POA

Cours de 2e année ingénieur
TSI

fabien.romeo@fromeo.fr
http://www.fromeo.fr

Plan

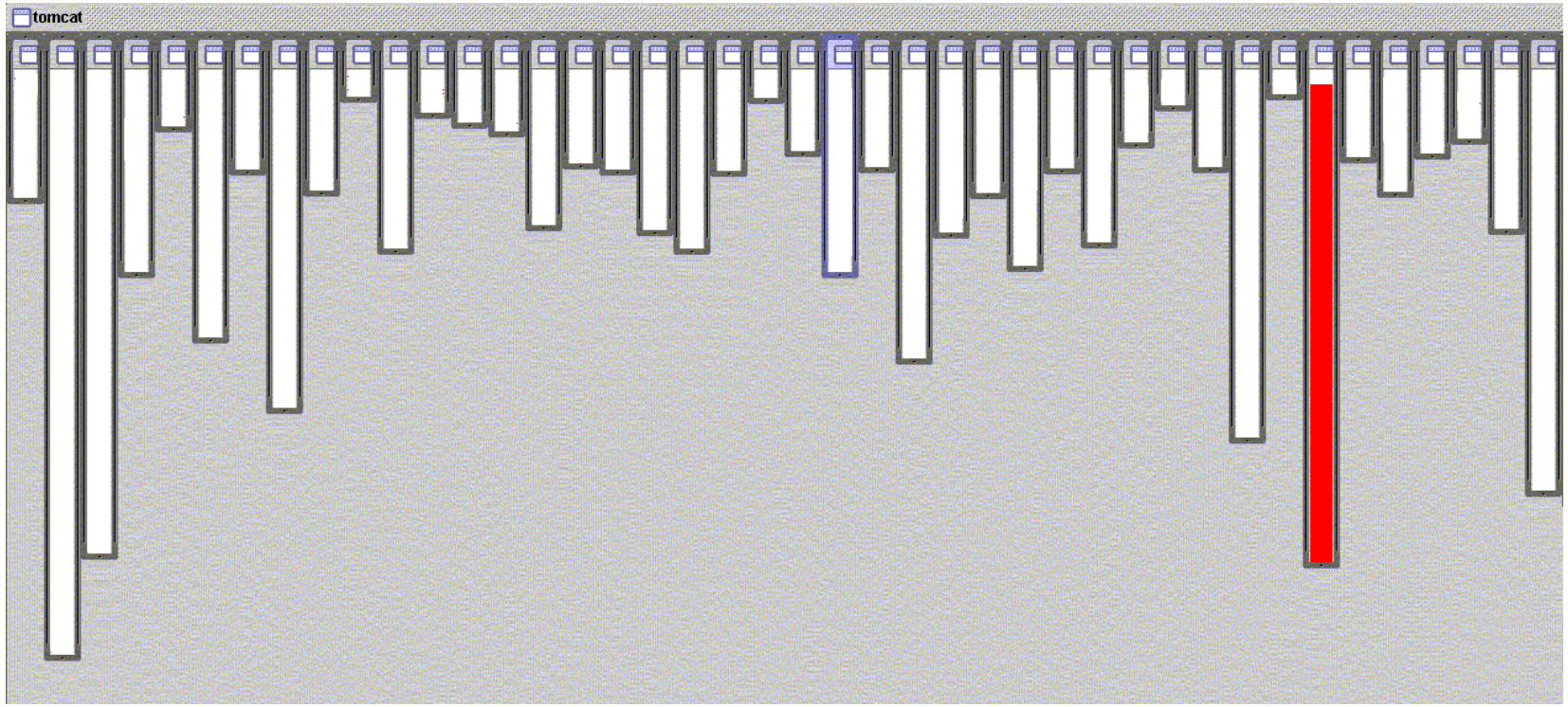
- Introduction
  - Origines
  - Limites de la POO
  - Apports de la POA
  - Bibliographie
- Motivation
  - Exemple appliqué au pattern Observer
- Concepts de la POA
  et mise en œuvre avec AspectJ
  - Aspects, points de jonction,coupes, advices
  - Mécanisme d'introduction
  - Fonctionnalités avancées
- Applications de la POA
  - Design Patterns
  - Logging, Management, …

Introduction

Origines

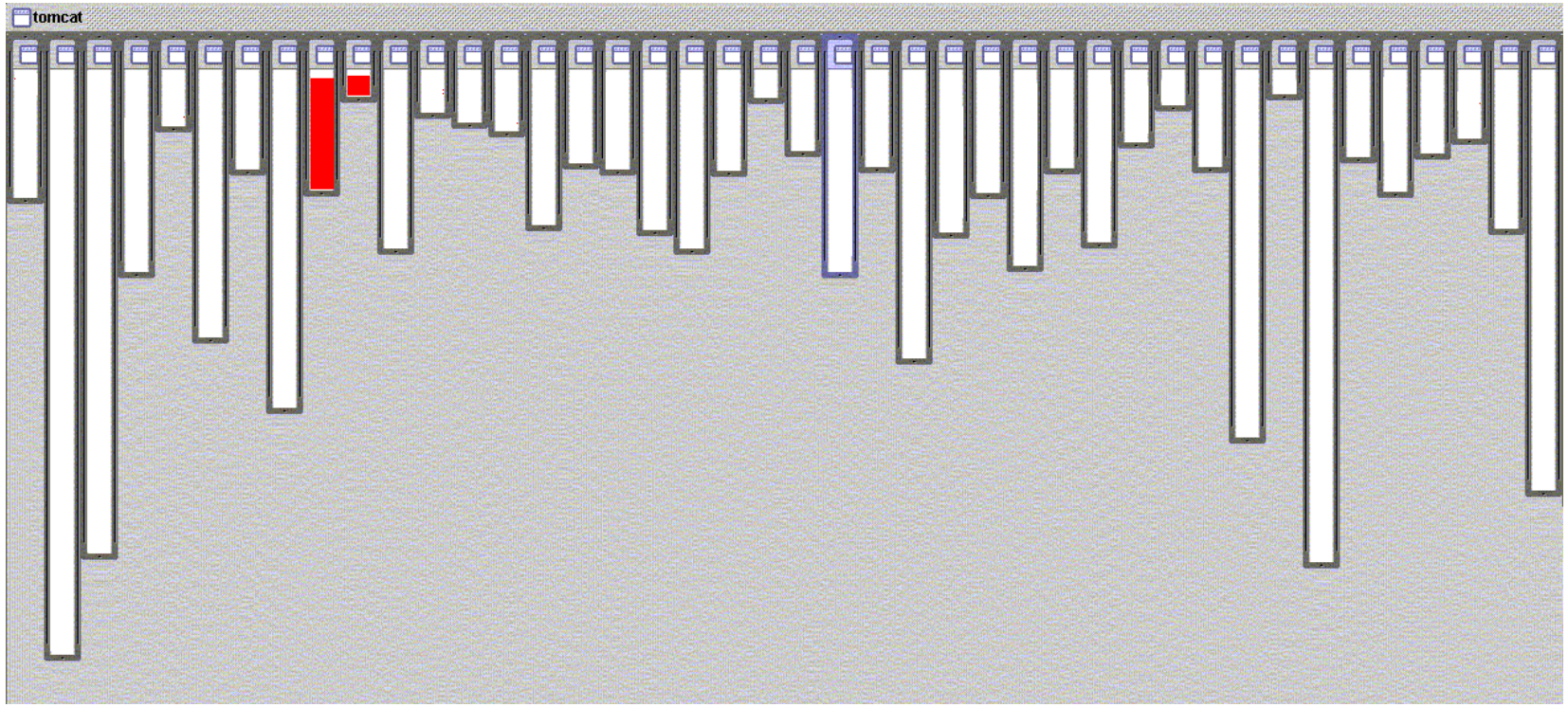
- Programmation Orientéé Aspect (POA) ou *Aspect-Oriented Programming* (AOP)
- Kiczales et al., Xerox PARC : AspectJ
  - Maintenant dans projet Eclipse
- Introduction d'un nouveau **concept** : **<u>l'aspect</u>**
- Ne se substitue pas aux autres paradigmes (en particulier la POO) mais cherche à les compléter pour obtenir des programmes mieux structurés
- Les idées de la POA proviennent en particulier de la métaprogrammation, de la réflexivité et des protocoles à méta-objets
- Connexions également avec les techniques actuelles comme la programmation générative, l'ingénierie des modèles (MDA), …

POO : bonne modularité



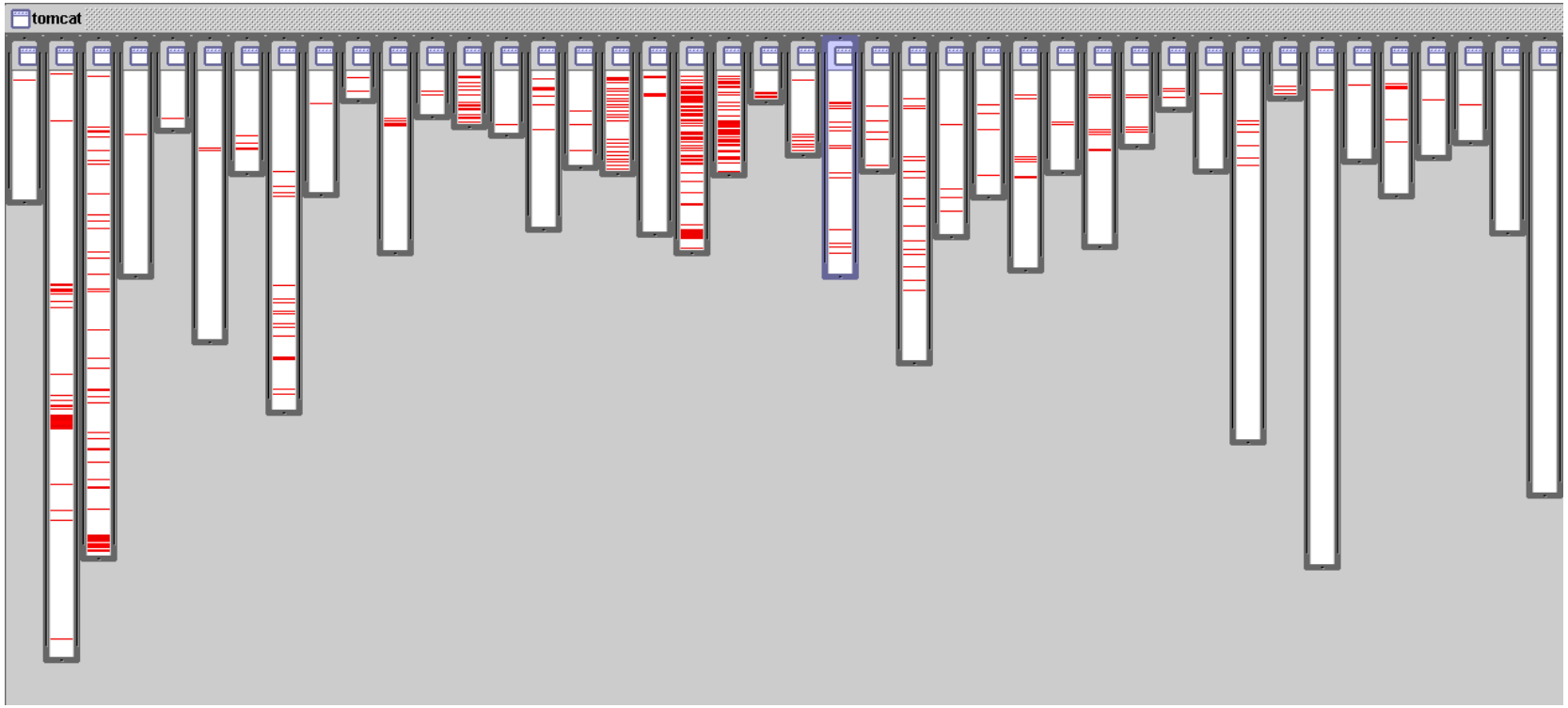
- XML parsing dans org.apache.tomcat
  - En rouge : les lignes de code correspondantes
  - Tout est dans une seule classe

POO : bonne modularité



- URL pattern matching dans org.apache.tomcat
  - En rouge : les lignes de code correspondantes
  - Tient dans deux classes (avec héritage)

POO : mauvaise modularité des fonctionnalités transversales



- logging dans org.apache.tomcat
  - En rouge : les lignes de code correspondantes
  - Ne peut être localisé dans une seule classe
  - Même pas dans un petit nombre

POO : mauvaise modularité des fonctionnalités transversales

ApplicationSession

StandardSession

- Expiration des sessions dans org.apache.tomcat

SessionInterceptor

StandardManager

StandardSessionManager

ServerSession

ServerSessionManager

Limites de la POO

- Fonctionnalités transversales
  - Code entrelacé avec le reste de l'application
- Dispersion du code
  - Interaction entre les objets est réalisée par l'invocation de méthode
  - L'implémentation d'une méthode est clairement localisée
  - Par contre, l'invocation (l'utilisation) est dispersée dans toutes les classes appelantes
    - Que se passerait-il si Sun modifiait System.out.println() en System.out.printLn() ?
  - → problèmes de maintenance et d'évolution du code

Problème du code entrelacé et dispersé

- Code redondant
  - Le même morceau de code est dupliqué dans de nombreux endroits
- Difficile à maîtriser
  - Pas de structure explicite
  - Pas de lien explicite entre les différents morceaux
- Difficile à changer
  - Trouver toutes les classes à modifier (sans en oublier)
  - Trouver le code à modifier dans les classes
  - Tout modifier de façon identique

Apports de la POA

- Fournit un moyen de rassembler dans un aspect du code qui, autrement, serait dispersé au sein d'une application
- Un aspect est une entité logicielle qui capture une fonctionnalité transversale à une application (crosscutting structure)
- En résumé, la POA est une solution
  - à l'implémentation des fonctionnalités transversales
  - aux phénomènes de dispersion du code

Bibliographie

- **Aspect-Oriented Programming**,
  Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin,
  in ECOOP'97---Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220--242, 1997.

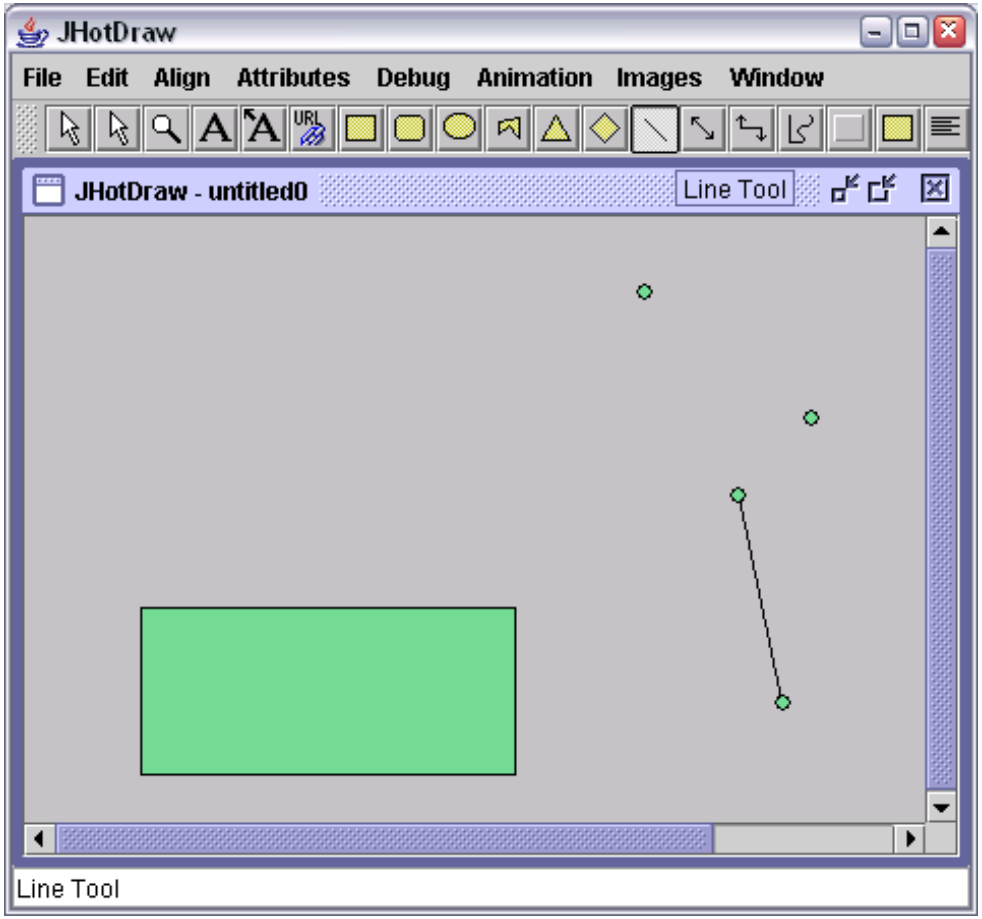
- **AspectJ Home Page**
  http://www.eclipse.org/aspectj/

- **Programmation orientée aspect pour Java/*J2EE*,**
  Renaud Pawlak, Jean-Philippe Retaillé, Lionel Seinturier,
  Eyrolles, 2004, 446 p., ISBN 2212114087

Motivation

(cf. http://video.google.com/videosearch?q=Kiczales)

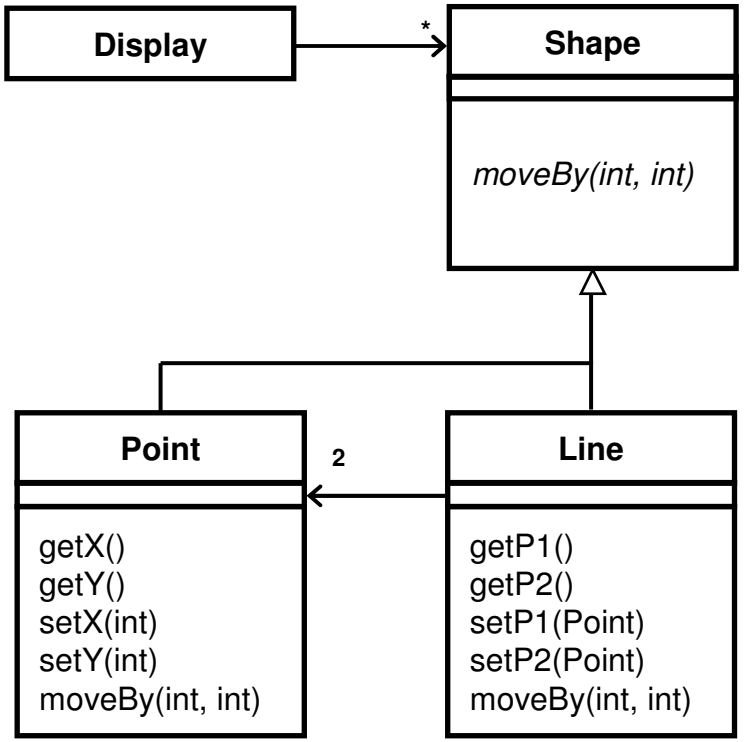
On veut construire une application de dessin



(ex : www.JHotDraw.org)

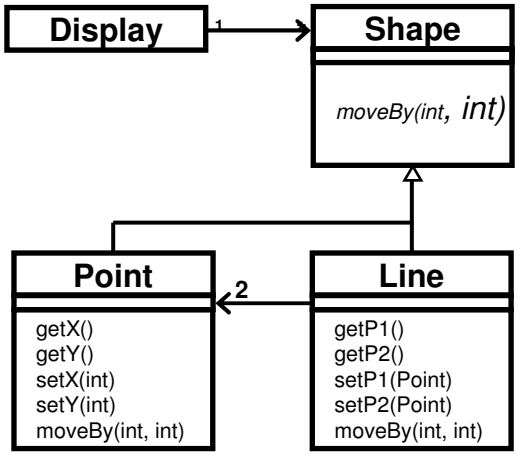
En conception orientée objet

- On décompose en objets :
  - Points
  - Lignes
  - Surfaces de dessin
  - Composants GUI
  - …



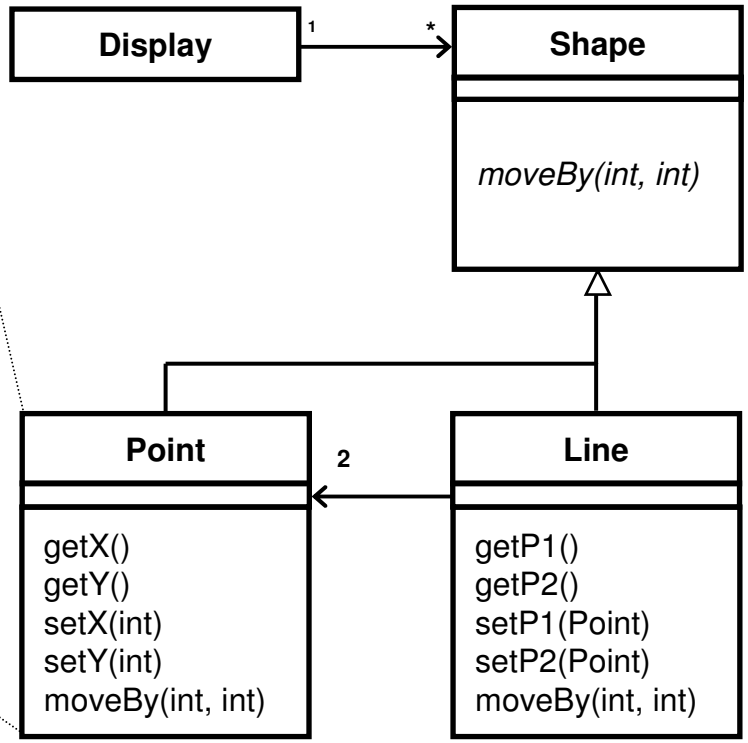
- Bonne correspondance entre code et design
  - Si on veut changer le fonctionnement d'un point, on modifie la classe Point
  - Si on veut ajouter une forme, on ajoute une classe dérivée de Shape
  - …

```
class Point extends Shape {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```



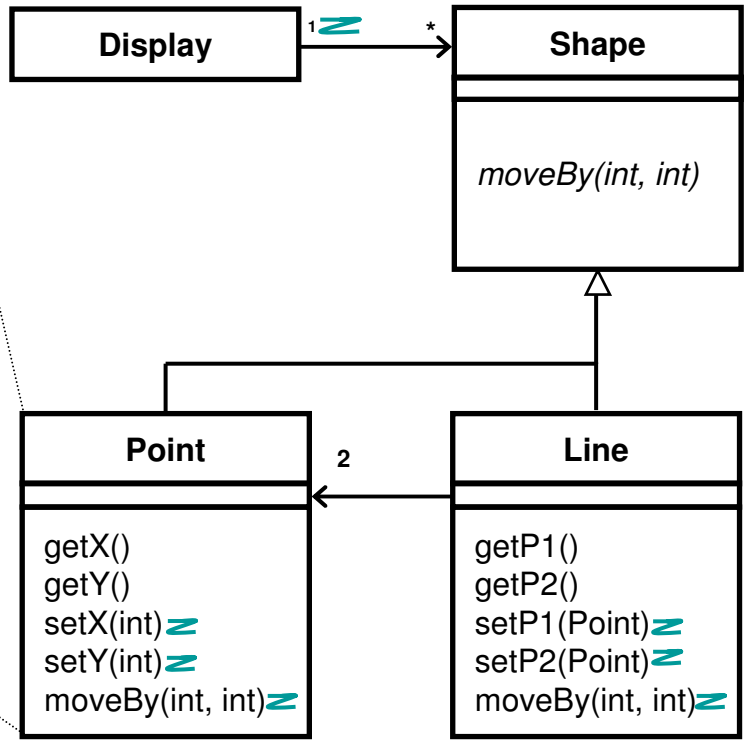
Mais où se trouve le pattern Observer ?

```
class Point extends Shape {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    display.update(this);
  }
  void setY(int y) {
    this.y = y;
    display.update(this);
  }
}
```



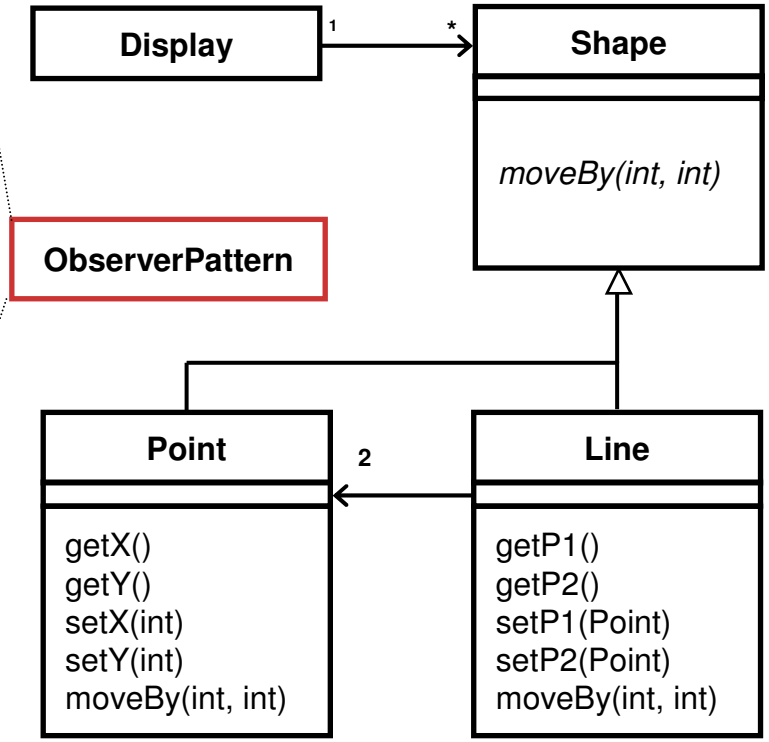
Mais où se trouve le pattern Observer ?

```
class Point extends Shape {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    display.update(this);
  }
  void setY(int y) {
    this.y = y;
    display.update(this);
  }
}
```



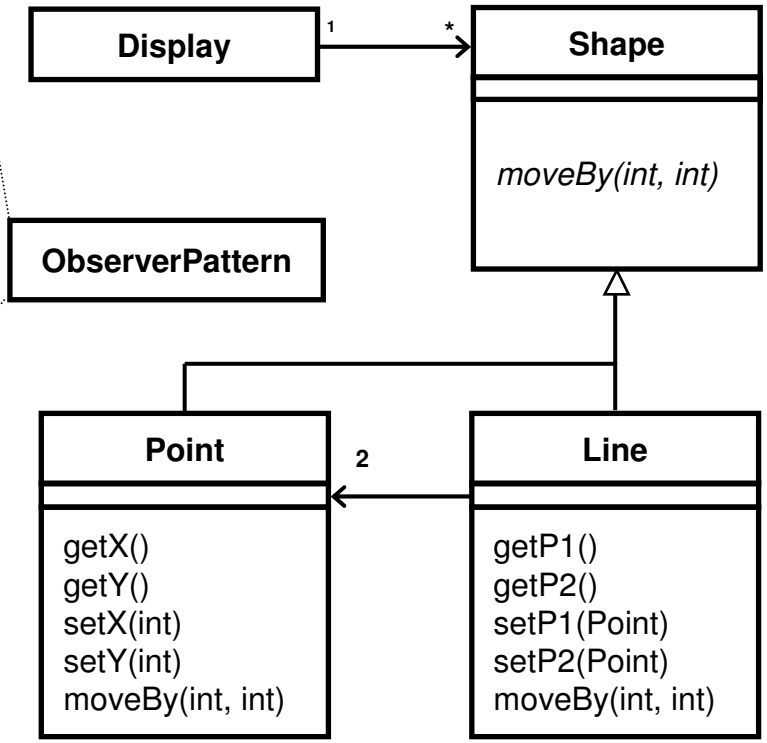
Solution : la POA

```
aspect ObserverPattern {

  private Display Shape.display;

  pointcut change():
    call(void figures.Point.setX(int))
    || call(void Point.setY(int))
    || call(void Line.setP1(Point))
    || call(void Line.setP2(Point))
    || call(void Shape.moveBy(int, int));

 after(Shape s) returning: change()
                           && target(s) {
   s.display.refresh();
  }
}
```



Le code ressemble au design

```
aspect ObserverPattern {

  private Display Shape.display;

  pointcut change():
    call(void Shape.moveBy(int, int))
    || call(void Shape+.set*(..));

  after(Shape s) returning: change()
                            && target(s) {
   s.display.refresh();
  }
}
```



TP

- Installation du plugin Eclipse AJDT
- HelloWorld avec AspectJ
  - http://www.eclipse.org/ajdt/demos/HelloWorldAspectJ.html
- Utilisation des vues AspectJ
  - Cross references
- Utilisation du Visualiser
- 1er exemple de [PRS]

TP : gestion de commande [PRS]

- L'application gestion de commande permet à un client d'ajouter des articles à une commande puis d'en calculer le montant.
- Les références et les prix des articles sont stockés dans un catalogue.
- 3 classes
  - Customer
  - Order
  - Catalog

TP: Customer

```
package aop.aspectj;

public class Customer {

    /**
     * L'activite du client.
     */
    public void run() {
        Order maCommande = new Order();
        maCommande.addItem("DVD",1);
        maCommande.addItem("CD",2);
        double montant = maCommande.computeAmount();
        System.out.println("Montant de la commande : "+montant+" euros");
    }

    public static void main(String[] args) {
        new Customer().run();
    }

}
```

TP: Order

```
package aop.aspectj;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class Order {

    private Map articles = new HashMap();

    public void addItem( String reference, int quantite ) {
        articles.put(reference,new Integer(quantite));
        System.out.println(
            quantite+" article(s) "+reference+" ajoute(s) a la commande");
    }

    public double computeAmount() {
        double montant = 0.0;
        for (Iterator iter = articles.entrySet().iterator();
   iter.hasNext();) {
            Map.Entry entry = (Map.Entry) iter.next();
            String article = (String) entry.getKey();
            Integer quantite = (Integer) entry.getValue();
            double prix = Catalog.getPrice(article);
            montant += prix*quantite.intValue();
        }
        return montant;
    }

}
```

TP: Catalog

```
package aop.aspectj;

import java.util.HashMap;
import java.util.Map;

public class Catalog {

    private static Map tarif = new HashMap();

    static {
        tarif.put( "CD", new Double(15.0) );
        tarif.put( "DVD", new Double(20.0) );
    }

    public static double getPrice( String reference ) {
        Double prix = (Double) tarif.get(reference);
        return prix.doubleValue();
    }

}
```

TP: TraceAspect

```
package aop.aspectj;

import java.util.Date;
import java.util.Map;

public aspect TraceAspect {

    /**
     * Cette coupe est tres detaillee puisque seuls les appels a la methode
     * dont la signature est citee sera tracee.
     */
    pointcut toBeTraced():
        call( public void aop.aspectj.Order.addItem(String,int) );

    /**
     * Code advice qui affiche les messages de trace.
     */
    void around(): toBeTraced() {
        System.out.println("-> Avant appel addItem");
        proceed();
        System.out.println("-> Apres appel addItem");
    }

}
```

TP: TraceAspect2

```
package aop.aspectj;

import java.util.Date;
import java.util.Map;

public aspect TraceAspect2 {

    /**
     * Cette coupe capture les appels a toutes les methodes de la classe Order.
     */
    pointcut toBeTraced(): call( * aop.aspectj.Order.*(..) );

    /**
     * Code advice qui affiche les messages de trace.
     */
    Object around(): toBeTraced() {
        System.out.println("-> Avant appel");
        Object ret = proceed();
        System.out.println("-> Apres appel");
        return ret;
    }

}
```

TP: TraceAspect3

```
public aspect TraceAspect3 {

    pointcut toBeTraced(): call( * aop.aspectj.Order.*(..) );

    /**
     * Code advice qui affiche les messages de trace et
     * les informations sur l'introspection des points de jonction.
     */
    Object around(): toBeTraced() {

        String methodName = thisJoinPoint.getSignature().getName();
        Object[] args = thisJoinPoint.getArgs();
        Object caller = thisJoinPoint.getThis();
        Object callee = thisJoinPoint.getTarget();

        System.out.println("-> Debut methode "+methodName);
        System.out.print("-> "+args.length+" parametre(s) ");
        for (int i = 0; i < args.length; i++) {
            System.out.print( args[i]+" " );
        }
        System.out.println();
        System.out.println("-> "+caller+" vers "+callee);
        Object ret = proceed();
        System.out.println("<- Fin methode "+methodName);
        return ret;
    }
}
```

Concepts de la POA
et mise en œuvre avec AspectJ

AspectJ

- Un langage pour la POA
  - Extensions syntaxiques de Java (aspect, pointcut, call, execution, before, after, …)
  - Définir des règles de métaprogrammation : modifications de l'architecture et du comportement
- Open Source (projet Eclipse)
- Outils AJDT
  - Tisseur d'aspects (compilateur)
  - Cross references
  - Visualizer

Possibilités d'AspectJ

- Sur l'architecture
  - Modification de la structure statique du système (classes, interfaces, …)
  - Ajout d'attributs et méthodes
- Sur le comportement
  - Ajout d'un nouveau comportement à l'exécution « normale » du programme
  - Etendre ou remplacer une opération
  - Action effectuée avant/après l'exécution de certaines méthodes ou la gestion d'exception (catch)

4 concepts de base

- Aspect
  - Entité logicielle qui capture une fonctionnalité transversale
  - Coupe(s) associée(s) à un advice
- Point de jonction (*join point*)
  - Point dans le flot d'exécution du programme
    (ex : appel de méthode, lecture d'attribut, …)
- Coupe (*pointcut*)
  - Ensemble de points de jonction
  - Quantification sur le programme (* .. +)
    (ex : coupe d'appel d'un ens. de méthodes)
- Greffon (*advice*)
  - Le comportement d'un aspect (code)
  - Différents types selon le moment d'insertion
    (ex : before, around , after, …)

Aspect

Coupe

```
aspect log {
   pointcut traceMethode() :
      call (void Figure.set*(int))  ||
         execution (!public static*com.xerox.*.*.*(..));

      before() : traceMethode():
     { System.out,println("..");     }

}
```

Advice

- Chaque advice est associé à une coupe
- Plusieurs coupes et advices par aspect

Types de points de jonction

- Méthodes
  - call : appel d'une méthode
  - execution : execution d'une méthode
- Données
  - get : lecture d'un attribut
  - set : affectation d'une valeur à un attribut (écriture)
- Constructeurs
  - initialization : instanciation d'un objet par un constructeur
  - preinitialization : instanciation par un constructeur avant l'appel à **super**
- Initialisation statique de classe
  - staticinitialization : initialisation d'une classe après chargement
- Exceptions
  - handler : gestion d'une exception avec **catch**

Coupes

- Syntaxe
  - [mode_acces] **pointcut** nom_pointcut([args]) : definition_pointcut

- Exemple
  - public **pointcut** printCall() : **call** (void Communicator.print(..))

Opérateurs de coupe

- * n’importe quel caractère sauf .
- .. n’importe quel niveau de sous-package
- .. n’importe quel profil de méthode
- + n’importe quelle sous-classe
- && || ! et, ou , non
- withincode(method) Tout PJ dans method
- within(class) Tout PJ dans class
- this(class) PJ dont la source est de type class
- target(class) PT dont la cible est de type class

Advices

- Syntaxe
  - specification_advice([args]):nom_pointcut([args]) {
    // code de l'advice
    }
- Exemple
  - before() : printCall() {
    System.out.println("before printCall()");
    }

Types d’advice

- before()
  - Avant chaque point de jonction
- after() returning
  - Après chaque terminaison normale des points de jonction
- after() throwing
  - Après chaque terminaison anormale des points de jonction (levée d’une exception)
- after()
  - Après chaque terminaison normale ou anormale des points de jonction
- around()
  - Avant et après chaque point de jonction
  - Délimitation avant/après avec le mot-clé **proceed()**

Introduction de code (inter-type definition)

- Possibilité d'étendre le comportement d'une application en ajoutant des éléments
  - Attributs, méthodes et constructeurs
  - Héritage et implémentation d'interface
  - Exceptions
- Pas de condition pour l'introduction contrairement aux advices (coupe)

Introduction de code : attributs, méthodes et constructeurs

- But : étendre des classes par injection d'attributs, de méthodes, de constructeurs

```
class Facture { ... }

aspect Horodateur {

  private Date Facture.date;
  public void Facture.setDate(Date date) { this.date=date; }
  public Date Facture.getDate() { return date; }
  public Facture.new(Date date) { this.date=date; }

 after(): initialization(Facture.new(..)) {
  Facture maFacture = (Facture) thisJoinPoint.getTarget();
  maFacture.date = new Date();
 }
}
```

Introduction de code : héritage et implémentation d'interface

- But : étendre des classes par injection de super-classe, de super-interface

```
aspect HorodateurBis {
 declare parents: Facture implements HorodateurItf;
 declare parents: Facture extends UnicastRemoteObject;
}
```

Introduction de code : exceptions

- But : ajout d'un gestionnaire d'exception
- Les exceptions de l'application sont encapsulées dans une exception de type org.aspectj.lang.SoftException
- Ex :

```
declare soft: IOException+: call(* InputStream.*(..));
```

Fonctionnalités avancées : héritage et aspects abstraits

- Aspect abstrait : permet de découpler coupe et advice

```
abstract aspect AbsLog {
  abstract pointcut toBeTraced();
  Object around(): toBeTraced() { ... }
}
aspect MyLog extends AbsLog {
  pointcut toBeTraced(): Command.new(..);
}
```

Fonctionnalités avancées : composition d'aspect

- Que se passe-t-il si plusieurs aspects interviennent sur un même point de jonction ?
- Ordonnancement explicite

```
aspect OrdreGlobal {
  declare precedence: Authentication, Log;
}
aspect Authentication { ... }
aspect Log { ... }
```

- Ordonnancement implicite (cf.doc)

Références

- RefCard :
  - http://www.eclipse.org/aspectj/doc/released/quick5.pdf
- Design Pattern & AspectJ
  - http://www.cs.ubc.ca/~gregor/papers/hannemann-OOPSLA2002-aop-patterns.pdf
  - Code : http://hannemann.pbwiki.com/Design+Patterns

Applications de la POA

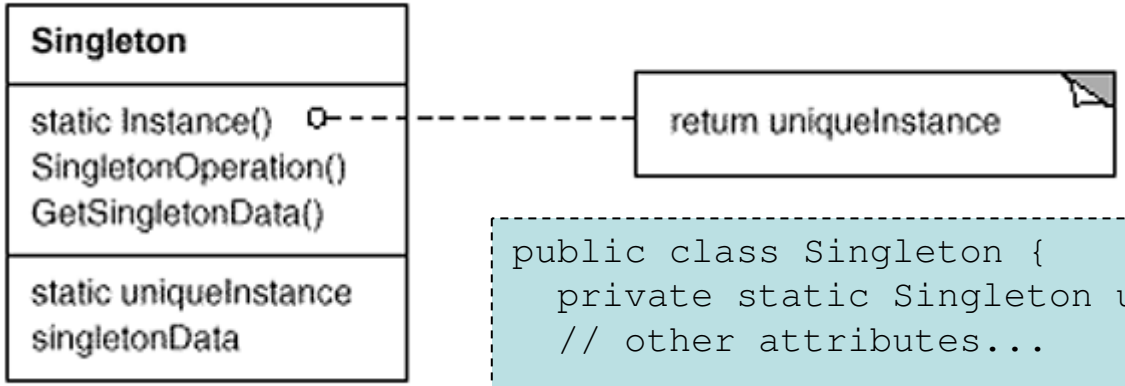
Design Patterns & POA

- Avantages de la POA
  - Localisation
    - Code contenu au sein d'aspects
  - Réutilisation
    - Factorisation du code et meilleure abstraction
  - Composition
    - Possibilité de faire participer un même objet à plusieurs patterns de façon transparente
  - Adaptabilité
    - Lien moins fort entre application et design patterns utilisés, paramétrage pour l'activation/désactivation

Table 1. Design pattern, roles, and desirable properties of their AspectJ implementations

	Modularity Properties				Kinds of Roles	
Pattern Name	Locality(**)	Reusability	Composition Transparency	(Un)pluggability	Defining(*)	Superimposed
Façade	Same implementation for Java and AspectJ				Façade	-
Abstract Factory	no	no	no	no	Factory, Product	-
Bridge	no	no	no	no	Abstraction, Implementor	-
Builder	no	no	no	no	Builder, (Director)	-
Factory Method	no	no	no	no	Product, Creator	-
Interpreter	no	no	n/a	no	Context, Expression	-
Template Method	(yes)	no	no	(yes)	(AbstractClass), (ConcreteClass)	(AbstractClass), (ConcreteClass)
Adapter	yes	no	yes	yes	Target, Adapter	Adaptee
State	(yes)	no	n/a	(yes)	State	Context
Decorator	yes	no	yes	yes	Component, Decorator	ConcreteComponent
Proxy	(yes)	no	(yes)	(yes)	(Proxy)	(Proxy)
Visitor	(yes)	yes	yes	(yes)	Visitor	Element
Command	(yes)	yes	yes	yes	Command	Commanding, Receiver
Composite	yes	yes	yes	(yes)	(Component)	(Composite, Leaf)
Iterator	yes	yes	yes	yes	(Iterator)	Aggregate
Flyweight	yes	yes	yes	yes	FlyweightFactory	Flyweight
Memento	yes	yes	yes	yes	Memento	Originator
Strategy	yes	yes	yes	yes	Strategy	Context
Mediator	yes	yes	yes	yes	-	(Mediator), Colleague
Chain of Responsibility	yes	yes	yes	yes	-	Handler
Prototype	yes	yes	(yes)	yes	-	Prototype
Singleton	yes	yes	n/a	yes	-	Singleton
Observer	yes	yes	yes	yes	-	Subject, Observer

Singleton pattern




```
public class Singleton {
  private static Singleton uniqueInstance;
  // other attributes...

  private Singleton() {}

  public static Singleton getInstance() {
    if (uniqueInstance == null) {
        uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  // other methods...
}
```

```
public static void main(String[] args) {
  Singleton singleton;
  singleton = Singleton.getInstance();
}
```

Singleton Aspect

```
public abstract aspect AbstractSingletonAspect {

    private Object singleton = null;

    abstract pointcut singletonPointcut();

        Object around(): singletonPointcut() {
                if (singleton == null) {
                        singleton = proceed();
                }
                return singleton;
        }
}
```

Pourquoi un aspect abstrait ?

Un aspect est lui-même un singleton, donc un aspect concret serait partagé par toutes les classes à transformer en singleton.

Il faut ensuite un aspect concret par classe à transformer.

```
public aspect SingletonAspect extends AbstractMultipleSingletonAspect {

        pointcut singletonPointcut() : call(MyClass.new(..));

}
```

```
public static void main(String[] args) {
  MyClass myClass = new MyClass();
}
```

Singleton Aspect

```
public abstract aspect SingletonPattern {
  private Hashtable singletons = new Hashtable();
  public interface Singleton {}

  pointcut selectSingletons(): call((Singleton+).new(..));

    Object around(): selectSingletons() {
      Class singleton = thisJoinPoint.getSignature().getDeclaringType();
        if (singletons.get(singleton) == null) {
          singletons.put(singleton, proceed());
        }
        return (Object) singletons.get(singleton);
    }
}
```

```
public aspect SingletonInstance extends SingletonPattern {

        declare parents: MyClass implements Singleton;
        declare parents: MyOtherlass implements Singleton;
}
```

```
public static void main(String[] args) {
  MyClass myClass = new MyClass();
  MyClass myOtherClass = new MyOtherClass();
}
```

Singleton Aspect

```
public abstract aspect SingletonPattern {
  private Hashtable singletons = new Hashtable();
  public interface Singleton {}
  public interface NonSingleton {}

  pointcut selectSingletons(): call((Singleton+).new(..));
  pointcut excludeNonSingletons(): !call((NonSingleton+).new(..));

    Object around(): selectSingletons() && excludeNonSingletons() {
      Class singleton = thisJoinPoint.getSignature().getDeclaringType();
        if (singletons.get(singleton) == null) {
          singletons.put(singleton, proceed());
        }
        return (Object) singletons.get(singleton);
    }
}
```

```
public aspect SingletonInstance extends SingletonPattern {

        declare parents: MyClass implements Singleton;
        declare parents: SpecializedMyClass implements NonSingleton;
}
```

IBM : AOP@Work

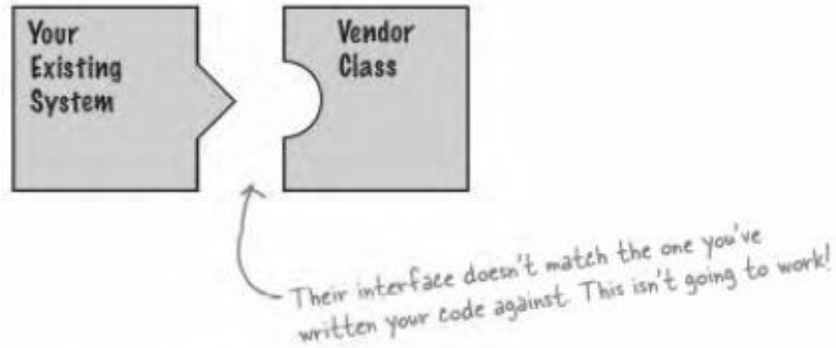
- Adapter & Decorator patterns

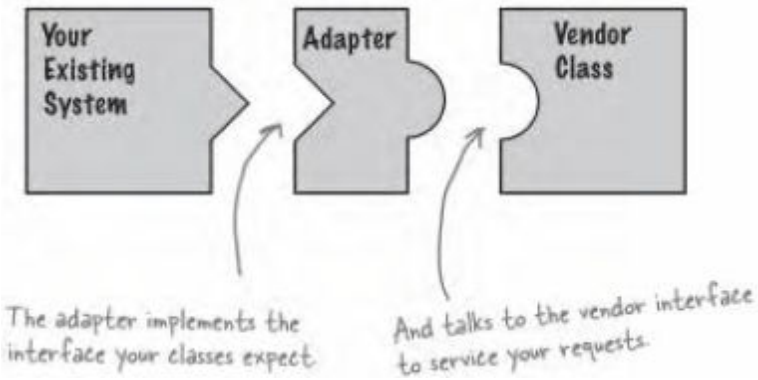
- http://www.ibm.com/developerworks/java/library/j-aopwork5/

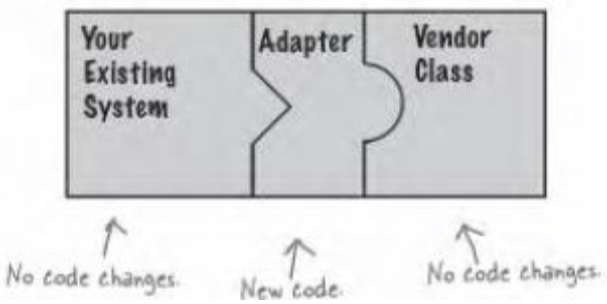
- Observer pattern

- http://www.ibm.com/developerworks/library/j-aopwork6/

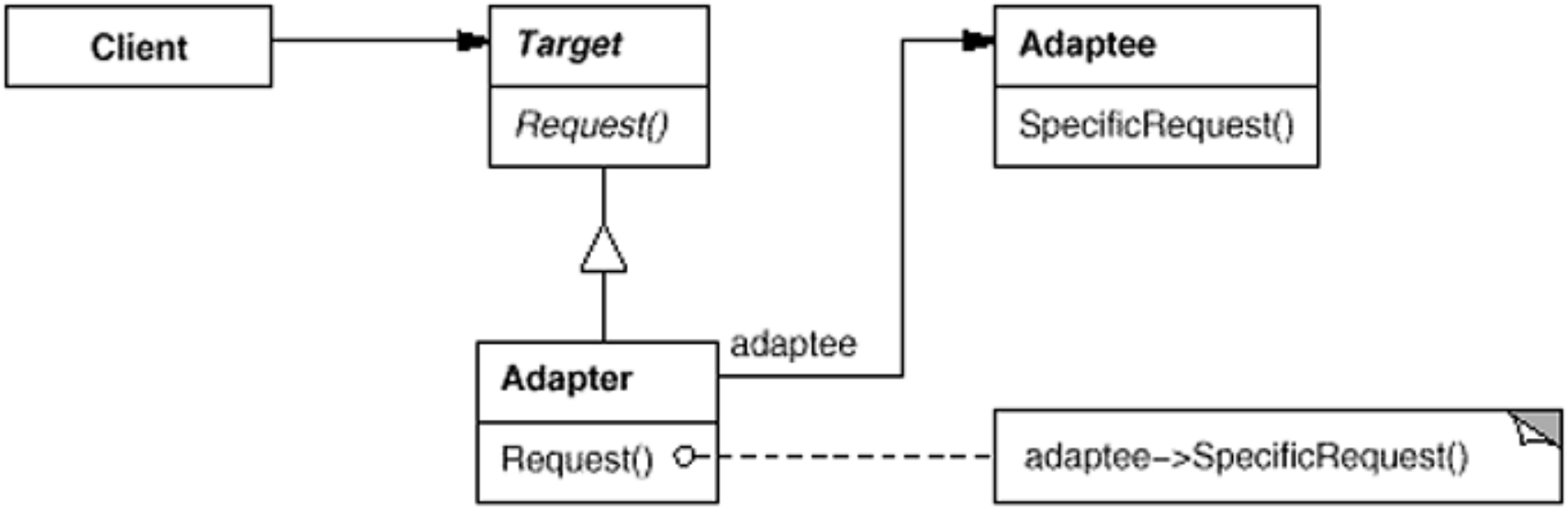
Adapter pattern







Adapter pattern



(version sans héritage multiple)

Adapter pattern

- Un vehicule spatial avec des capteurs non modifiables (bytecode)

```
public class TemperatureGauge {
  public int readTemperature(){
    //accesses sensor internals
  }
}
```

```
public class RadiationDetector {
  public double getCurrentRadiationLevel (){
    //read radiation somehow
  }
}
```

- On veut un rapport d'état de tous les capteurs

```
Readout:
Sensor 1 status is OK
Sensor 2 status is OK
Sensor 3 status is BORDERLINE
Sensor 4 status is DANGER!
```

Adapter pattern

- Implémentation désirée du rapport d'état

```
public static void main(String[] args){
    RadiationDetector radiationDetector = //find critical detector
    TemperatureGauge gauge = //get exhaust nozzle gauge

    List allSensors = new ArrayList();
    allSensors.add(radiationDetector);
    allSensors.add(gauge);

    int count = 1;
    for (Sensor sensor : allSensors) {
      //How to read each type of sensor...?
    }
  }
```

- Modifications sur les capteurs : comment ajouter une méthode **`getStatus()`** ?

```
if(this.readTemperature() > 160){
  return "DANGER";
}
return "OK"
```

Adapter pattern

- Interface commune aux capteurs (Target du pattern)

```
public interface StatusSensor {
  String getStatus();
}
```

- Implémentation du rapport d'état (Client du pattern)

```
for (StatusSensor sensor : allSensors) {
   System.out.print("Sensor " + count++);
   System.out.println(" status is " + sensor.getStatus());
 }
```

Adapter pattern

- Adapters et code glue du client

```
//Adapter classes
public class RadiationAdapter implements StatusSensor {
  private final RadiationDetector underlying;

  public RadiationAdapter(RadiationDetector radiationDetector) {
    this.underlying = radiationDetector;
  }

  public String getStatus() {
    if(underlying.getCurrentRadiationLevel() > 1.5){
      return "DANGER";
    }
    return "OK";
  }
}

public class TemperatureAdapter implements StatusSensor {
  //...similar
}
```

```
//glue code to wrap each sensor with its adapter...
allSensors.add(new RadiationAdapter(radiationDetector));
allSensors.add(new TemperatureAdapter(gauge));
```

Adapter pattern

- AspectJ Adapter : les capteurs implementent l'interface StatusSensor

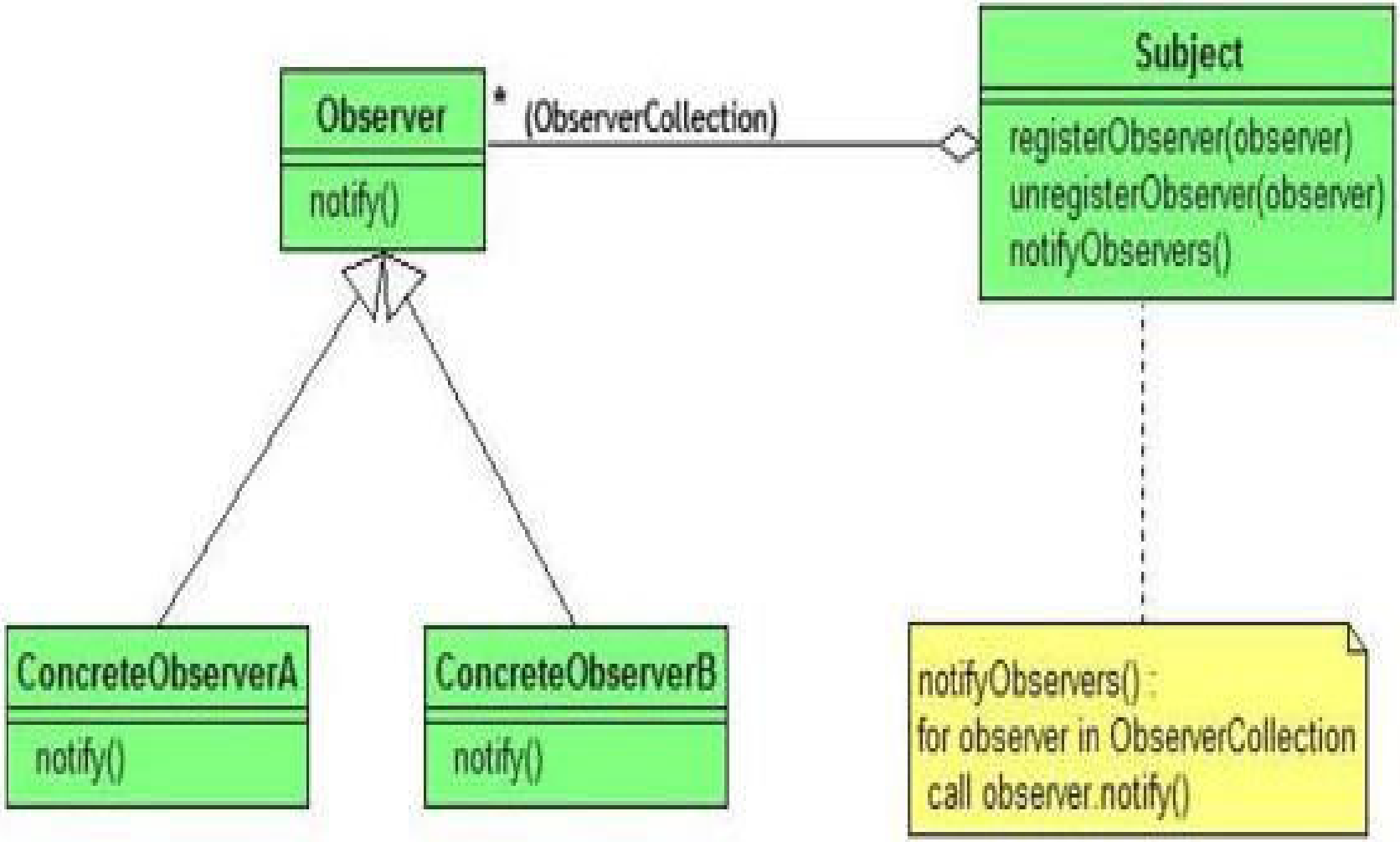
```
public aspect SensorAdapter {

  declare parents : TemperatureGauge implements StatusSensor;
  declare parents : RadiationDetector implements StatusSensor;

}
```

- AspectJ Adapter : les capteurs doivent implémenter les méthodes de l'interface StatusSensor (**`getStatus()`**)

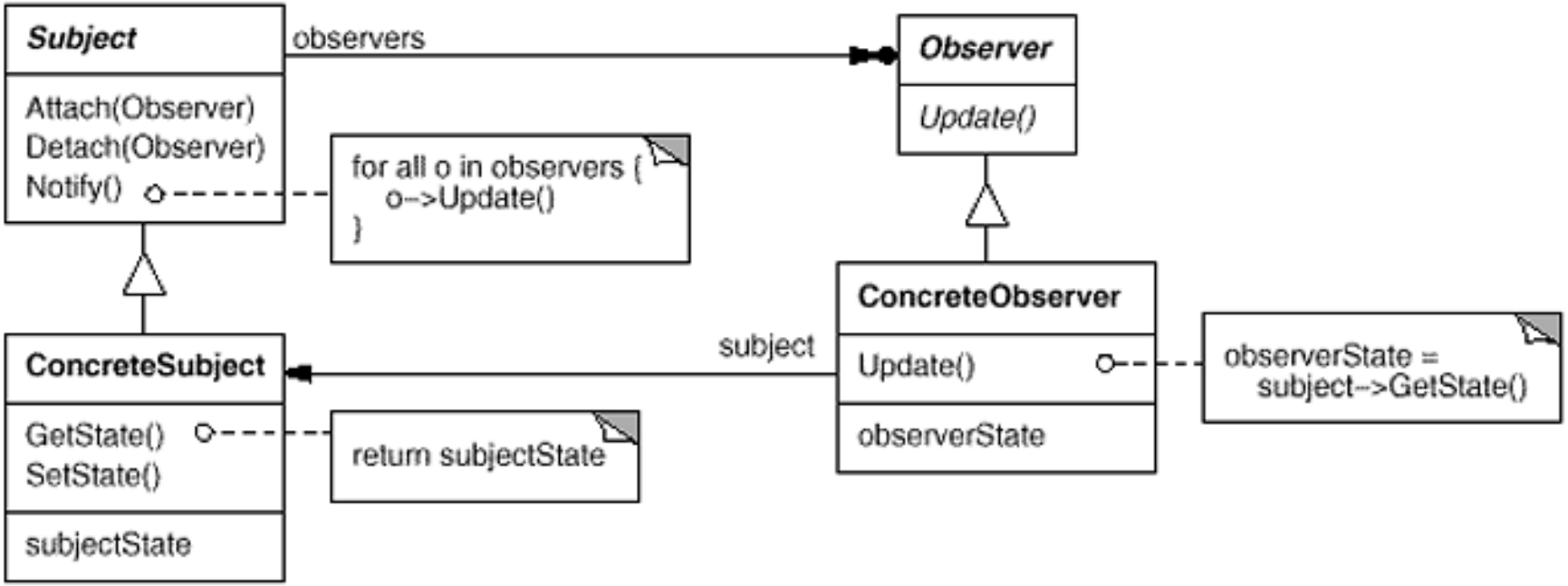
```
public String TemperatureGauge.getStatus() {
    if(this.readTemperature() > 160) {
      return "DANGER";
    }
    return "OK";
  }
```

- AspectJ Adapter : pas besoin de code glue,
  les capteurs sont directement adaptés

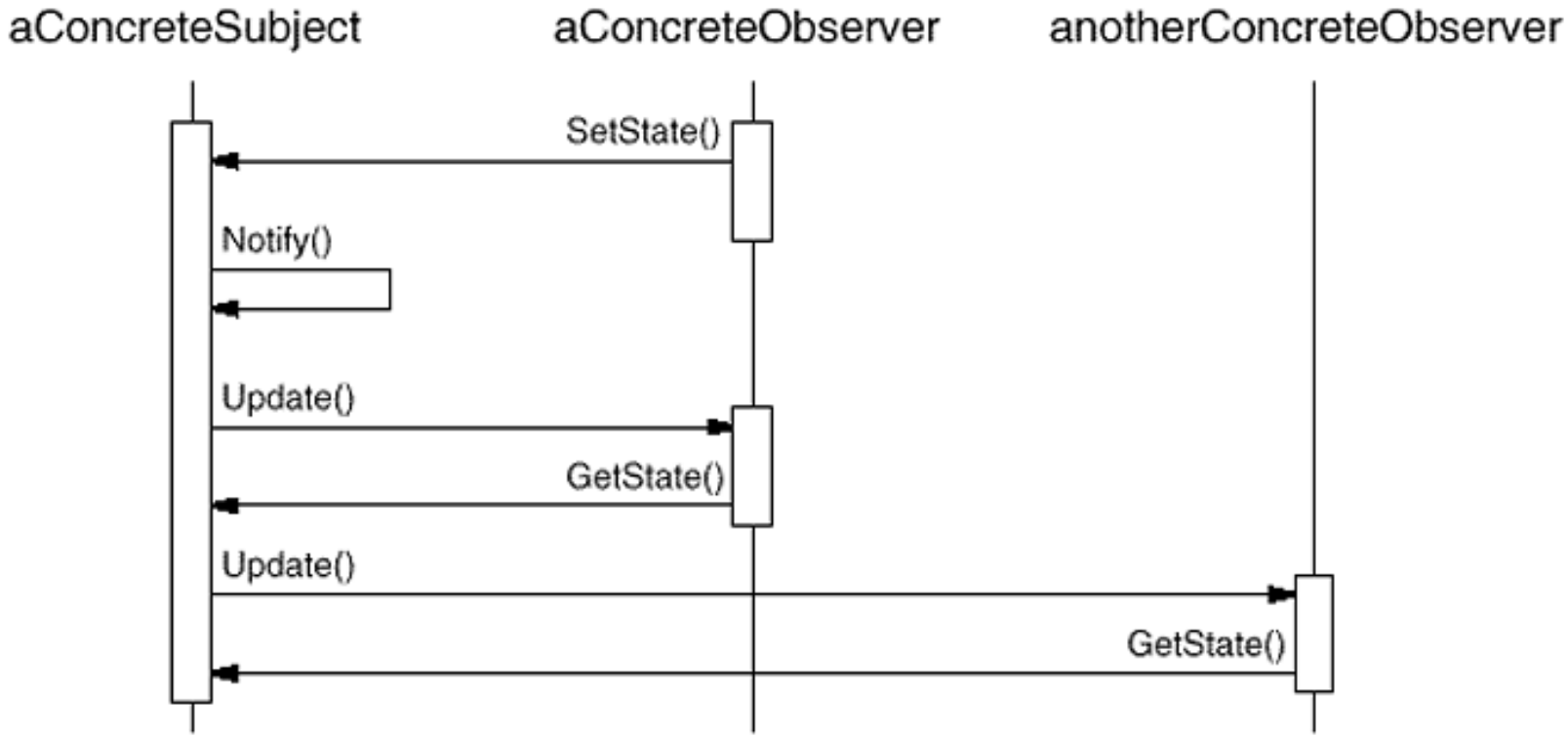
Observer pattern (cours COO)



Observer pattern (GoF)

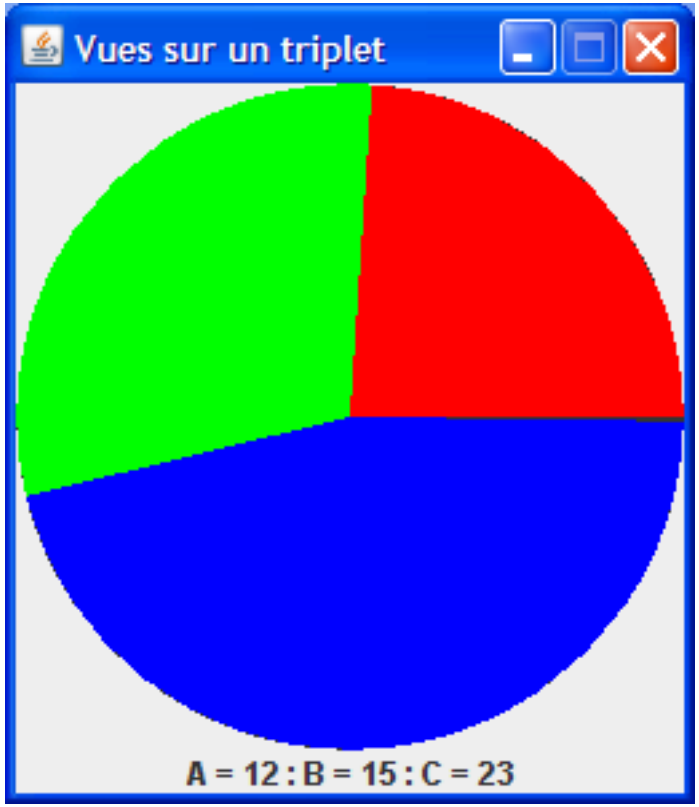


Observer pattern



Exemple : Triplet

- Participants
  - Subject / ConcreteSubject :
    - class Triplet
  - Observer :
    - interface Affichage
  - ConcreteObservers :
    - class AffichageCamembert
    - class AffichageValeurs



Triplet.java

```
package tripletObserver;

class Triplet {

   private int int_a;
   private int int_b;
   private int int_c;

   public Triplet(int int_pA, int int_pB, int int_pC) {
        int_a = int_pA; int_b = int_pB; int_c = int_pC;
   }

   void setA(int int_pA) { this.int_a = int_pA; /* ... */ }
   void setB(int int_pB) { this.int_b = int_pB; /* ... */ }
   void setC(int int_pC) { this.int_c = int_pC; /* ... */ }

   int getA() { return this.int_a; }
   int getB() { return this.int_b; }
   int getC() { return this.int_c; }

   /* Plus code du pattern Observer... */

}
```

Affichage.java

```
package tripletObserver;

interface Affichage {

  public void notifier();

}
```

AffichageCamembert.java

```
package tripletObserver;

class AffichageCamembert extends JPanel implements Affichage {
   private Triplet triplet;

   public AffichageCamembert(Triplet pTriplet) {
        /* init ... */
   }

   public void paint(Graphics g) {
        /* affichage des valeurs du Triplet sous forme de
   camembert... */
   }

   public void notifier() {
        // on force le redessin de la zone de dessin
        this.repaint();
   }
}
```

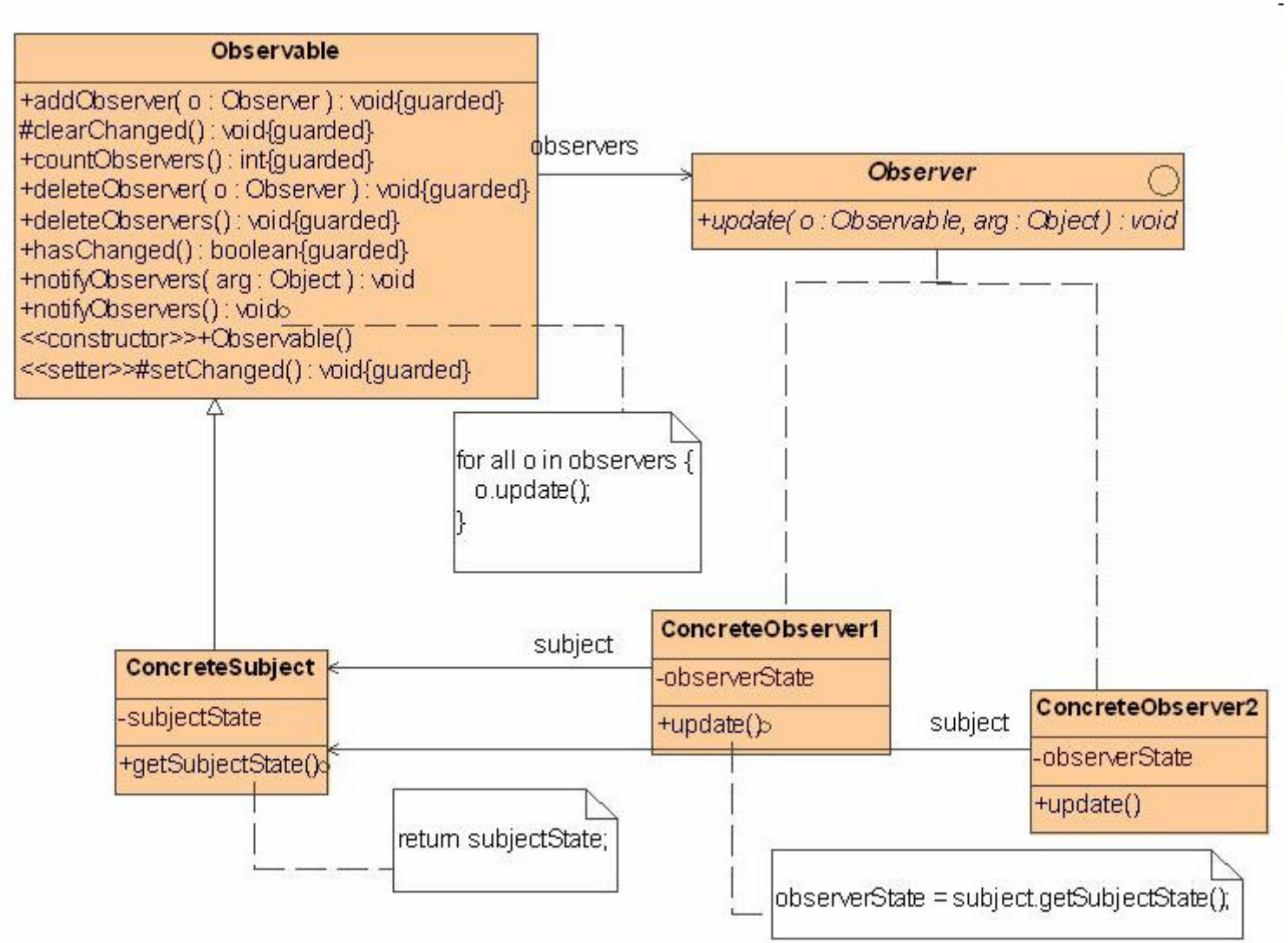
AffichageValeurs.java

```
package tripletObserver;

class AffichageValeurs extends JLabel implements Affichage {
   private Triplet triplet;

   public AffichageValeurs(Triplet pTriplet) {
        /* init... */
   }

   public void afficher() {
        /* affichage des valeurs du Triplet dans le label... */
   }

   public void notifier() {
        // on met à jour le texte dans le label
        this.afficher();
   }
}
```

Observer pattern avec Java

- Package java.util (cf. next slide)
  - interface Observer
  - class Observable
- Pour créer un observer
  - Implémenter l'interface Observer
  - Appeler addObserver() sur un objet Observable
- Pour créer un subject
  - Étendre la classe Observable
- Pour que le subject envoie les notifications
  - 1. appeler setChanged()
  - 2. appeler une des deux méthodes notifyObservers() :
    - PULL : notifyObservers()
    - PUSH : notifyObservers(Object arg)
- Pour que l'observer reçoive les notifications
  - Implémenter la méthode update(Observable o, Object arg)

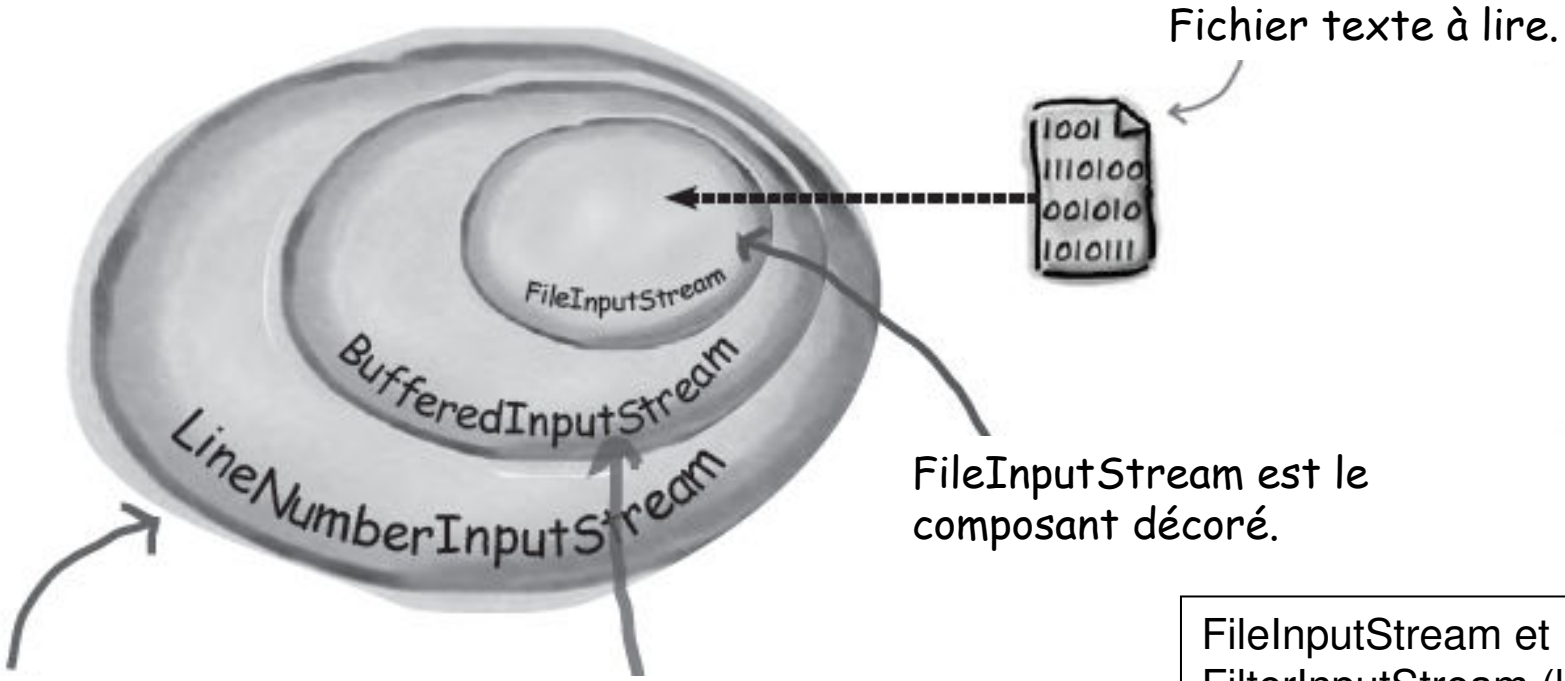
Observer pattern avec Java



Devoir à rendre

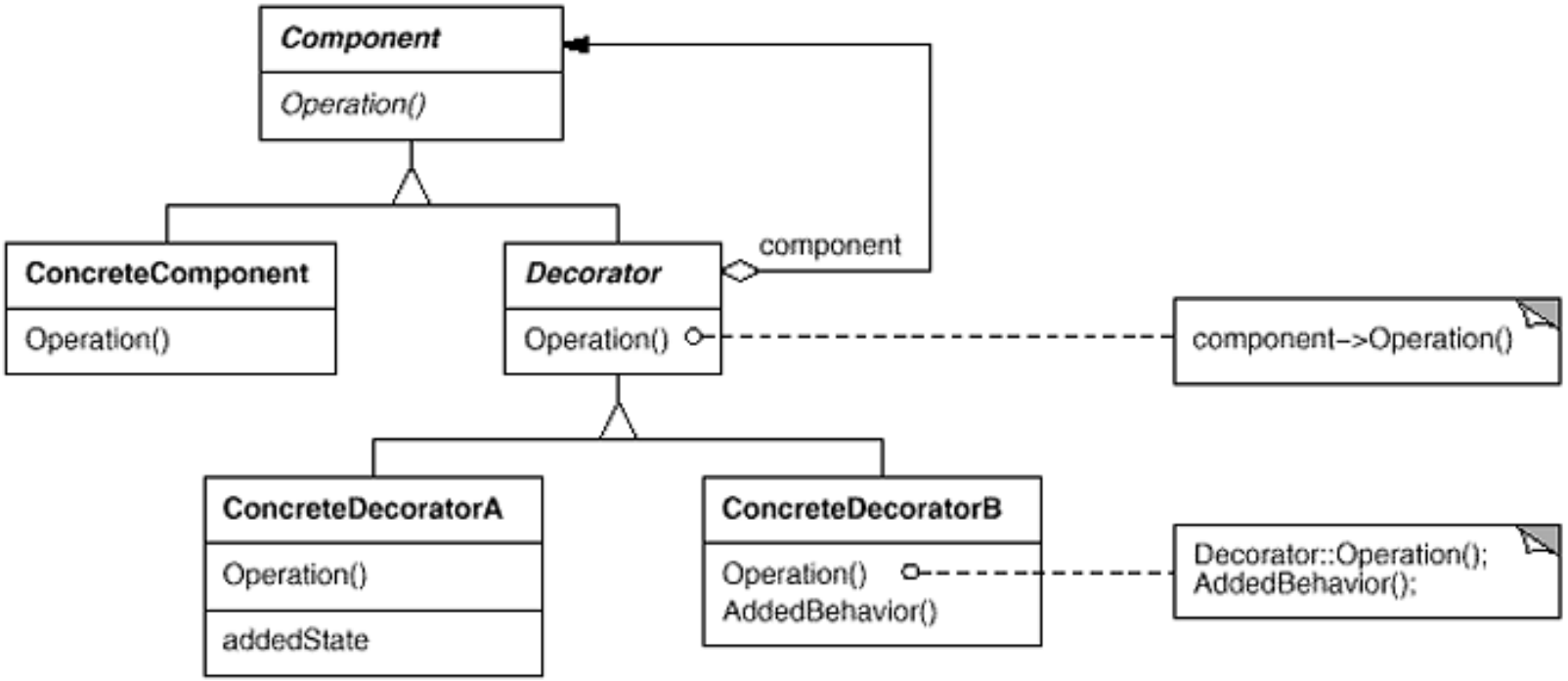
- A partir de l'application Triplet, réalisez une version aspectisée du pattern Observer, suivant:
  - l'implémentation proposée en cours de COO
  - l'implémentation fournie dans java.util
- Utilisez sur l'application Triplet, la version « générique » du pattern Observer fournie avec AspectJ (utilisation des aspects abstraits)
  - Voir Project/AspectJ/AspectJ Examples/Observer Example
- Modifiez la version « générique » d'AspectJ pour réaliser une implémentation utilisant java.util et fonctionnant sur l'application Triplet

Decorator pattern



LineNumberInputStream est un décorateur concret. Il ajoute la possibilité de compter le nombre de lignes lues.

BufferedInputStream est un décorateur concret. Il augmente le comportement en ajoutant un buffer et la méthode readLine().

FileInputStream et FilterInputStream (le décorateur abstrait) étendent la classe InputStream (le composant abstrait). BufferedInputStream et LineNumberInputStream (les décorateurs concrets) étendent FilterInputStream.

Decorator pattern