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 ou mon collègue
- 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

1. Une classe abstraite implémente la partie stable d'un algorithme

2. 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 déri
        Simple "placefolder", "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();
}
```

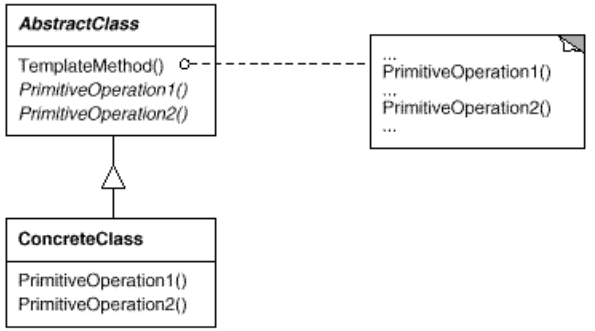
```
class Realization extends Specialization {
   // 4. les classes dérivées peuvent impl. certaines étapes
   protected void stepTwo() { System.out.println( "Realization   .stepTwo" ); }
   protected void step3_2() { System.out.println( "Realization   .step3_2" ); }
   // 5. les classes dérivées peuvent 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
```

© Huston

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

Encore un moyen de limiter le couplage entre appelant et objet traitant la requête

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

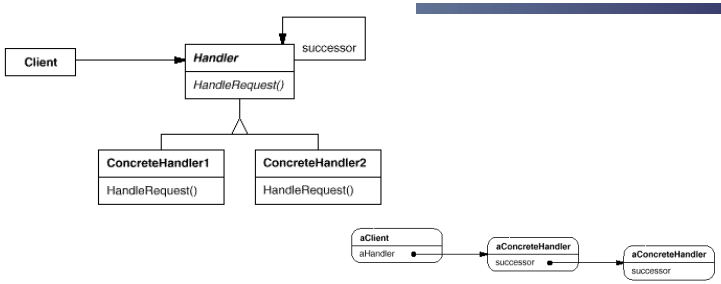
```
public class ChainBidDemo {

  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;
  } }
```

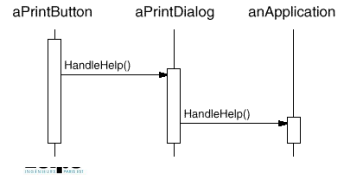
```
  static Link setUpChain() {
    Link first = new Link( 1 );
    for (int i=2; i < 7; i++)
      first.addLast( new Link( i ) );
    return first;
  }

  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

Encore un moyen de limiter le couplage
1. entre l'appelant et l'objet traitant la requête
2. entre l'appelant et la manière dont la requête sera traitée

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, auprès d'un « invoqueur »
- 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 (1)

```
public interface Command {
  void do();
}
```

```
public class InsertCharacter implements Command {
  private final char c;
  private final Document doc;
  public InsertCharacter(Document doc, char c) { … }
  void do() {
    int pos = doc.getCarretPosition();
    doc.appendAfter(pos,c);
  }
}
```

```
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() {
    int[] positions = doc.findAll(from);
    for(int i=positions.length-1;pos>=0;pos--)
      doc.replace(positions[i],from.length,to);
  }
}
```


Command : exemple (2)

```
public class Commander {
   public void apply(Command command) {
     command.do();
   }
}
```

```
// Client

…
     // sur réception de caractère entré
     Command c = new InsertCharacter(doc, c) ;
     commander.apply(c) ;
…

     // sur OK dans dialogue de rechercher&remplacer
     Command c = new ReplaceAll(doc, searchStr, replaceStr) ;
     commander.apply(c) ;
```

avec undo (1)

```
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);
  }
}
```


avec undo (2)

```
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);
  }
}
```


avec undo (3)

```
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);
  }
}
```


avec undo (4)

IDENTIQUE

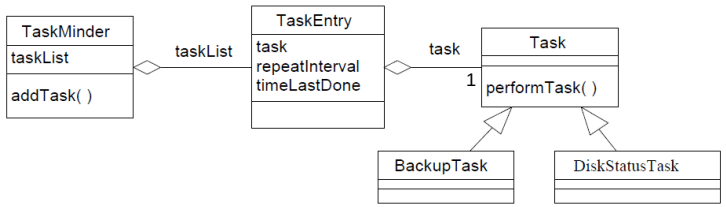
```
// Client

…
    // sur réception de caractère entré
    Command c = new InsertCharacter(doc, c) ;
    commander.apply(c) ;
…

    // sur OK dans dialogue de rechercher&remplacer
    Command c = new ReplaceAll(doc, searchStr, replaceStr) ;
    commander.apply(c) ;
```

Nouvelle fonctionnalité

```
// Client

…
    // sur CTRL+Z
    commander.undo() ;
…
```

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)

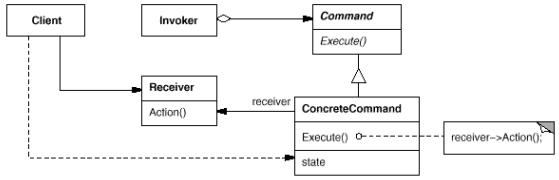
```
public class TaskMinder extends Thread {
    private long sleepInterval; // How often the TaskMinder should
    private List<TaskEntry> taskList; // The list of tasks

    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);
            }
        }
    }
}
```

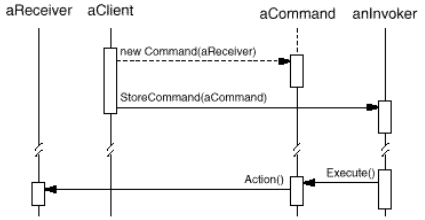
Scheduler (5)

```
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





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* et *comment* appeler execute().

Observer : quand

- Je veux savoir quand un objet A change
  - mettre à jour une vue graphique
  - programmer la sauvegarde les 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 = mo
   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 précise 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 : 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 ) { maj(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 des observers
- 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 : comment

- L'information sur les changements est donnée par le choix de la méthode, et éventuellement son argument
- Les observateurs doivent être prévenus **après** le changement effectué dans l'objet
- On doit préciser dans l'argument 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 get*Property*
- On peut fournir des informations complémentaires pour optimisation :
  - Ex : l'ancienne valeur,
  - Ex : pour une liste, dire quels indices ont changés

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);
  }
}
```

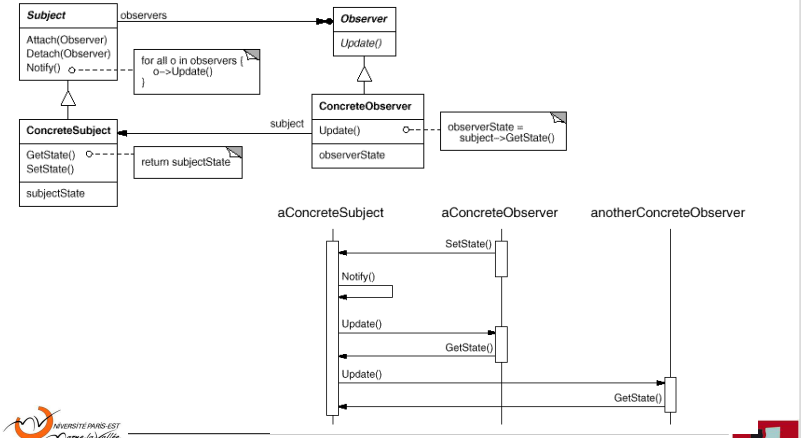

Observer : exemple

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

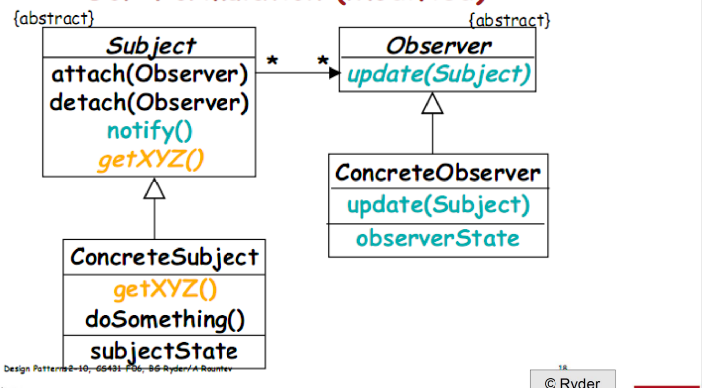
Observer : check-list

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. Ou les Observers s'enregistrent eux-même auprès du Subject.
7. Le Subject diffuse des évènements 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)



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) {…}
    public boolean hasChanged()  {...}
}
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