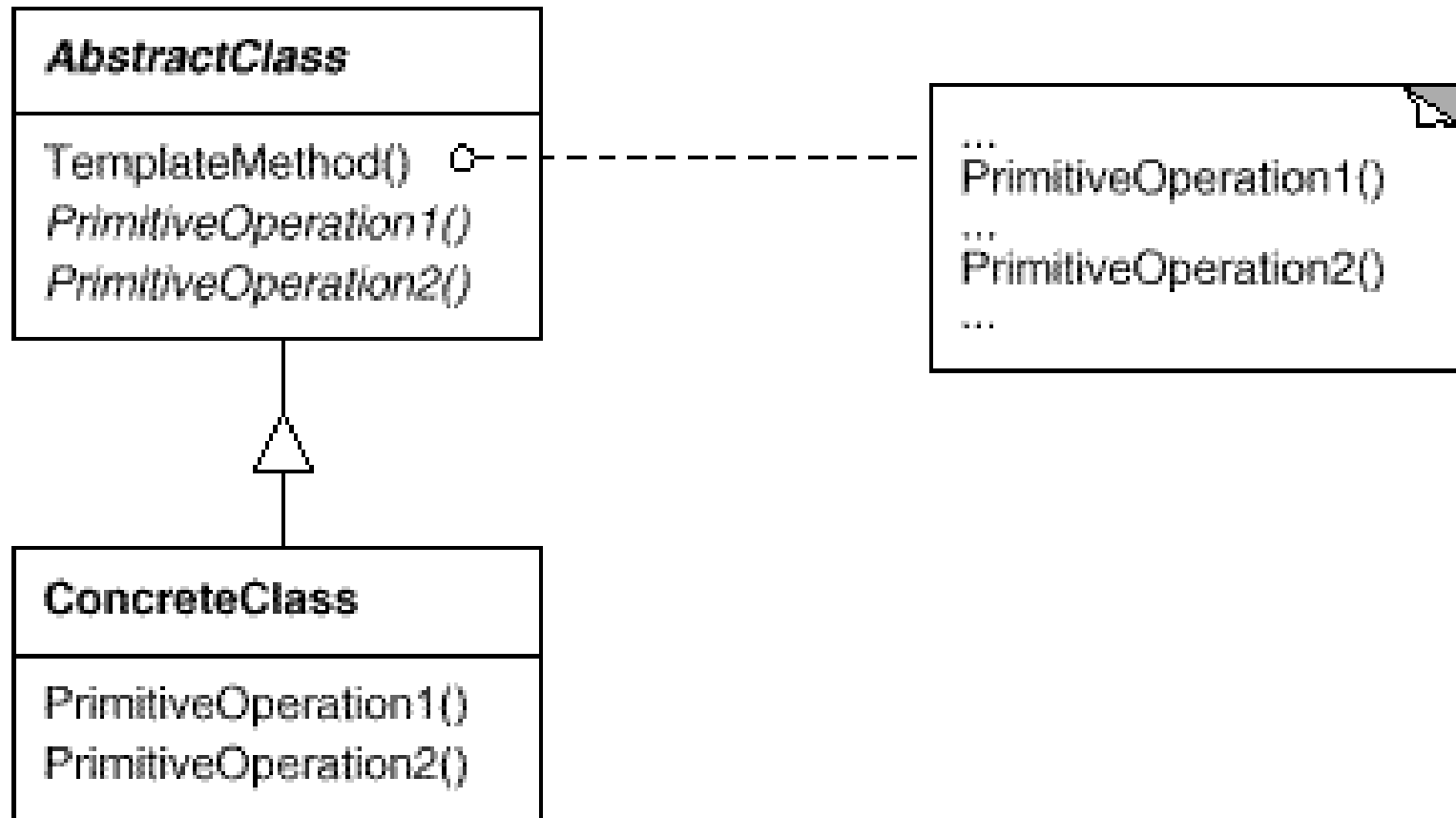
```
class TemplateMethodDemo {
    public static void main( String[] args ) {
        Generalization algorithm = new Realization();
        algorithm.findSolution();
    }
}
```

```
// Generalization.stepOne
// Realization .stepTwo
// Specialization.step3_1
// Realization .step3_2
// Specialization.step3_3
// Realization .stepFor
// Generalization.stepFor
```

Template method check-list

- Examiner l'algorithme, décider quelles étapes sont standards et lesquelles sont particulières à chaque classe
- Définir une classe abstraite pour héberger la "template method" et déplacer dans la classe de base la structure de l'algo (la "template method") et la définition de toutes les étapes standard.
- Définir les méthodes "hook/placeholder" dans la classe de base pour chaque étape qui requiert des implémentations différentes. Une implémentation par défaut peut optionnellement être fournie
- Invoquer les méthodes « hook » depuis la template method.
- Définir chacune de ces classes comme une classe dérivée de cette classe abstraite
- Supprimer l'algorithme des classes dérivées (maintenant implémenté dans la classe de base) ou les étapes avec impl. par défaut.
- Il ne reste dans les classes dérivées que les détails spécifiques à chaque classe

Template method



Chaîne de responsabilité : quand

- On cherche à traiter des requêtes (au sens le plus général)
- On ne veut pas figer l'objet qui y répond, mais le choisir en fonction de la nature de la requête
- Les objets qui répondent savent traiter une partie, mais pas toutes les requêtes
- Par contre ils connaissent un objet qui saurait répondre

Chaîne de responsabilité : comment

- Chaque objet traitant la requête a deux choix :
 - traiter le requête,
 - déléguer à son chef/parent/frère
- Le choix de l'un ou de l'autre dépend du type de la requête
- En général, deux manières de faire
 - si on ne sait pas faire, on délègue au suivant/chef
 - si le suivant/chef ne sait pas faire, on s'en charge
- On peut aussi utiliser un composite : pour déléguer au suivant, le composant renvoie sur son composite

Chaîne de responsabilité : exemple

- Les ClassLoader en java
- Les systèmes d'aide (touche F1)
 - si le composant sous le curseur a une aide spécifique, il l'affiche
 - sinon, il demande au panneau qui contient le composant (cases à cocher ou bouton radio)
 - si le panneau ne sait pas non plus, il demande à la fenêtre, qui demande à l'application
- Le traitement des clics de l'utilisateur

Chaîne de responsabilité : exemple

```
class Handler
{
    private static java.util.Random s_rn = new java.util.Random();
    private static int s_next = 1;
    private int m_id = s_next++;

    public boolean handle(int num)
    {
        if (s_rn.nextInt(4) != 0)
        {
            System.out.print(m_id + "-busy ");
            return false;
        }
        System.out.println(m_id + "-handled-" + num);
        return true;
    }
}

public class ChainDemo
{
    public static void main(String[] args)
    {
        Handler[] nodes =
        {
            new Handler(), new Handler(), new Handler(), new Handler()
        };
        for (int i = 1, j; i < 10; i++)
        {
            j = 0;
            while (!nodes[j].handle(i))
                j = (j + 1) % nodes.length;
        }
    }
}
```

Le client est responsable pour parcourir la liste des handlers et trouver le « bon »

Chaîne de responsabilité : exemple

```
class Handler
{
    private static java.util.Random s_rn = new java.util.Random();
    private static int s_next = 1;
    private int m_id = s_next++;
    private Handler m_next;

    public void add(Handler next) {
        if (m_next == null) m_next = next;
        else m_next.add(next);
    }
    public void wrap_around(Handler root) {
        if (m_next == null) m_next = root;
        else m_next.wrap_around(root);
    }
    public void handle(int num) {
        if (s_rn.nextInt(4) != 0) {
            System.out.print(m_id + "-busy ");
            m_next.handle(num);
        }
        else
            System.out.println(m_id + "-handled-" + num);
    }
}

public class ChainDemo {
    public static void main(String[] args) {
        Handler chain_root = new Handler();
        chain_root.add(new Handler());
        chain_root.add(new Handler());
        chain_root.add(new Handler());
        chain_root.wrap_around(chain_root);
        for (int i = 1; i < 10; i++)
            chain_root.handle(i);
    }
}
```

Une fois la chaîne initialisée,
le client a juste a soumettre
sa requête

Chaîne de responsabilité : exemple

```
public class ChainBidDemo {

    static class Link {
        private int id;
        private Link next;           // 1. "next" pointer

        private static int theBid = 999; // 4. Current bid and bidder
        private static Link bidder;

        public Link( int num ) {
            id = num;
        }
        public void addLast( Link l ) {
            if (next != null) next.addLast( l ); // 2. Handle and/or pass on
            else next = l;
        }
        public void bid() {
            int num = ((int)(Math.random() * 100)) % 9;
            System.out.print( id + "-" + num + " " );
            if (num < theBid) {
                theBid = num;           // 4. Current bid and bidder
                bidder = this;
            }
            if (next != null) next.bid(); // 2. Handle and/or pass on
            else bidder.execute(); // 5. The last 1 assigns the job
        }
        public void execute() {
            System.out.println( id + " is executing" );
            theBid = 999;
        }
    }
}
```

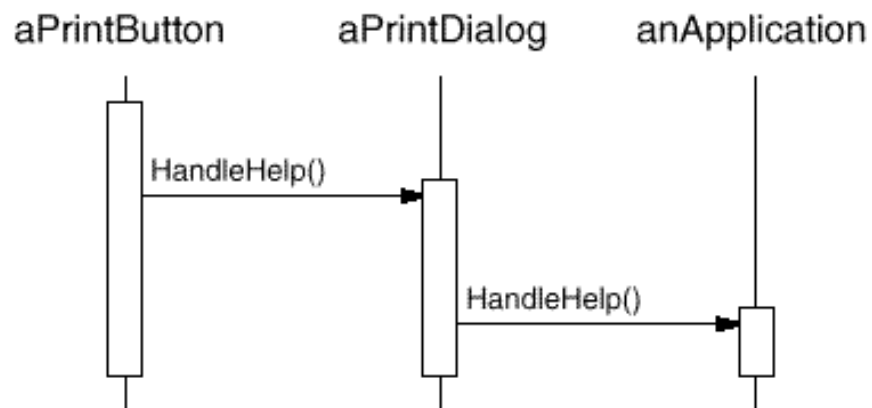
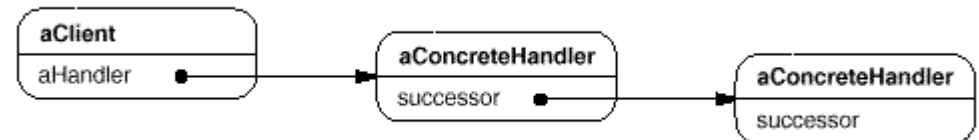
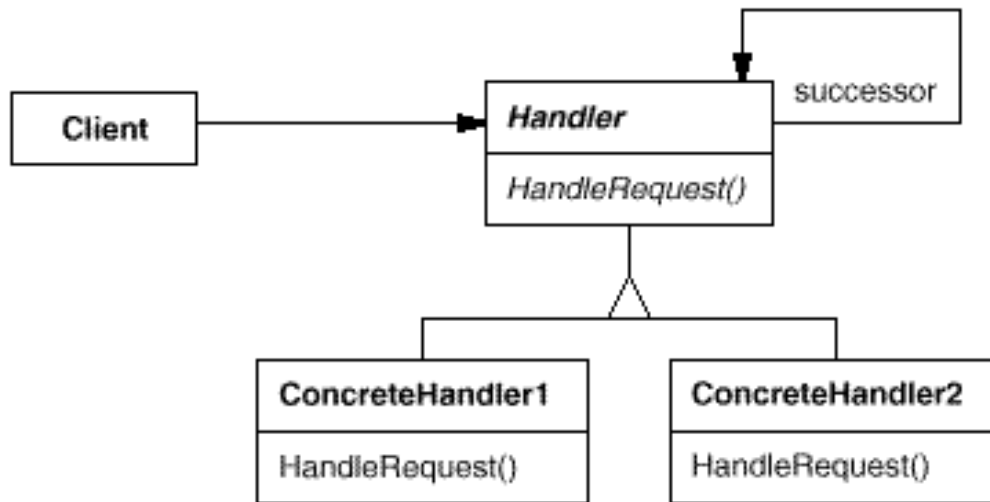
```
static Link setUpChain() {
    Link first = new Link( 1 );
    for (int i=2; i < 7; i++)
        first.addLast( new Link( i ) );
    return first;
}

public static void main( String[] args ) {
    Link chain = setUpChain(); // code bien séparé
    for (int i=0; i < 10; i++)
        // 3. Client "launches & leaves"
        chain.bid();
} }
```

Chaîne de responsabilité check-list

- La classe de base maintient un pointeur "suivant"
- Chaque classe dérivée implémente sa contribution pour traiter certaines requêtes
- Si la requête doit être transmise, la classe dérivée appelle la classe de base, qui délègue au suivant
- Le client (ou un tiers) crée et lie la chaîne
- Le client « lance » chaque requête vers la racine de la chaîne
- La délégation récursive fait le reste !

Chaîne de responsabilité



Command : quand

- On n'a pas de pointeurs de fonctions en java (sauf le vieux java de microsoft)
 - Mais on veut pouvoir passer des actions à exécuter :
 - Sur click d'un bouton
 - juste avant la sortie du programme (atexit)
- En plus, on veut pouvoir faire un système d'*undo*
- On veut permettre une exécution asynchrone et contrôlée des opérations, gérer des priorités
- Permettre des rollbacks ou pouvoir rejouer des commandes en cas de crash

Command : comment

- Déclarer une interface avec comme méthode `run()`, ou `execute()`, ou `actionPerformed()`, ou `call()`, éventuellement avec des exceptions et/ou des paramètres et/ou une valeur de retour
- Passer des implémentations de cette interface pour passer une fonction
- Si on veut pouvoir annuler l'action (undo), prévoir une méthode `execute()` et une méthode `unexecute()/undo()`
- S'il est nécessaire de stocker un état pour l'undo, passer aux méthodes un objet `Context`, plutôt que de le stocker dans l'objet `Command` (un objet = une responsabilité)

Command : exemple

```
public interface Command {  
    void do(Context context);  
    void undo(Context context);  
}
```

```
public class InsertCharacter implements Command {  
    private final char c;  
    private final Document doc;  
    public InsertCharacter(Document doc, char c) { ... }  
    void do(Context context) {  
        int pos = doc.getCarretPosition();  
        doc.appendAfter(pos,c);  
    }  
    void undo(Context context) {  
        int pos = doc.getCarretPosition();  
        doc.deleteChar(pos-1);  
    }  
}
```

Command : exemple

```
public interface Command {  
    void do(Context context);  
    void undo(Context context);  
}
```

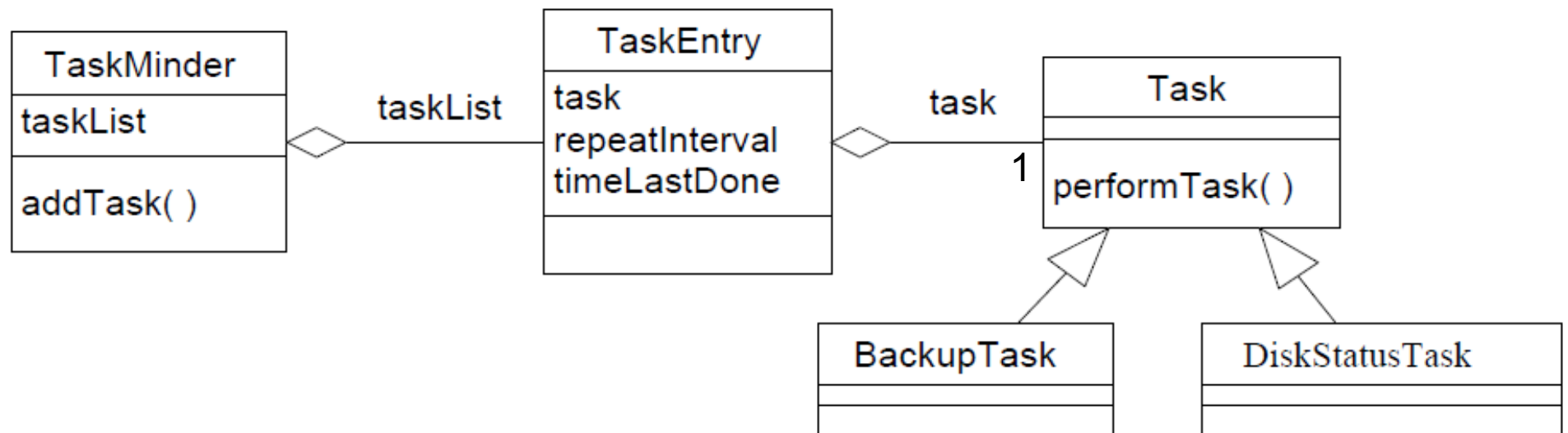
```
public class ReplaceAll implements Command {  
    private String from;  
    private String to;  
    private final Document doc;  
  
    public ReplaceAll(Document doc, String from, String to) {...}  
  
    void do(Context context) {  
        int[] positions = doc.findAll(from);  
        context.save(positions);  
        for(int i=positions.length-1;pos>=0;pos--)  
            doc.replace(positions[i],from.length,to);  
    }  
    void undo(Context context) {  
        int[] positions = (int[])context.restore();  
        for(int pos: positions)  
            doc.replace(pos,to.length,from);  
    }  
}
```

Command : exemple

```
public class Commander {  
    private final ArrayDeque<Command> commands = new ...;  
    private final ArrayDeque<Context> contexts = new ...;  
    private int depth = INIT_DEPTH;  
    public void apply(Command command) {  
        if (commands.size()==depth) {  
            commands.removeFirst();  
            contexts.removeFirst();  
        }  
        Context context = new Context();  
        // handle RuntimeException  
        command.do(context);  
        commands.addLast(command);  
        contexts.addLast(context);  
    }  
    public void undo() {  
        Context context = contexts.removeLast();  
        commands.removeLast().undo(context);  
    }  
}
```

Command : exemple

- Un scheduler



Scheduler (2)

```
public interface Task {  
    public void performTask();  
}
```

```
public class FortuneTask implements Task {  
    int nextFortune = 0;  
    String[] fortunes = {"She who studies hard, gets A",  
        "Seeth the pattern and knoweth the truth",  
        "He who leaves state the day after final, graduates not" };  
    public FortuneTask() {}  
    public void performTask() {  
        System.out.println("Your fortune is: " +  
            fortunes[nextFortune]);  
        nextFortune = (nextFortune + 1) % fortunes.length;  
    }  
    public String toString() {return ("Fortune Telling Task");}  
}
```

```
public class FibonacciTask implements Task {  
    int n1 = 1;  
    int n2 = 0;  
    int num;  
    public FibonacciTask() {}  
    public void performTask() {  
        num = n1 + n2;  
        System.out.println("Next Fibonacci number is: " + num);  
        n1 = n2;  
        n2 = num;  
    }  
    public String toString() {return ("Fibonacci Sequence Task");}  
}
```

Scheduler (3)

```
public class TaskEntry {
    private Task task; // The task to be done
    // It's a Command object!

    private long repeatInterval; // How often task should be executed
    private long timeLastDone; // Time task was last done
    public TaskEntry(Task task, long repeatInterval) {
        this.task = task;
        this.repeatInterval = repeatInterval;
        this.timeLastDone = System.currentTimeMillis();
    }
    public Task getTask() {return task;}
    public void setTask(Task task) {this.task = task;}
    public long getRepeatInterval() {return repeatInterval;}
    public void setRepeatInterval(long ri) {
        this.repeatInterval = ri;
    }
    public long getTimeLastDone() {return timeLastDone;}
    public void setTimeLastDone(long t) {this.timeLastDone = t;}
    public String toString() {
        return (task + " to be done every " + repeatInterval +
            " ms; last done " + timeLastDone);
    }
}
```

Scheduler (4)

```
public class TaskMinder extends Thread {
    private long sleepInterval; // How often the TaskMinder should
    private List<TaskEntry> taskList; // The list of tasks

    public TaskMinder(long sleepInterval) {
        this.sleepInterval = sleepInterval;
        taskList = new ArrayList<TaskEntry>();
        start();
    }

    public void addTask(Task task, long repeatInterval) {
        long ri = (repeatInterval > 0) ? repeatInterval : 0;
        TaskEntry te = new TaskEntry(task, ri);
        taskList.add(te);
    }

    public long getSleepInterval() {
        return sleepInterval;
    }

    public void setSleepInterval(long si) {
        this.sleepInterval = si;
    }

    public void run() {
        while (true) {
            try {
                sleep(sleepInterval);
                long now = System.currentTimeMillis();
                for (TaskEntry te : taskList) {
                    if (te.getRepeatInterval() + te.getTimeLastDone() < now) {
                        te.getTask().performTask();
                        te.setTimeLastDone(now);
                    }
                }
            } catch (Exception e) {
                System.out.println("Interrupted sleep: " + e);
            }
        }
    }
}
```

Scheduler (5)

```
package fr.umlv.poo.scheduler;

public class TestTaskMinder {
    public static void main(String args[]) {
        // Create and start a TaskMinder.
        // This TaskMinder checks for things to do every 500 ms
        TaskMinder tm = new TaskMinder(500);

        // Create a Fortune Teller Task.
        FortuneTask fortuneTask = new FortuneTask();
        // Have the Fortune Teller execute every 3 seconds.
        tm.addTask(fortuneTask, 3000);

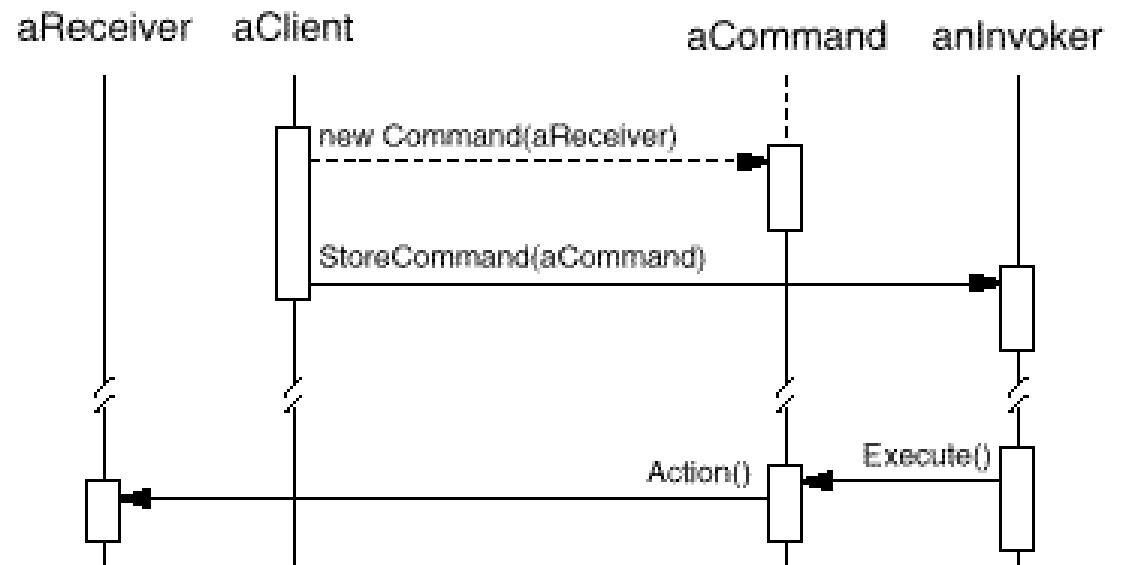
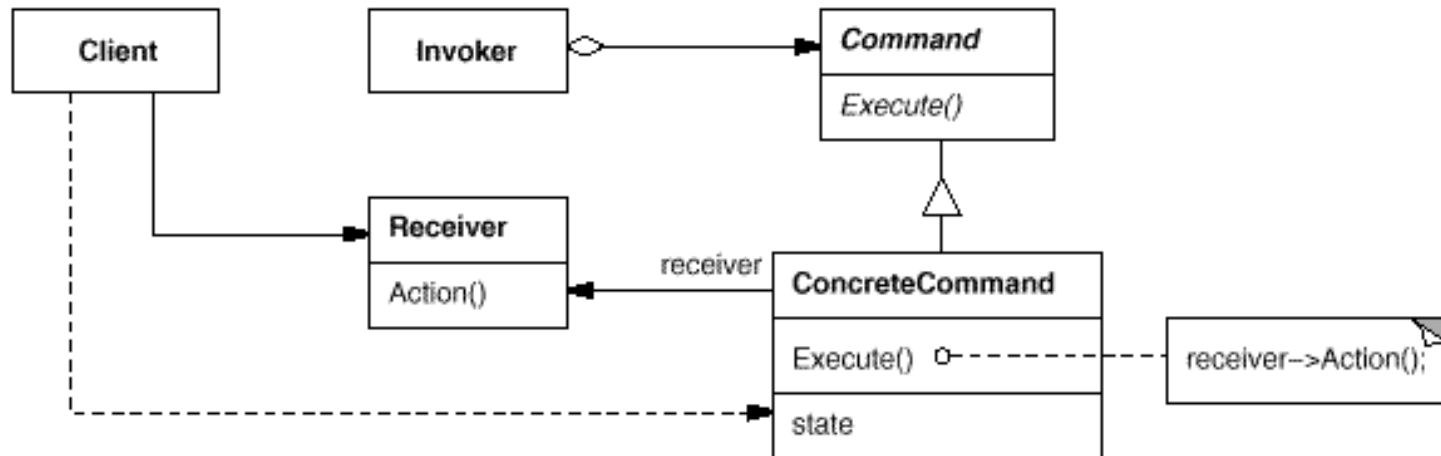
        // Create a Fibonacci Sequence Task.
        FibonacciTask fibonacciTask = new FibonacciTask();
        // Have the Fibonacci Sequence execute every 700
        // milliseconds.
        tm.addTask(fibonacciTask, 700);
    }
}
```

```
Next Fibonacci number is: 1
Next Fibonacci number is: 1
Your fortune is: She who studies hard, gets A
Next Fibonacci number is: 2
Next Fibonacci number is: 3
Next Fibonacci number is: 5
Next Fibonacci number is: 8
Your fortune is: Seeth the pattern and knoweth the truth
Next Fibonacci number is: 13
Next Fibonacci number is: 21
Next Fibonacci number is: 34
Your fortune is: He who leaves state the day after final,
graduates not
Next Fibonacci number is: 55
Next Fibonacci number is: 89
Next Fibonacci number is: 144
```

Command : check-list

- Définir une interface Command avec une méthode execute() (ou similaire)
- Créer une ou plusieurs classes dérivées qui encapsulent : un "receiver", la méthode à invoquer , les arguments
- Instancier un objet Command pour chaque requête d'exécution différée
- Transmettre cet objet Command du créateur (sender/client) à l' « invoqueur ».
- L'invoqueur décide quand appeler execute().

Command



Observer : quand

- Je veux savoir quand un objet A change
 - mettre à jour une vue graphique
 - programmer la sauvegarde des changements sur le disque
- Mais sans que ça soit la responsabilité de l'objet A
- Et sans que l'objet A ne me connaisse ni ne connaisse mon interface

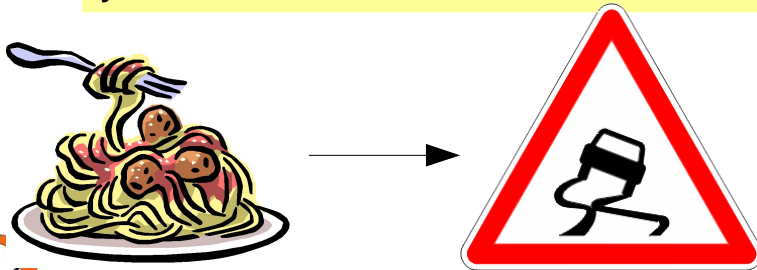
Observer: pourquoi

// les objets dépendants sont divers et leur mise à jour est codé « en dur », de manière static

```
class DivObserver {  
    private int m_div;  
    public DivObserver( int div ) { m_div = div; }  
    public void update( int val ) { ...}  
}  
class ModObserver {  
    private int m_mod;  
    public ModObserver( int mod ) { m_mod = mod; }  
    public void maj( int val, String msg ) {...}  
}  
class Subject {  
    int m_value;  
    public Subject() {... }  
    void set_value( int value ) {  
        m_value = value;  
    }  
}
```

- Connaissance des observers par les clients
- Connaissance des actions que réalisent les observers
- Dépendances statiques
- redondances de code !

```
int main( void ) {  
    DivObserver div_obj = new DivObserver(0);  
    ModObserver mod_obj = new ModObserver(0);  
    ...  
    Subject subj = new Subject();  
    subj.set_value( 14 );  
    div_obj.update( 14 );  
    mod_obj.maj( 14, « ... » );  
}
```



Mieux ?

// les objets dépendants sont divers et leur mise à jour est codé « en dur », de manière static

```
class DivObserver {
    private int m_div;
    public DivObserver( int div ) { m_div = div; }
    public void update( int val ) { ...}
}
class ModObserver {
    private int m_mod;
    public ModObserver( int mod ) { m_mod = mod; }
    public void maj( int val, String msg ) {...}
}
class Subject {
    int m_value;
    DivObserver m_div_obj = new DivObserver(0);
    ModObserver m_mod_obj = new ModObserver(0);
    public Subject() {...}
    void set_value( int value ) {
        m_value = value;
        notify();
    }
    void notify() {
        m_div_obj.update( m_value );
        m_mod_obj.maj( m_value, « ... » );
    }
}
```

```
int main( void ) {
    DivObserver div_obj = new DivObserver(0);
    ModObserver mod_obj = new ModObserver(0);
    ...
    Subject subj = new Subject();
    subj.set_value( 14 );
}
```

- Connaissance des observers
- Connaissance des actions que réalisent les observers
- Dépendances statiques

Observer : comment

- On créer une/des interfaces Observer/Listener avec autant de méthodes que de caractéristiques à surveiller
- On répartit les méthodes dans les interfaces, de manière logique
- Les observateurs qui veulent être prévenus des changements s'enregistrent sur l'observable : ils lui fournissent une implémentation d'un Observer/Listener.
- L'observable, dès que son état change, appelle la méthode correspondante des observateurs (listeners) enregistrés

Observer : comment

- L'information sur les changements est donnée par le choix de la méthode, et éventuellement son argument
- On ne précise dans l'argument que ce que l'observateur ne peut pas obtenir
 - si on attend le changement de *Property*, la nouvelle valeur n'a pas besoin d'être transmise, car il suffit d'appeler *getProperty* (sauf certains cas)
 - on peut en revanche fournir l'ancienne valeur, ou pour une liste dire quels indices ont changés
 - Dans certains cas, on fournit plus d'information pour des raisons d'optimisation
- Les observateurs doivent être prévenus **après** le changement effectué dans l'objet

Observer : exemple

```
interface Observer {
    public void update( int value );
}

class Subject {
    int m_value;
    List<Observer> m_views;
    public void attach( Observer obs ) { m_views.add( obs ); }
    public void detach( Observer obs ) { m_views.remove( obs ); }
    void set_val( int value ) { m_value = value; notify(); }
    void notify() {for (Observer o : m_views) o.update( m_value ); }
}

class DivObserver implements Observer {
    int m_div;
    public DivObserver( Subject model, int div ) {
        model.attach( this );
        m_div = div;
    }
    void update( int v ) { ... }
}

class ModObserver implements Observer {
    int m_mod;
    public ModObserver( Subject model, int mod ) {
        model.attach( this );
        m_mod = mod;
    }
    void update( int v ) {...}
}
```

- couplage faible
- Connaissance uniquement des abstractions
- aucune connaissance des actions que réalisent les observers
- pas de dépendances statiques

```
int main( void ) {
    Subject    subj = new Subject();
    DivObserver divObs1( subj, 4 );
    DivObserver divObs2( subj, 3 );
    ModObserver modObs3( subj, 3 );
    ...
    subj.set_val( 14 );
}
```

Observer : exemple avec Event

```
public interface ParkingListener {  
    void carEntered(ParkingEvent event);  
    void carLeaved(ParkingEvent event);  
}
```

```
public class ParkingEvent {  
    public int getNumber();  
    ...  
}
```

```
public class Parking {  
    public void addParkingListener(ParkingListener listener) {  
        listeners.add(listener);  
    }  
    public void removeParkingListener(ParkingListener listener) { ... }  
    protected void fireCarEntered(int number) {  
        if (listeners.isEmpty())  
            return;  
        ParkingEvent event = new ParkingEvent(number);  
        for(ParkingListener listener : listeners)  
            listener.carEntered(event);  
    }  
    protected void fireCarLeaved(int number) { ... }  
    public void enter(Car car) {  
        int position = findFreePosition();  
        cars[position] = car;  
        fireCarEntered(position);  
    }  
}
```

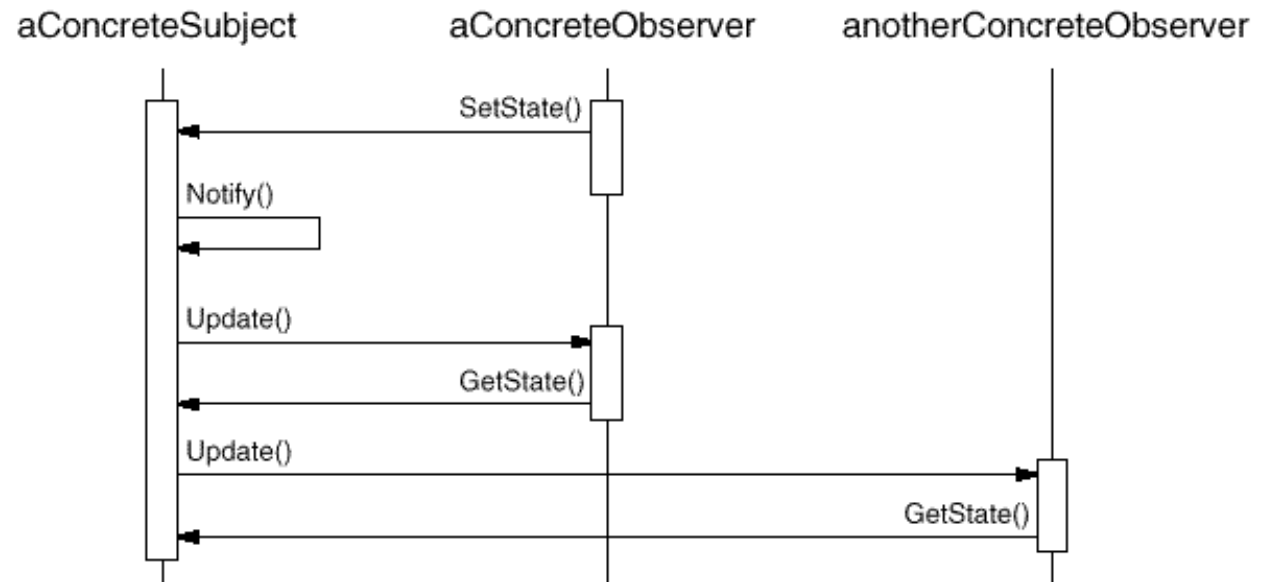
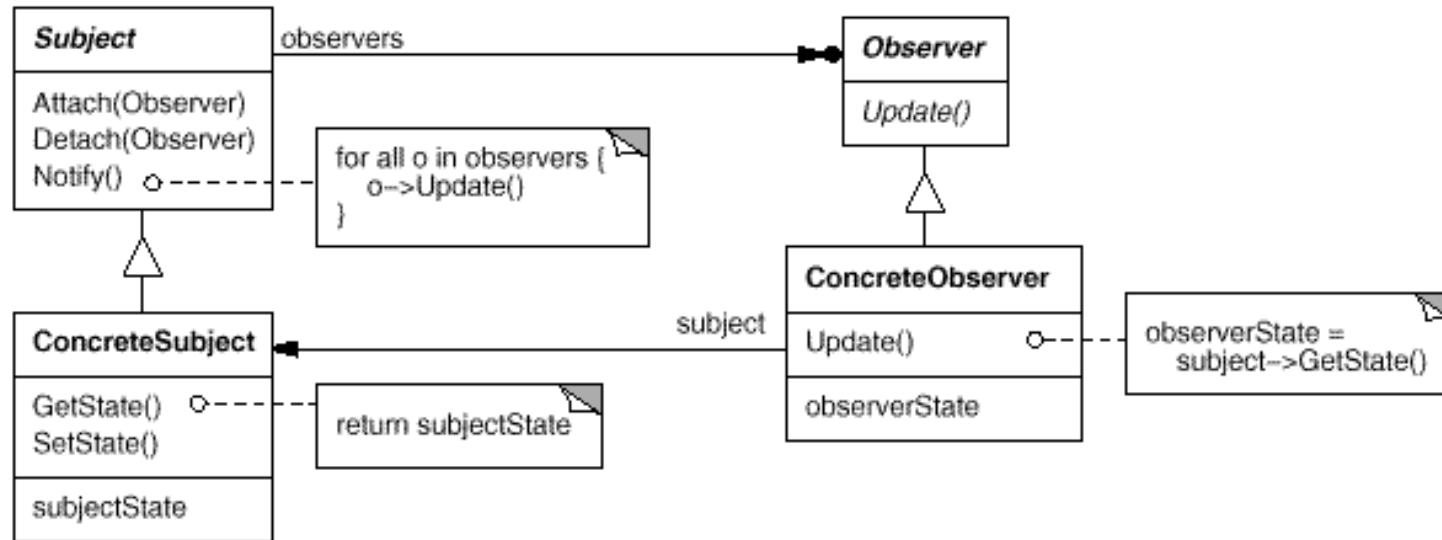
Observer : exemple

- Les modèles de swing :
 - TreeModel/TreeModelListener, ListModel/ListDataListener, TableModel/TableModelListener
 - les MouseListener, KeyListener sont juste des listeners/handlers
- Les java beans
 - PropertyChangeListener

Observer : check-list

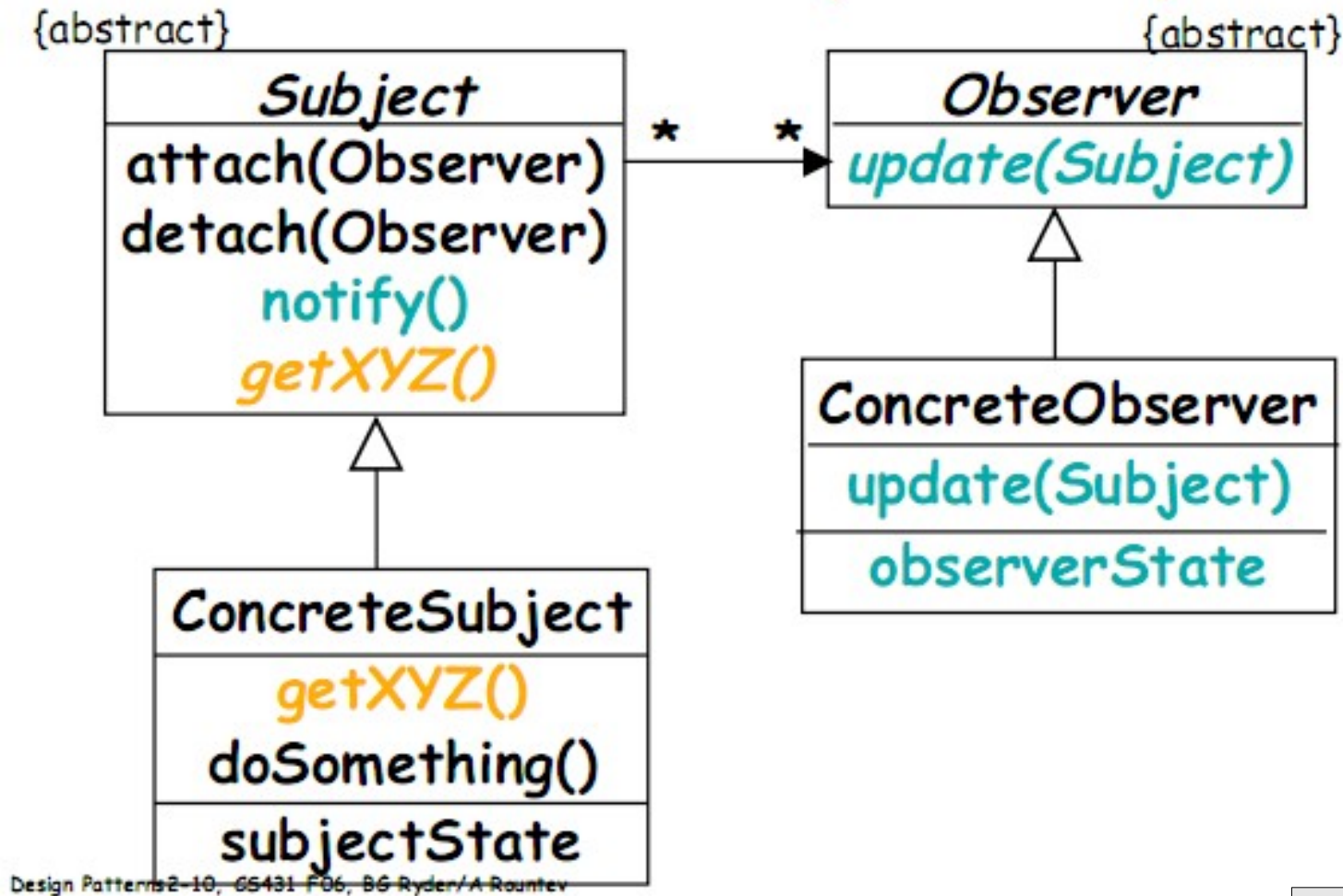
1. Différencier entre la fonctionnalité « core » et « indépendante »
2. Concevoir la fonctionnalité «core» avec une abstraction "subject".
3. Concevoir la fonctionnalité «indépendante» avec une hiérarchie "observer".
4. Le Subject n'est couplé qu'à la classe de base Observer
5. Le client configure le nombre et le type des Observers.
6. Les Observers s'enregistrent eux-même auprès du Subject.
7. Le Subject diffuse des événements à tous les Observers enregistrés.
8. Le Subject peut "diffuser" des information aux Observers, ou les Observers peuvent demander les information au Subject (push ou pull)

Observer



Observer (modified)

GoF Formulation (Modified)



18

© Ryder

Java.util.Observable/Observer

- Super, built-in dans le JDK !
- Observable est une classe, Observer une interface

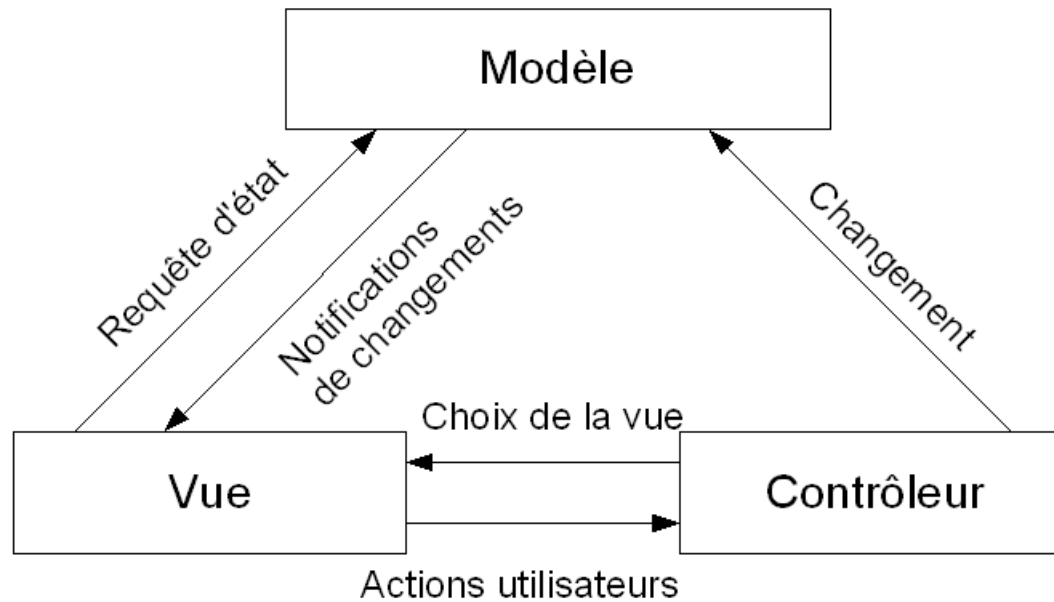
- ?

```
class Observable { // subject
    public void addObserver(Observer o) {...}
    public void deleteObserver(Observer o) {...}
    public void deleteObservers() {...}
    public int countObservers() {...}
    public void notifyObservers()
        { notifyObservers(null); }
    public void notifyObservers(Object arg) {...}
}
interface Observer {
    public void update(Observable o, Object arg);
}
```

Observer & MVC

- Le design pattern Modèle-Vue-Contrôleur (MVC) sépare les données (le modèle), l'interface homme-machine (la vue) et la logique de contrôle (le contrôleur).
- Ce modèle de conception impose donc une séparation en 3 couches :
 - Le modèle : Il représente les données de l'application. Il définit aussi l'interaction avec la base de données et le traitement de ces données.
 - La vue : Elle représente l'interface utilisateur, ce avec quoi il interagit. Elle n'effectue aucun traitement, elle se contente simplement d'afficher les données que lui fournit le modèle. Il peut y avoir plusieurs vues qui présentent les données d'un même modèle.
 - Le contrôleur : Il gère l'interface entre le modèle et le client. Il va interpréter la requête de ce dernier pour lui envoyer la vue correspondante. Il effectue la synchronisation entre le modèle et les vues.
- La synchronisation entre la vue et le modèle passe par le pattern Observer. Il permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour.

Observer & MVC



State : quand

- L'exécution dépend d'un état, qui change à l'exécution :
 - ```
public void read(byte[] buffer) throws IOException {
 if (state == State.CLOSED)
 throw new IOException("Stream is closed");
 if (state == State.NEW)
 throw new IOException("Stream is not opened");
 if (state == State.ERROR)
 throw new IOException("Error state",ioException);
 if (!modes.contains(Mode.READ))
 throw new IOException("Stream is not readable");
 if (position != localBuffer.length)
 copyRemainingToBuffer(buffer); return;
 else read(); copyRemainingToBuffer(buffer); return;
}
```
- Autre repère: d'importantes parties conditionnelles dépendant de l'état de l'objet

# State : comment

- On utilise POO !
  - si l'objet qui implémente le « protocole » à état est A
  - l'état est codé par un objet B
  - B possède une méthode pour chaque méthode de A qui réalise le «protocole» (délégation)
  - Connaissance du contexte: B a souvent besoin de connaître A (le contexte). deux choix possibles :
    - les méthodes de B reçoivent l'objet A en paramètre
    - les instances de B sauvent la référence vers l'objet A
  - Changement d'état:
    - B ou A se charge de changer l'état de A
    - la classe B est toujours invisible hors du paquetage (setState de paquetage)
  - Les états: on peut utiliser un singleton (ou enum en Java) si B ne conserve pas la référence sur A

# State : exemple

- Automates finis
- Protocoles, entrée-sortie
- Interface graphique :
  - l'état correspond à l'outil sélectionnée
  - il possède des méthodes qui traitent les événements souris
- Les classes/enum internes sont très adaptées à State

# State : exemple

```
public class Stream {
 private State state = init;
 final State init = new InitState(this);
 final State read = new ReadState(this);
 final State write = new WriteState(this);
 final State readWrite = new ReadWriteState(this);
 final State closed = new ClosedState(this);
 final State error = new ErrorState(this);
 setState(State state) { this.state = state }
 public void open(Input input, Mode mode) throws IOException {
 state.open(input,mode);
 }
 public void read(byte[] buffer) throws IOException {
 state.read(buffer,offset,length);
 }
 public void write(byte[] buffer) throws IOException {
 state.read(buffer,offset,length);
 }
 public void close() throws IOException {
 state.close();
 }
}
```

# State : exemple

```
abstract class State {
 protected final Stream stream;
 State(Stream stream) { this.stream = stream; }
 abstract void open(Input input, Mode mode) throws IOException;
 abstract void read(byte[] buffer) throws IOException;
 abstract void write(byte[] buffer) throws IOException;
 abstract void close() throws IOException;
}
```

```
abstract class ReadWriteState extends State {
 ReadWriteState(Stream stream) { super(stream); }
 void open(Input input, Mode mode) throws IOException {
 throw new IOException("Already opened");
 }
 void read(byte[] buffer) throws IOException {
 throw new IOException("Write only stream");
 }
 void write(byte[] buffer) throws IOException {
 throw new IOException("Read only stream");
 }
 void close() throws IOException {
 ...
 stream.setState(stream.closed);
 }
}
```

# State : exemple

```
abstract class State {
 protected final Stream stream;
 State(Stream stream) { this.stream = stream; }
 abstract void open(Input input, Mode mode) throws IOException;
 abstract void read(byte[] buffer) throws IOException;
 abstract void write(byte[] buffer) throws IOException;
 abstract void close() throws IOException;
}
```

```
class ReadState extends ReadWriteState {
 ReadState(Stream stream) { super(stream); }
 void read(byte[] buffer) throws IOException {
 ... stream.get...
 }
}
```

```
class WriteState extends ReadWriteState {
 WriteState(Stream stream) { super(stream); }
 void write(byte[] buffer) throws IOException {
 ... stream.get...
 }
}
```

# State : exemple

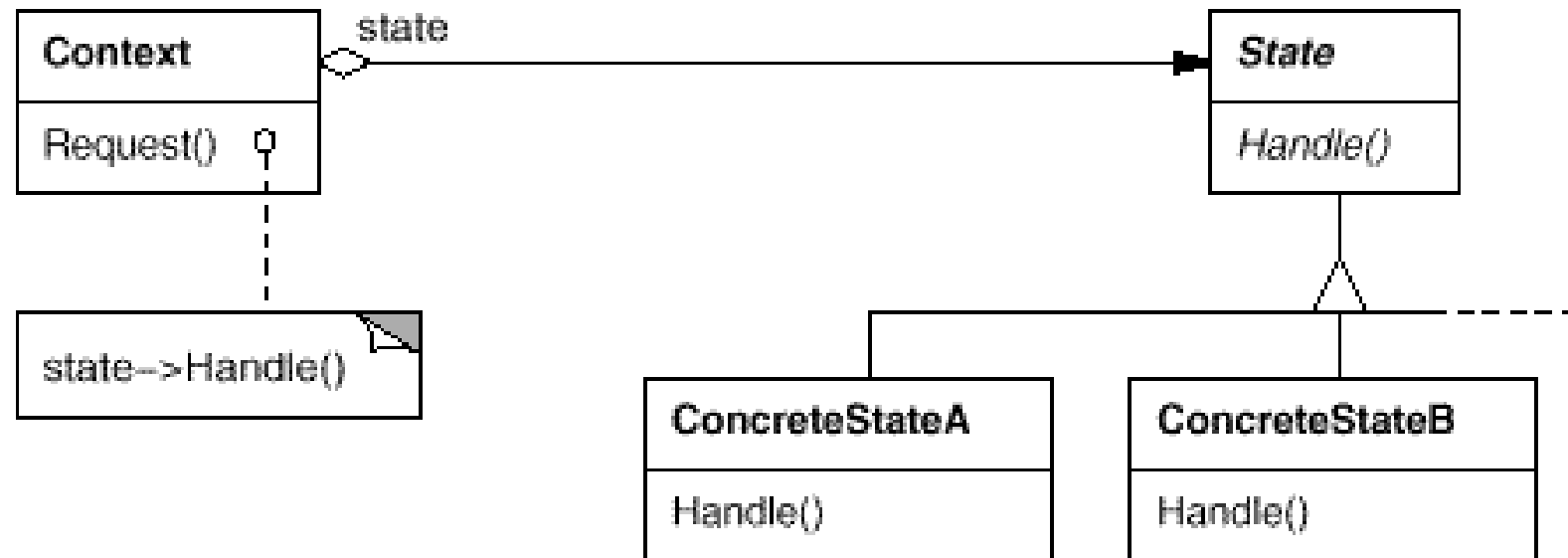
```
abstract class State {
 protected final Stream stream;
 State(Stream stream) { this.stream = stream; }
 abstract void open(Input input, Mode mode) throws IOException;
 abstract void read(byte[] buffer) throws IOException;
 abstract void write(byte[] buffer) throws IOException;
 abstract void close() throws IOException;
}

class OpenState extends State {
 OpenState(Stream stream) { super(stream); }
 void open(Input input, Mode mode) throws IOException {
 switch(mode) {
 case READ : stream.setState(stream.read);
 case WRITE : stream.setState(stream.write);
 case READWRITE : stream.setState(stream.readWrite);
 default : throw new AssertionError("No Such Mode");
 }
 stream.setInput(input); ...
 }
 void read(byte[] buffer) throws IOException {
 throw new IOException("Not yet opened");
 }
 ...
}
```

# State : check-list

- Identifier (ou créer) la classe qui servira de « machine à état », du point de vue du client (le wrapper/contexte)
- Créer une classe de base, qui réplique les méthodes de la « - machine à état ». Chaque méthode prend un argument supplémentaire : une instance du wrapper/contexte (ou juste dans le constructeur). La classe State peut spécifier des comportements par défaut.
- Créer une classe dérivée de State pour chaque état. Ces classes redéfinissent uniquement le nécessaire.
- La classe maintient un State "courant"
- Toutes les requêtes au wrapper sont déléguées au State courant.
- Le changement de State courant se fait dans le wrapper/contexte OU dans les State's

# State



# Strategy : quand

- On veut choisir l'algorithme pour faire un certain travail
- Ce choix :
  - intervient à l'exécution,
  - peut être différent en fonction des clients,
  - peut être changé dynamiquement
- Les clients ne veulent pas/ ne doivent pas connaître les détails d'implémentation des différents algorithmes

# Strategy : quand

- On veut choisir l'algorithme pour faire un certain travail
- Ce choix :
  - intervient à l'exécution,
  - peut être différent en fonction des clients,
  - peut être changé dynamiquement
- Les clients ne veulent pas/ ne doivent pas connaître les détails d'implémentation des différents algorithmes

# Strategy : comment

- On utilise une classe abstraite ou une interface pour effectuer les actions de l'algorithme
- Quelles méthodes y mettre :
  - ce dont ont besoin les clients (de l'algo)
  - ce dont ont besoin tous les algorithmes
- En général, pour respecter l'encapsulation, une interface ou une classe abstraite dont les méthodes sont de paquetage
- Le client peut choisir parmi les différents algorithmes en fournissant une des implémentations proposées par le paquetage (utiliser un singleton)

# Strategy : exemple

```
Interface SortStrategy {
 public void sort(Array a);
}
```

```
public class BubbleSort implements SortStrategy {
 public void sort(Array a) {
 ...
 }
 ...
}
```

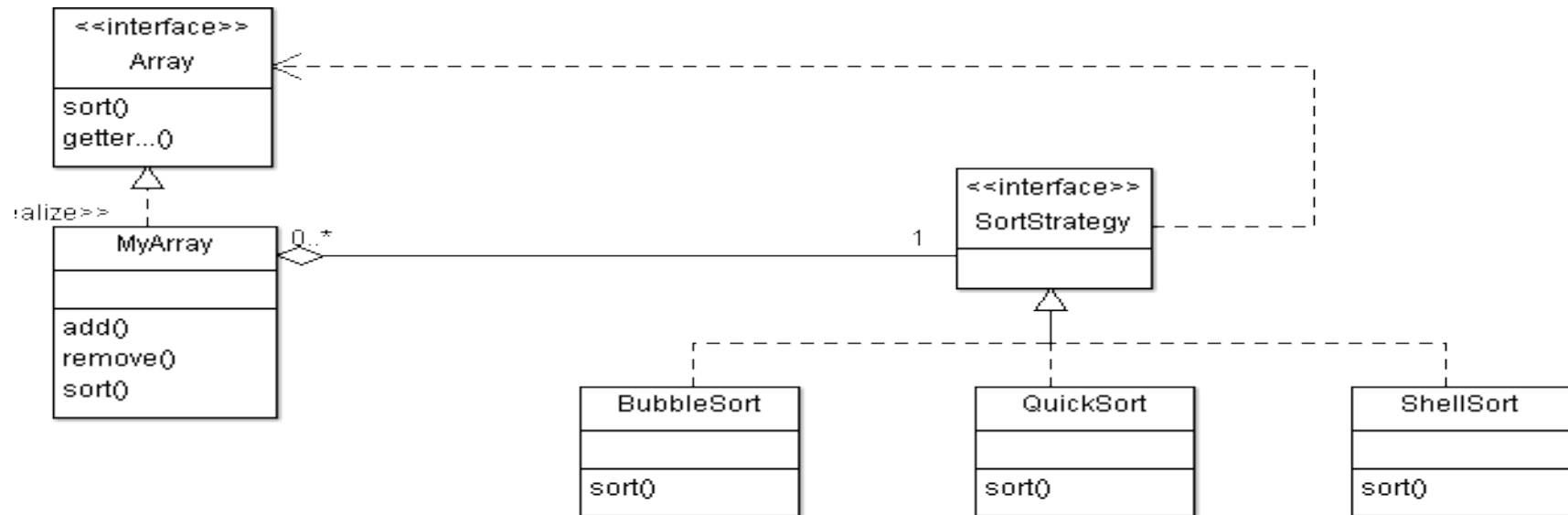
```
public class QuickSort implements SortStrategy {
 public void sort(Array a) {
 ...
 }
 ...
}
```

```
public class SortArray implements Array {
 private SortStrategy sortAlgo;
 public void sort() {
 sortAlgo.sort(this);
 }

 private final bubble=new BubbleSort();
 private final qs=new QuickSort();

 void add() {
 ...
 if (size() > 10) sortAlgo=qs;
 else sortAlgo=bubble;
 }
}
```

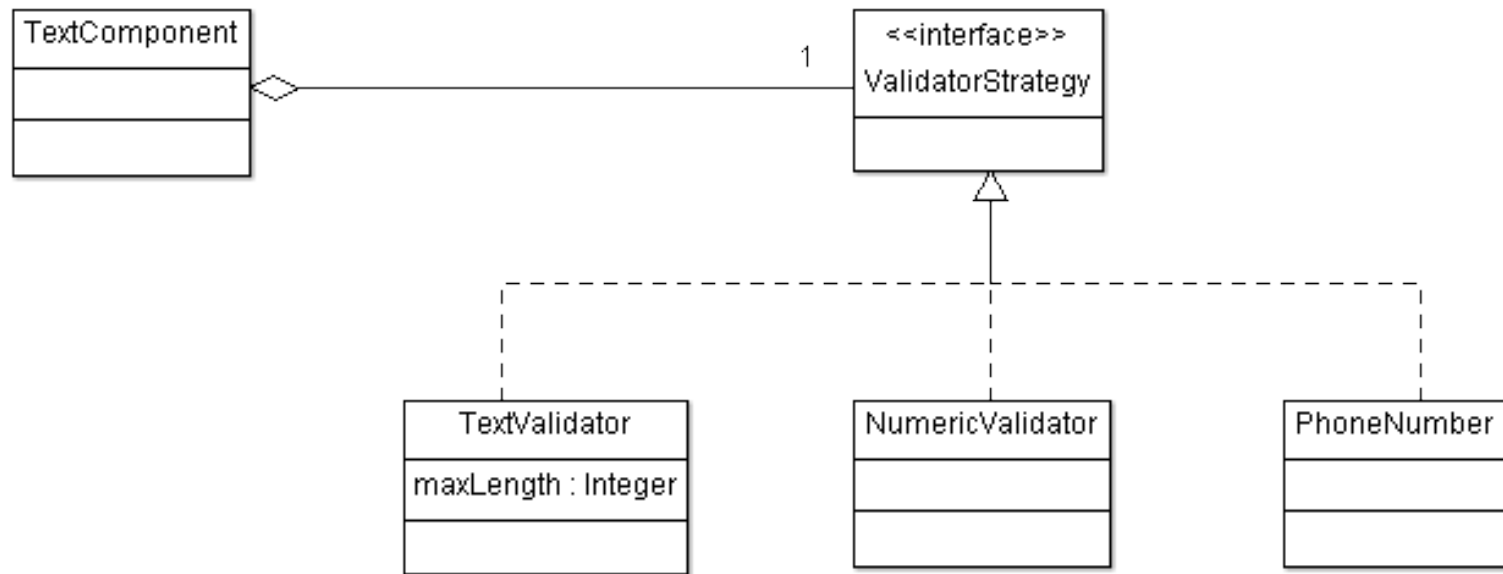
# Strategy : exemple



# Strategy : exemple

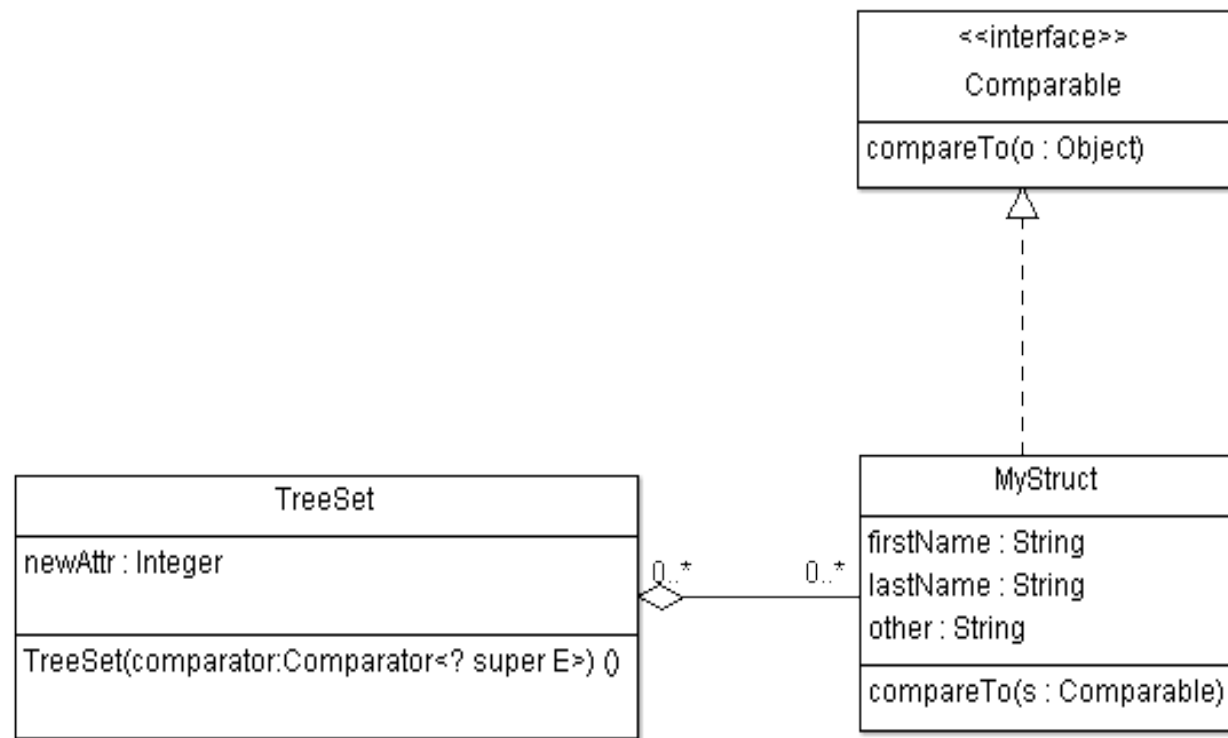
- On construit une IHM dynamiquement d'après une description dans un fichier XML.
- Il y a plusieurs types de validation de l'entrée utilisateur (texte/longueur, numérique, numéro de téléphone, ...)
- On implémente le « validateur » de la saisie utilisateur avec le pattern Strategy

# Strategy : exemple



# Strategy : exemple TreeSet java

- Une structure avec 3 champs
- UN ordre naturel
- Le TreeSet est ordonné d'après cet ordre

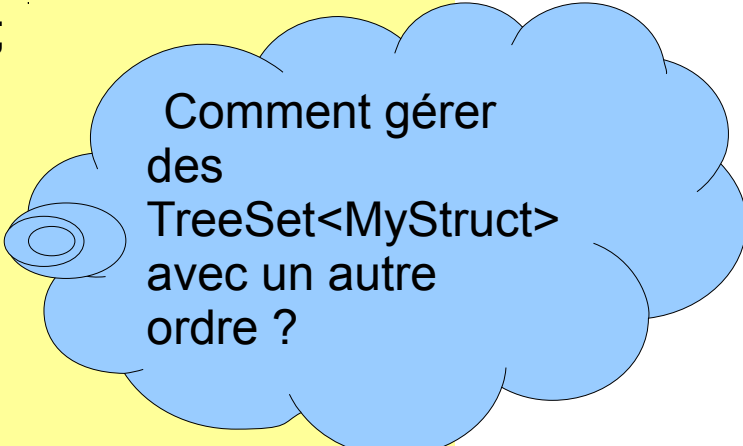


# Strategy : exemple TreeSet java

```
class MyStruct implements Comparable {
 public String firstName;
 public String lastName;
 public String other;
 public Struct(String f, String l, String o) { firstName=f; lastName=l; other=o; }
 public int compareTo(Object o) {
 return firstName.compareTo(((MyStruct)o).firstName);
 }
}
```

```
public class StrategyDemo {
 public static void main(String[] args) {
 TreeSet<MyStruct> one = new TreeSet<MyStruct>();
 one.add(new MyStruct("aaa", "cc", "b"));
 one.add(new MyStruct("ddd", "ee", "d"));
 one.add(new MyStruct("bbb", "aa", "e"));
 one.add(new MyStruct("eee", "dd", "a"));
 one.add(new MyStruct("ccc", "bb", "c"));
 for (MyStruct s : one)
 System.out.print(s.firstName + " ");
 System.out.println();
 } }

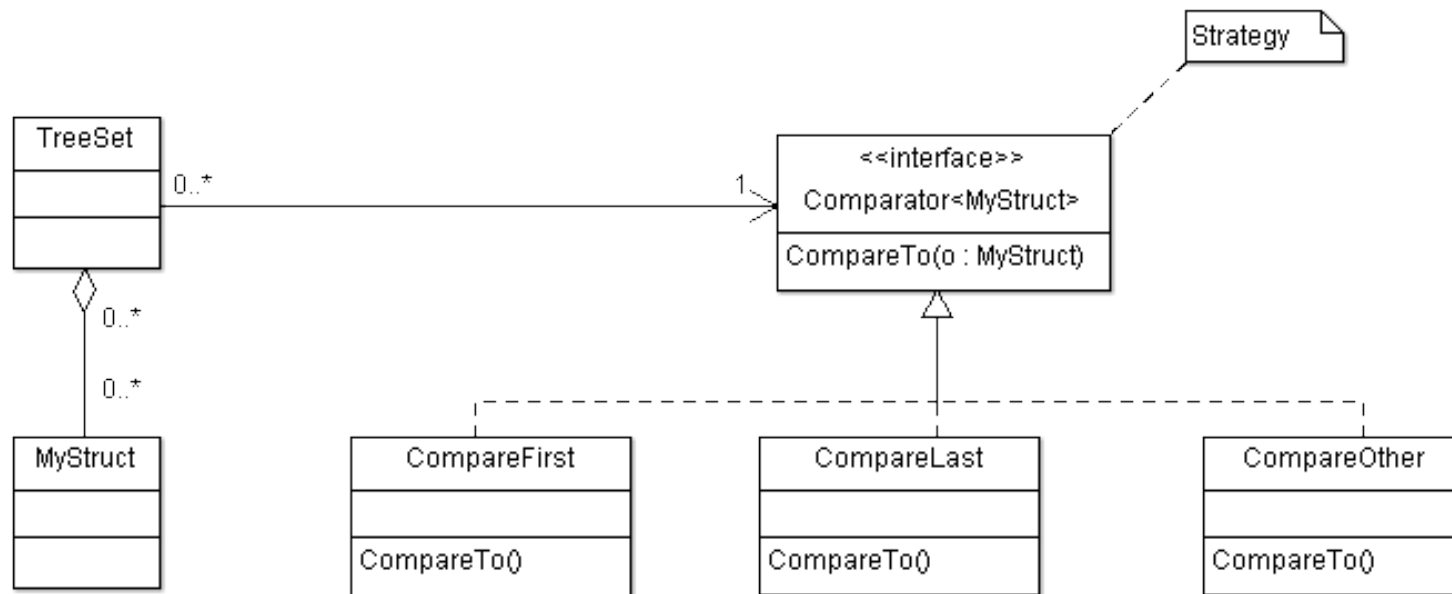
// aaa bbb ccc ddd eee
```



Comment gérer  
des  
TreeSet<MyStruct>  
avec un autre  
ordre ?

# Strategy : TreeSet java (2)

- Solution : pattern Strategy !



# Strategy : TreeSet java (2)

```
public class StrategyDemo {
 public static void main(String[] args) {
 TreeSet<MyStruct> ts1 = new TreeSet<MyStruct>(new CompareFirst());
 ts1.add(new MyStruct("aaa", "cc", "b"));
 ts1.add(new MyStruct("ddd", "ee", "d"));
 ts1.add(new MyStruct("bbb", "aa", "e"));
 ts1.add(new MyStruct("eee", "dd", "a"));
 ts1.add(new MyStruct("ccc", "bb", "c"));
 for (MyStruct s : ts1)
 System.out.print(s.firstName + " ");
 System.out.println();

 TreeSet<MyStruct> ts2 = new TreeSet<MyStruct>(new CompareLast());
 ...

 TreeSet<MyStruct> ts3 = new TreeSet<MyStruct>(new CompareOther());
 ...
 }

 // aaa bbb ccc ddd eee <= CompareFirst, display firstName's
 // aa bb cc dd ee <= CompareLast, display lastName's
 // a b c d e <= CompareOther, display other's
}
```

# Strategy : TreeSet java (2)

```
class MyStruct {
 String firstName;
 String lastName;
 String other;
 public Struct(String f, String l, String o) { firstName=f; lastName=l; other=o; }
}

class CompareFirst implements Comparator {
 public int compare(MyStruct x, MyStruct y) {
 return x.firstName.compareTo(y.firstName);
 }
}

class CompareLast implements Comparator {
 public int compare(MyStruct x, MyStruct y) {
 return x.lastName.compareTo(y.lastName);
 }
}

class CompareOther implements Comparator {
 public int compare(MyStruct x, MyStruct y) {
 return x.otherName.compareTo(y.otherName);
 }
}
```

# Strategy + template method

```
// 1. Define the interface of the algorithm
interface Strategy {
 public void solve();
}
```

```
class ConcreteStrategy1 implements Strategy
 public void solve() { // simple impl
 ...
 }
}
```

```
abstract class AbsStrategy2 implements Strategy {
 public void solve() { // Template Method
 {
 step3();
 step4();
 step5();
 }
 protected abstract void step3();
 protected abstract void step4();
 protected abstract void step5();
}

class ConcreteStrategy4 extends AbsStrategy2 {
 protected void step3() { ... }
```

```
abstract class AbsStrategy1 implements Strategy {
 public void solve() { // Template Method
 step1();
 step2();
 }
 protected abstract void step1();
 protected abstract void step2();
}

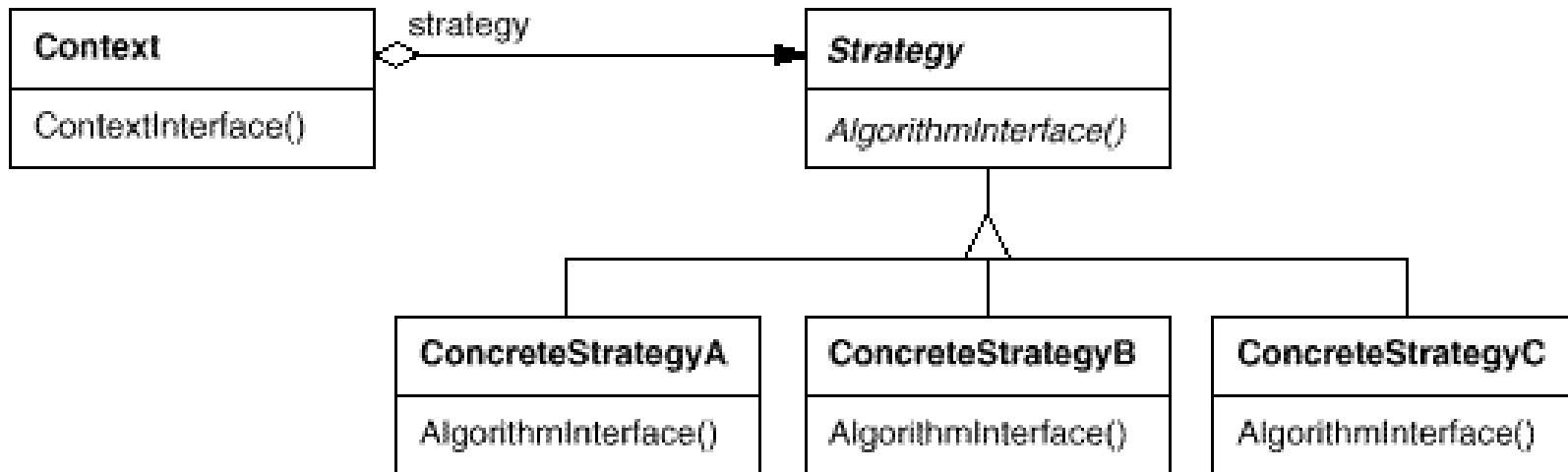
class ConcreteStrategy2 extends AbsStrategy1 {
 protected void step1() { ... }
 protected void step2() { ... }
}

class ConcreteStrategy3 extends AbsStrategy1 {
 protected void step1() { ... }
 protected void step2() { ... }
}
```

# Strategy: check-list

- Identifier un algorithme que le client a besoin d'utiliser de manière « flexible »
- Spécifier la signature pour cette algorithme dans une interface.
- « Enterrer » les différentes implémentations dans les classes dérivées
- Les clients de cet algorithme dépendent de cette interface (et uniquement de cette interface)

# Strategy



# Stratégie & ...

- La comparaison des patterns est un axe intéressant pour mieux les cerner :
  - Strategy ressemble à Template Method :
    - Différent dans la granularité
    - Délégation par composition et non aux classes dérivées
  - State et Strategy ont des structures identiques. C'est dans l'intention qu'ils diffèrent. C'est à dire qu'ils résolvent des problèmes différents.
  - State et strategy diffèrent dans la construction du lien
    - en général, une initialisation pour Strategy
    - Beaucoup plus dynamique pour State
  - Strategy et Decorator se ressemblent :
    - Strategy permet de changer les « entrailles » d'un objet.
    - Decorator permet de changer sa « peau »

# *Iterator: quand*

- Accéder aux éléments d'une collection, sans exposer sa structure interne
- Permettre des parcours différents,
- Permettre des parcours concurrents
- Accès homogène :
  - Pour des types de collections différentes (ArrayList, LinkedList)
  - Pour des types de parcours différent (prefixe, postfixe)

=> NE PAS ajouter 50 méthodes à la classe collection !

# *Iterator: comment*

- 2 possibilités :
  - La collection fournit les méthodes de bases de parcours, l'itérateur maintient juste le « curseur »
  - L'itérateur connaît la structure de la collection

# Iterator: exemple

```
Class List1 {
 Class Cell1 { // maillon
 Object value; Cell1 next; }

 // liste chaînée + méthodes add, remove, size ...
 Cell1 first() { return head;}
 private Cell1 head;
 static Cell1 next(Cell1 c) { return c.next; }
 static Object value(Cell1 c) { return c.value;}
}
```

La collection fournit  
les fonctions  
de bas-niveau

```
Class Iterator1 {
 public Iterator1(List1 l) { current=l.first();}
 public Object next() {
 object tmp=List1.value(current);
 current=List1.next(current);
 return tmp;
 }
 public boolean hasNext() { return List1.next(current)!=null;}
 private Cell1 current;
}
```

# Iterator: exemple

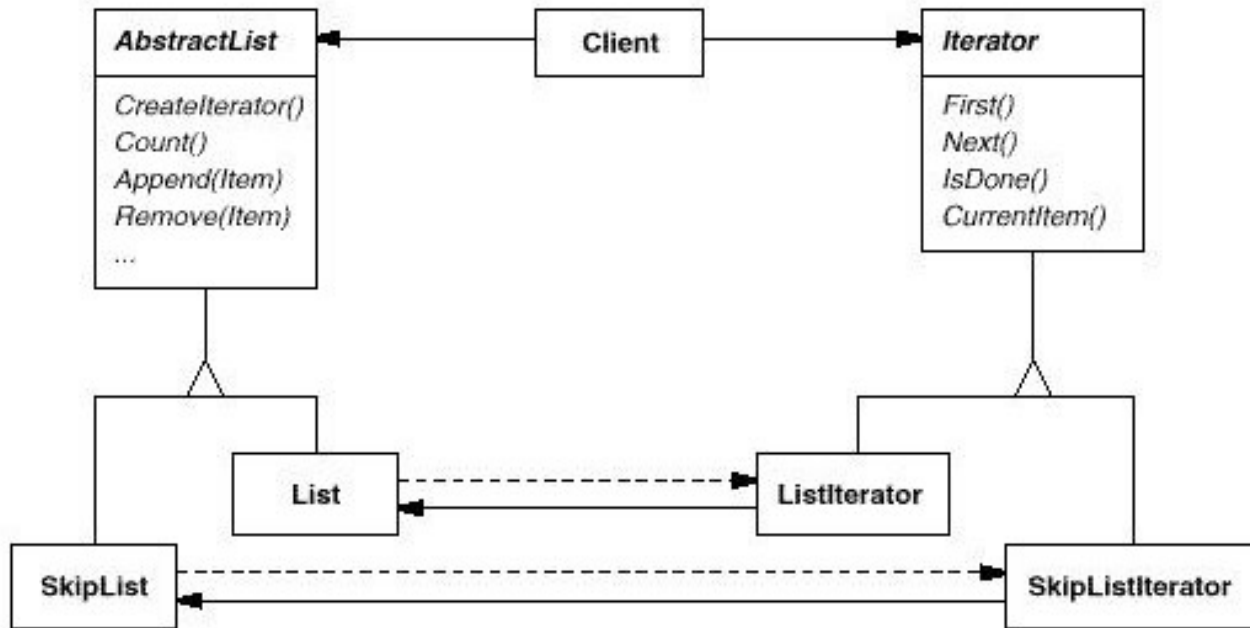
```
Class List1 {
 Class Cell1 { // maillon
 Object value; Cell1 next; }

 // liste chaînée + méthodes add, remove, size ...
 Cell1 first() { return head;}
 private Cell1 head;
}
```

L'itérateur connaît  
la structure interne  
de la collection

```
Class Iterator1 {
 public Iterator1(List1 l) { current = l.first(); }
 public Object next() {
 object tmp=current.value;
 current=current.next;
 return tmp;
 }
 public boolean hasNext() { return current.next!=null;}
 private Cell1 current;
}
```

# Iterator : double polymorphisme

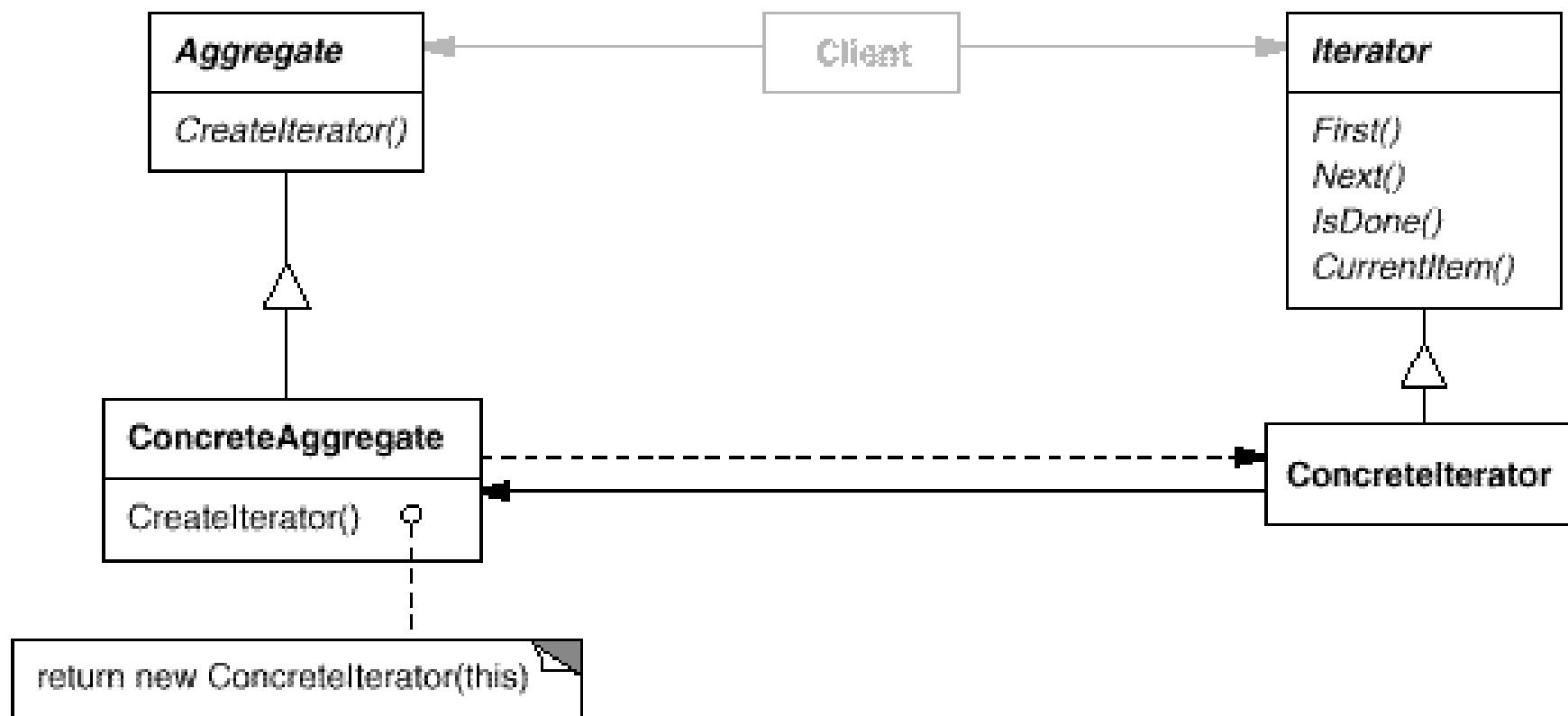


```
List list = new List();
SkipList skipList = new SkipList();
Iterator listIterator = list.CreateIterator();
Iterator skipListIterator = skipList.CreateIterator();
handleList(listIterator);
handleList(skipListIterator);
...
public void handleList(Iterator iterator) {
 iterator.First();
 while (!iterator.IsDone()) {
 Object item = iterator.CurrentItem();
 // process item
 iterator.Next();
 }
}
```

# Iterator : check-list

- Ajouter une méthode `create_iterator()` à la classe "collection", et donner à la classe `iterator` un accès privilégié à la classe `collection`.
- Concevoir la classe « `Iterator` » qui encapsule le parcours des éléments de la collection
- Les clients demandent à la collection de créer un objet itérateur
- Les clients utilisent les méthodes `first()`, `is_done()`, `next()`, and `current_item()` pour accéder aux éléments de la classe « `collection` »

# Iterator



# Iterator : Avantages & risque

- Avantages :
  - Possibilité d'avoir plusieurs types de parcours (exemple pré-fixe, postfixe pour les arbres)
  - Parcours simplifié et uniformisé pour différents types de collections; structure interne cachée
  - Possibilité d'avoir plusieurs parcours en concurrence
  - Découpler collections et algorithmes
- Risque :
  - La responsabilité de parcours est confiée à l'iterator;  
=> pas de vérification de cohérence de l'itérateur en cas de modification de la collection. 3 cas:
    - Crash (ou pire) en cas de modification de la collection pendant l'itération
    - Modifications possibles en passant VIA l'itérateur  
→ add(), remove() fournis par Iterator et ListIterator
    - La collection et l'itérateur peuvent «coopérer » => fail-fast

# Visitor : quand

- Nous avons une hiérarchie d'objets (ou plusieurs!), utilisés dans une structure complexe
  - Arbre, graphe, ...
- Les clients veulent pouvoir exécuter un code qui dépend du type **réel** des objets
  - On ne veut pas avoir plein de `if` et/ou `instanceof` !
- Nous ne connaissons ni ne voulons connaître le code du client !
  - impossible d'utiliser l'appel de méthode virtuel directement car cela suppose d'inclure le code du client dans les objets de la hiérarchie
- Mais on veut tout de même avoir l'efficacité de l'appel virtuel
- Et que le code puisse changer en fonction du client

# Visitor : exemple d'applicabilité

- On a une hiérarchie d'objets, on veut ajouter des méthodes de sérialisation pour obtenir différentes représentations XML.
- 2 solutions
  - Ajouter des méthodes toXml1(), toXml2(), ... à chacune des classes de ma hiérarchie pour ensuite parcourir ma structure (comment?) et me reposer sur le polymorphisme
    - Impossible si je ne peux modifier les objets
    - Non souhaité si je veux que ma hiérarchie d'objets restent relativement simple (et ne pas la rouvrir à chaque nouvelle opération)
  - Créer une classe externe pour implémenter ce traitement
    - Avec plein de *instanceof* !

# Visitor : comment

- On utilise une implémentation qui effectue un **double dispatch** :
  - on crée une interface **Visitor** qui contient une méthode `visit(X x)` différente par type **concret**  $X$  de la hiérarchie
  - le type le plus haut dans la hiérarchie a une méthode
    - `abstract void accept(Visitor v)`
  - Tous les types concrets de la hiérarchie **implémentent** cette méthode de la manière suivante :
    - `void accept(Visitor v) {  
    v.visit(this);  
}`

La surcharge va permettre  
d'appeler  
la BONNE fonction

# Visitor : comment

- Pour utiliser le visiteur, plutôt que d'appeler `visit` sur l'objet de la hiérarchie, on appelle `accept` de l'objet avec le visiteur en argument
  - ~~`visitor.visit(object)`~~  $\longrightarrow$  `object.accept(visitor)`
- *Attention* : même dans les méthodes `visit`, on n'appelle pas directement les méthodes `visit` sur l'objet, mais bien `accept` de l'objet sur visiteur (*i.e.* `this`).
  - ~~`visit(object)`~~  $\longrightarrow$  `object.accept(this)`

La surcharge est static, pas le polymorphisme

# Visitor : pourquoi ça marche

- Quand on appelle `object.accept(visitor)`, la méthode `accept` choisie est celle du **type réel** de `object`
  - Polymorphisme classique
  - cette implémentation appelle la *bonne* méthode `visit` du visiteur ...
- Dans chaque type concret  $X$ , `visitor.visit(this)` appelle la méthode `visit( $X$  x)` car `this` est typé statiquement  $X$ 
  - rappel : le choix de la méthode parmi les surcharges se fait en fonction du type déclaré

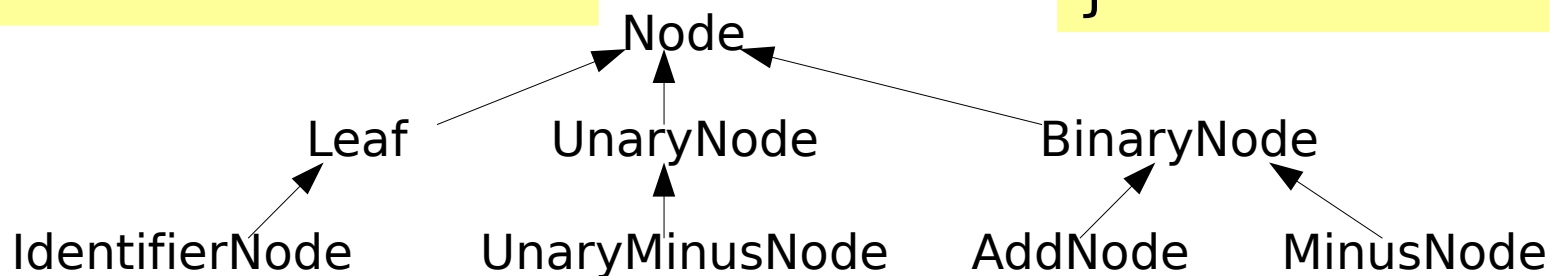
# Visitor : exemple

```
public interface Visitor {
 void visit(AddNode node);
 void visit(MinusNode node);
 void visit(UnaryMinusNode node);
 void visit(IdentifierNode node);
}
```

```
public class AddNode
 extends BinaryNode
{
 void accept(Visitor visitor) {
 return visitor.visit(this);
 }
}
```

```
public interface Node {
 void accept(Visitor visitor);
}
```

```
public class MyVisitor
 implements Visitor{
 ...
}
```

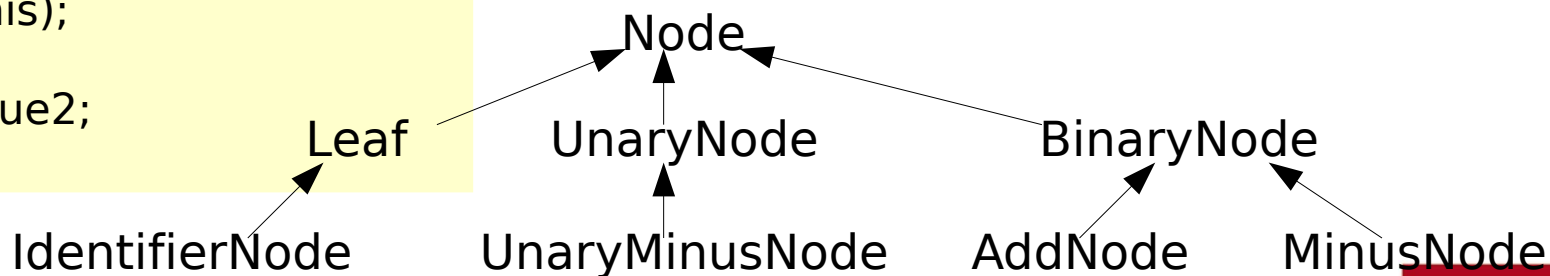


# Visitor : exemple

```
public class EvalVisitor implements Visitor {
 private int value;
 private final Memory memory;
 EvalVisitor(Memory memory) {
 this.memory = memory ;
 }
 int getValue() { return value ; }
 void visit(AddNode node) {
 node.left.accept(this);
 int value1 = value;
 node.right.accept(this);
 int value2 = value;
 value = value1 + value2;
 }
 void visit(MinusNode node) {
 node.left.accept(this);
 int value1 = value;
 node.right.accept(this);
 int value2 = value;
 value = value1 - value2;
 }
}
```

```
void visit(UnaryMinusNode node) {
 node.subNode.accept(this);
 value=-value;
}
void visit(IdentifierNode node) {
 value=memory.get(node.identifier);
}
}
```

```
Expr e ;
...
v= new EvalVisitor(memory) ;
e.accept(v) ;
... v.getValue() ...
```



# Visitor : pour les pros !

- On a utilisé un champ pour transmettre un résultat :(
- On préférera paramétrer le visiteur par
  - le type de retour : R
  - un éventuel argument supplémentaire : P
  - l'exception lancée par les méthodes visit : E
- Les signatures sont modifiées :

```
public interface Visitor<R,P,E extends Throwable> {
 R visit(AddNode node, P parameter) throws E;
 R visit(MinusNode node, P parameter) throws E;
 R visit(UnaryMinusNode node, P parameter) throws E;
 R visit(IdentifierNode node, P parameter) throws E;
}
```

```
public class Minus extends BinaryOp {

 <R,P,E extends Throwable> R
 accept(Visitor<R,P,E> v,P p) throws E {
 return v.visit(this, p) ; }
}
```

```
public interface Node {

 <R,P,E extends Throwable> R
 accept(Visitor<R,P,E> v,P p) throws E ;
}
```

# Visiteur : variantes

- Qui a la responsabilité du parcours de la structure ?
  - Le visiteur (comme dans les exemples abordés)
    - Flexible mais risque de redondance de code
  - La structure : cas fréquent
  - Un itérateur externe
- Les objets de la structure n'implémentent pas forcément la même interface
  - Penser à la différence avec l'itérateur...

# Visitor : astuce

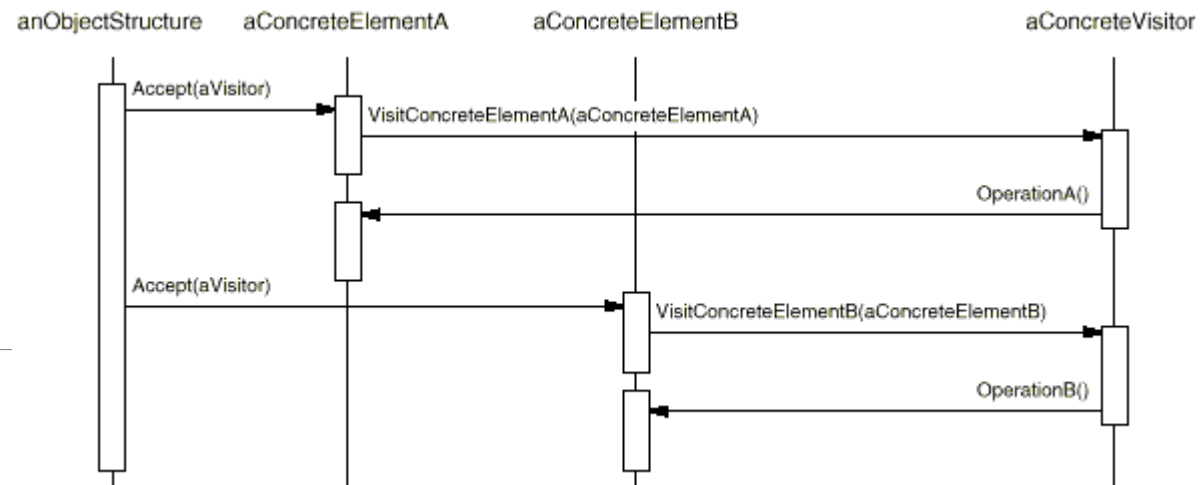
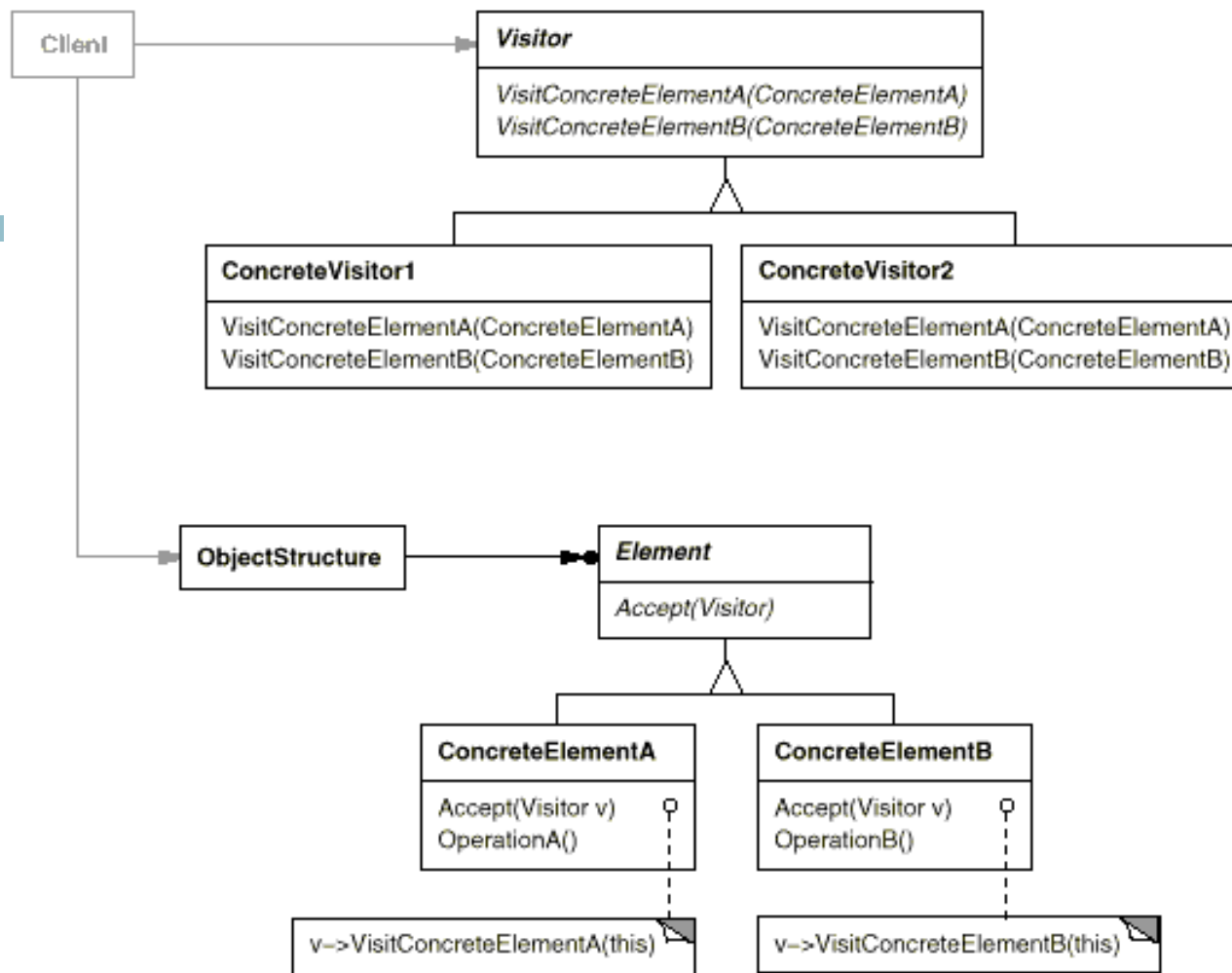
- On veut pouvoir spécifier le même code pour tous les descendants d'un type de la hiérarchie
  - On remplace l'interface Visitor par une classe abstraite (mais sans méthode abstraite) où :
    - par défaut, une méthode visit appelle la méthode visit du supertype direct
    - la méthode visit du type le plus haut lève une AssertionError
- on ne change rien dans la hiérarchie, seul le visiteur change

# Visitor : check-list

- Vérifier que la hiérarchie de la structure d'objets sera « assez » stable; et que l'interface de ces classes fournit ce qui sera nécessaire aux visiteurs.
- Créer la classe de base Visitor avec une méthode `visit(ElementXxx)` pour chaque classe dérivée d'Element
- Ajouter une méthode `accept(Visitor)` à la hiérarchie d'Element. L'implémentation standard est :  

```
accept(Visitor v) { v.visit(this); }
```
- La hiérarchie Element est juste couplée à l'interface Visitor. Les classes de la hiérarchie Visitor sont couplées à chaque classe de la hiérarchie Element.
- Chaque « operation » donne lieu à la création d'un Visiteur. L'implémentation des fonctions `visit()` doit s'appuyer sur l'interface public des classes Element's
- Le client crée les objets Visitor et les passe aux Element's en appelant `accept()`

# Visitor



# Visitor : conséquences

- « facile » d'ajouter une nouvelle opération
  - Un nouveau visiteur
- Sépare fortement les opérations indépendantes
  - Pas de couplage technique si pas de couplage fonctionnel
- Ajouter de nouvelles classes concrètes dans la structure est compliqué !
  - Nouvelle méthode visit + impl. Pour tous les visiteurs
- Visiteur est donc adapté quand
  - La structure ne change pas
  - Les opérations sur la structure changent souvent
- Le visiteur oblige souvent à étendre l'interface publique des objets de la structure