Design patterns

UNIVERSITÉ DE LORRAINE

Conception Architectures Distribuées

Horatiu CIRSTEA

Devenir un champion aux échecs

- **Apprendre les règles**
  - le nom des pièces, les mouvements permis, la géométrie de l'échiquier, ...
- **Apprendre les principes**
  - la valeur relative de certaines pièces, la valeur stratégique des emplacements centraux, ...
- **Etudier le jeu d'autres champions**
  - ces jeux contiennent des patterns : il faut les comprendre, les mémoriser, et les appliquer de manière répétée.

Devenir un champion du développement

- **Apprendre les règles**
  - les algorithmes, les structures de données, UML, les langages de programmation, ...
- **Apprendre les principes**
  - la conception UML, la programmation orientée-objet, la programmation générique, ...
- **Etudier la conception d'autres champions**
  - ces conceptions contiennent des patterns : il faut les comprendre, les mémoriser, et les appliquer de manière répétée.

La conception OO

- Il est extrêmement difficile de réaliser une conception
  - réutilisable, extensible, adaptable, performante
- Novice vs. concepteur expérimenté ?
  - le novice hésite beaucoup entre différentes variantes
  - l'expert trouve tout de suite la bonne solution.
- Quel est le secret ?
  - l'EXPERIENCE !

Design patterns

- L'expérience
  - ne pas réinventer la roue
  - réutiliser systématiquement des solutions qui ont fait leurs preuves
- Répétition de certains profils de classes ou collaboration d'objets

design patterns
= patrons de conception
= modèles de conception

Design patterns

Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander, [AIS+77]

Design patterns

- **Christopher Alexander, 1977 :**

  Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

- **Une solution à un problème dans un contexte.**
- **Le catalogue de la bande des quatre (GoF) :**
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995)

Historique

- 1987 - Cunningham et Beck utilisent les idées d'Alexander pour développer un petit langage de patrons pour Smalltalk
- 1990 – Le « Gang of Four » commence à travailler à la compilation d'un catalogue de patrons de conceptions
- 1993 - Beck et Booch sponsorisent la première réunion du groupe Hillside (`www.hillside.net`)
- 1995 - Le « Gang of Four » publie le livre des Patrons de conception

Types de patrons logiciels*

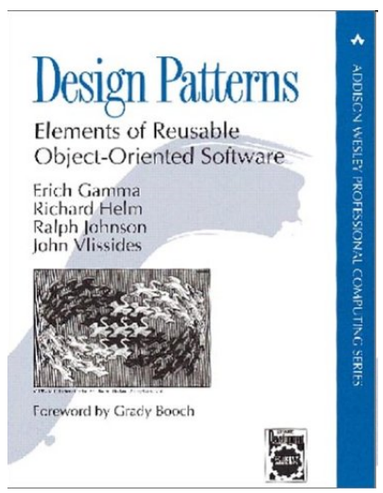
- **Patrons conceptuels :** la forme est décrite par les termes et concepts du domaine d'application
- **Patrons de conception :** la forme est décrite par les éléments de construction de conception logicielle
- **Patrons de programmation :** la forme est décrite par les éléments de construction du langage de programmation

* "Understanding and Using Patterns in Software Development" (Riehle et Zullighoven, 1996)

Patrons de conception

« Descriptions d'objets et de classes communicantes qui sont adaptées à la résolution d'un problème général de conception dans un contexte particulier »

Design Patterns, GoF, 1995
23 patrons de conception
3 catégories



Un bon patron de conception

- il résout un problème
- c'est un concept éprouvé
- la solution n'est pas évidente
- il décrit une collaboration entre objets
- il a une composante humaine : utilité, esthétique
- éléments : nom, problème, solution, conséquences

Catégories

- ***Patrons de création***
  - concernent le processus de création d'objets
  - aident à créer des objets pour vous, au lieu d’avoir à instancier les objets directement

Catégories

- ***Patrons de structure***
  - concernent la composition de classes et d'objets
  - aident à composer des groupes d’objets en des structures plus larges, telles que des interfaces utilisateur complexes

Catégories

- ***Patrons de comportement***
  - concernent l'interaction des classes et des objets
  - aident à définir la communication entre les objets du système et définir comment le flux est contrôlé

Classification GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Scope : domaine d'application du pattern

Purpose : l'objectif du pattern

Template

- **Nom et classification du patron**
  - l'essence du pattern
  - un vocabulaire de référence
- **Intention**
  - but, raison d'être
  - cas particuliers de conception concernés
- **Alias**
  - autres noms du patron
- **Motivation**
  - scénario qui illustre un cas de conception et montre comment le patron contribue à la solution

Template

- **Applicabilité**
  - cas qui justifient son utilisation
  - situations peu satisfaisantes qui tirent avantage de son utilisation
- **Structure**
  - modèle de classe
  - (modèle de collaboration)
- **Participants**
  - les classes et objets participant au patron (rôles)

Template

- **Collaborations**
  - comment les participants interagissent pour gérer leurs responsabilités?
- **Conséquences**
  - impacts sur l'architecture de conception
  - degrés de liberté laissés par le patron, compromis éventuels
- **Implémentation**
  - techniques particulières, spécificités du langage

Template

- **Exemples de code**
  - extraits de programmes en Java, C++, ...
- **Utilisations connues**
  - exemples du patron dans des systèmes réels
- **Patrons liés**
  - patrons en étroite relation
  - différences importantes
  - utilisation conjointe d'autres patrons

Bénéfices

- Capturent l'expertise et la rendent accessible à des non-experts
- Réduisent le temps de développement
- Facilitent la communication entre les développeurs en fournissant un langage commun
- Facilitent la réutilisation réussie de conceptions
- Améliorent la documentation de conception
- Améliorent la compréhensibilité des conceptions

Inconvénients

- Les patrons sont simplistes, mais ils doivent être appliqués à votre problème
- Utiliser des patrons requiert la capacité de faire le lien entre un patron et votre problème
- Les patrons sont validés par l'expérience, plutôt que par des tests automatiques

Observer

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Problème

- Comment un (objet) client peut collecter des informations provenant d'un autre objet?

Solution 1

`getSpeed()`

Avec quelle fréquence ?

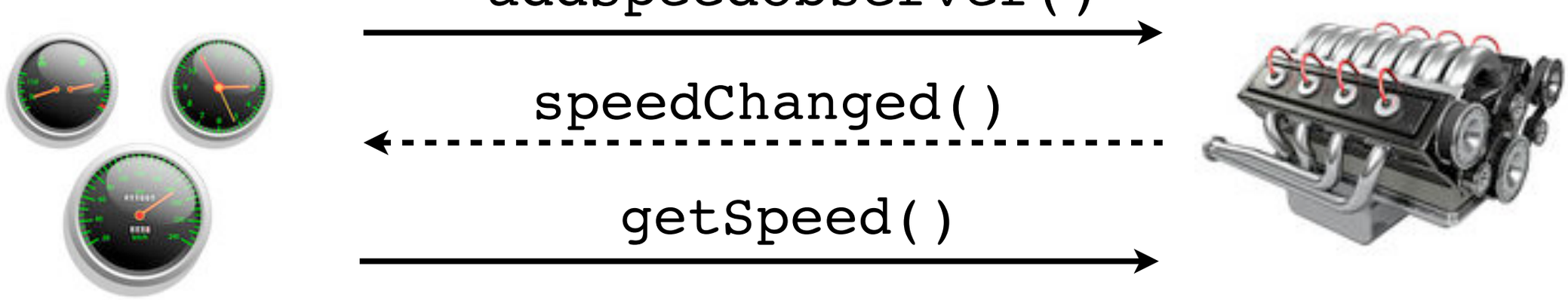
Solution 2

`setSpeedIndicator()`

KM ou Miles ?

distance parcourue aussi ?

Solution 3



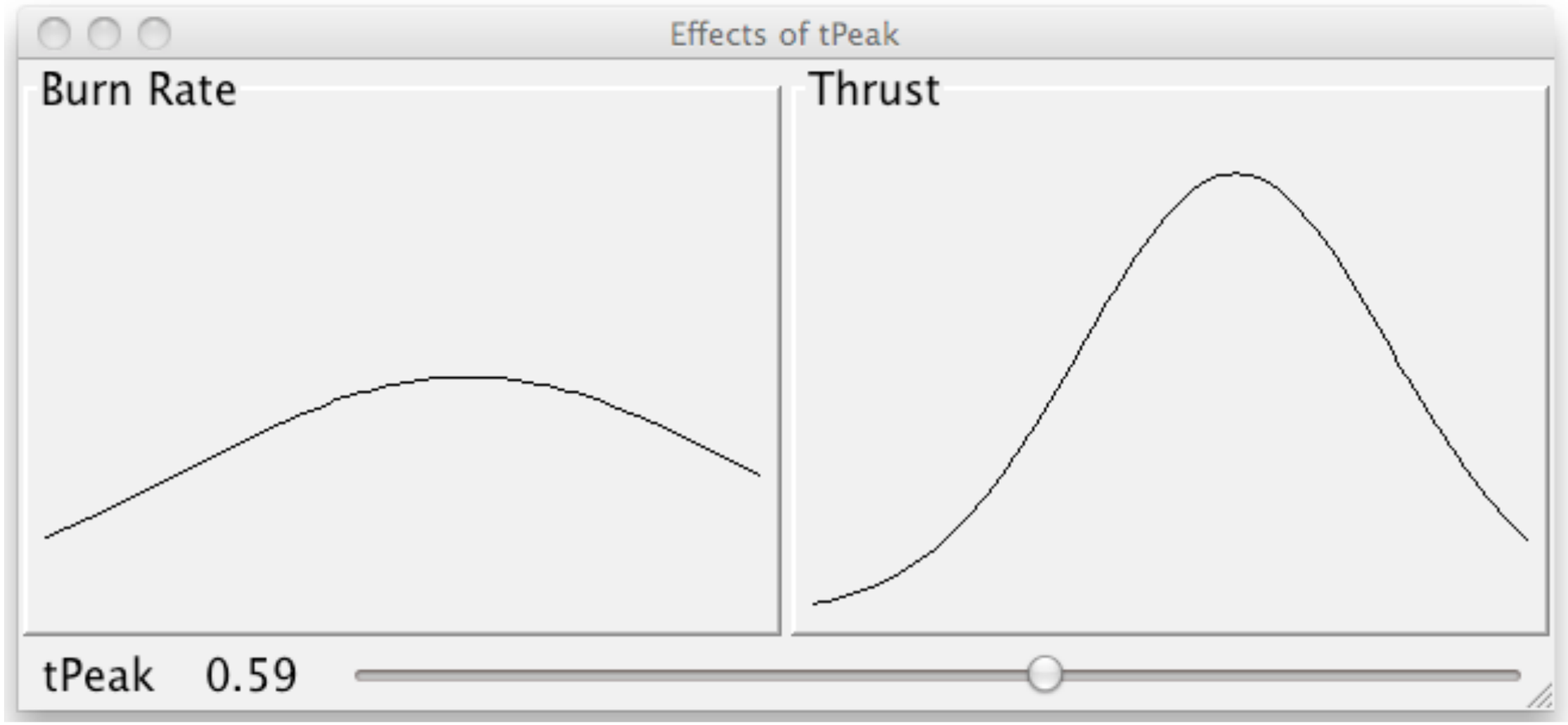
Le client modifie son état en fonction des informations reçues

Observer

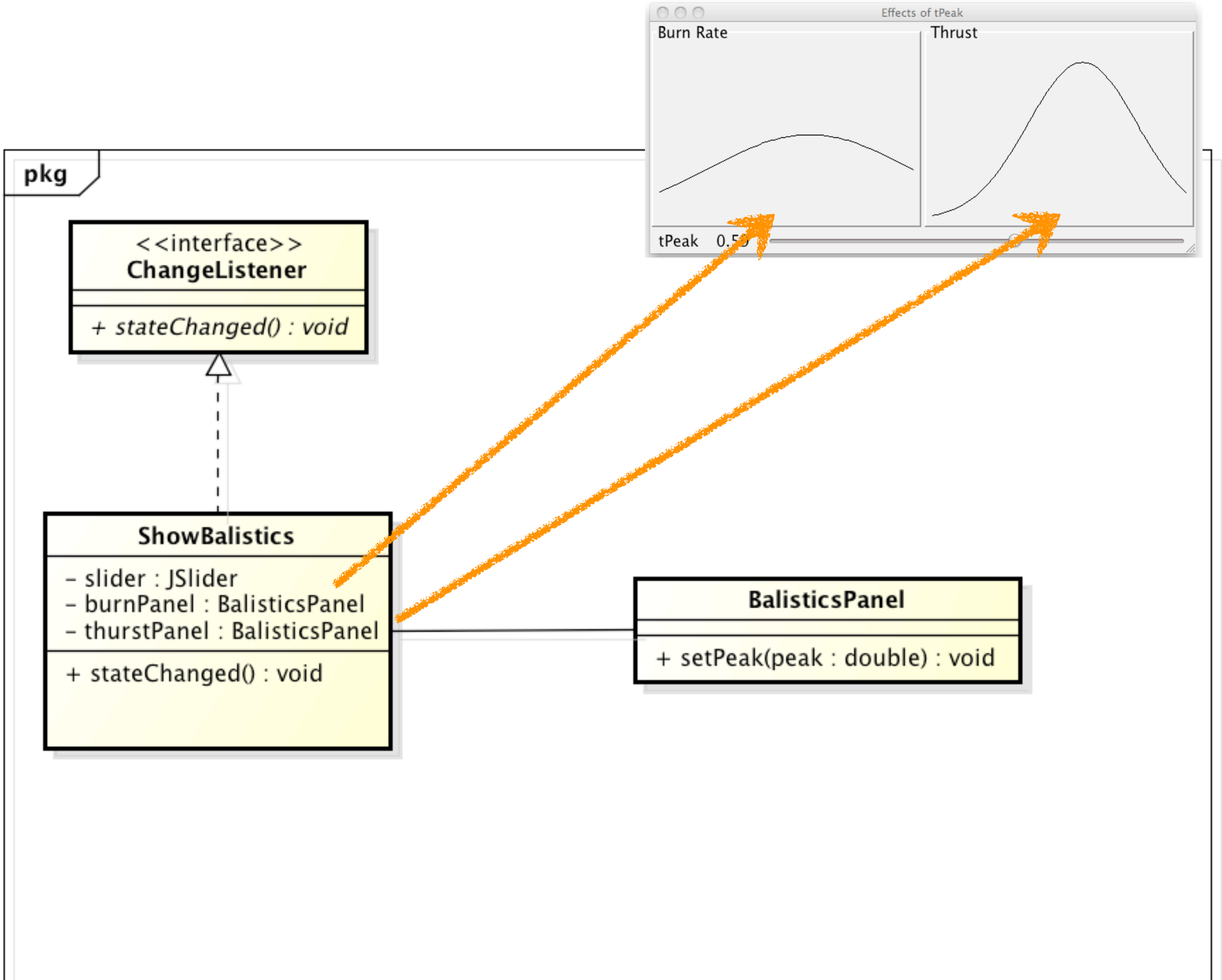
Intention

- Disposer d'un mécanisme de notification qui fait que, quand un objet change d'état (le sujet), tous ceux qui en dépendent sont automatiquement mis à jour (les observateurs).
- Encapsuler le composant principal dans une abstraction Subject, et les composants variables (ou optionnelles, ou UI) dans une hiérarchie de Observers.

Exemple*



*exemple inspiré de Oozinoz (S.J.Metsker)


Effects of tPeak
Burn Rate
Thrust
tPeak
0.5
pkg
<<interface>>
ChangeListener
+ stateChanged() : void
ShowBalistics
- slider : JSlider
- burnPanel : BalisticsPanel
- thurstPanel : BalisticsPanel
+ stateChanged() : void
BalisticsPanel
+ setPeak(peak : double) : void

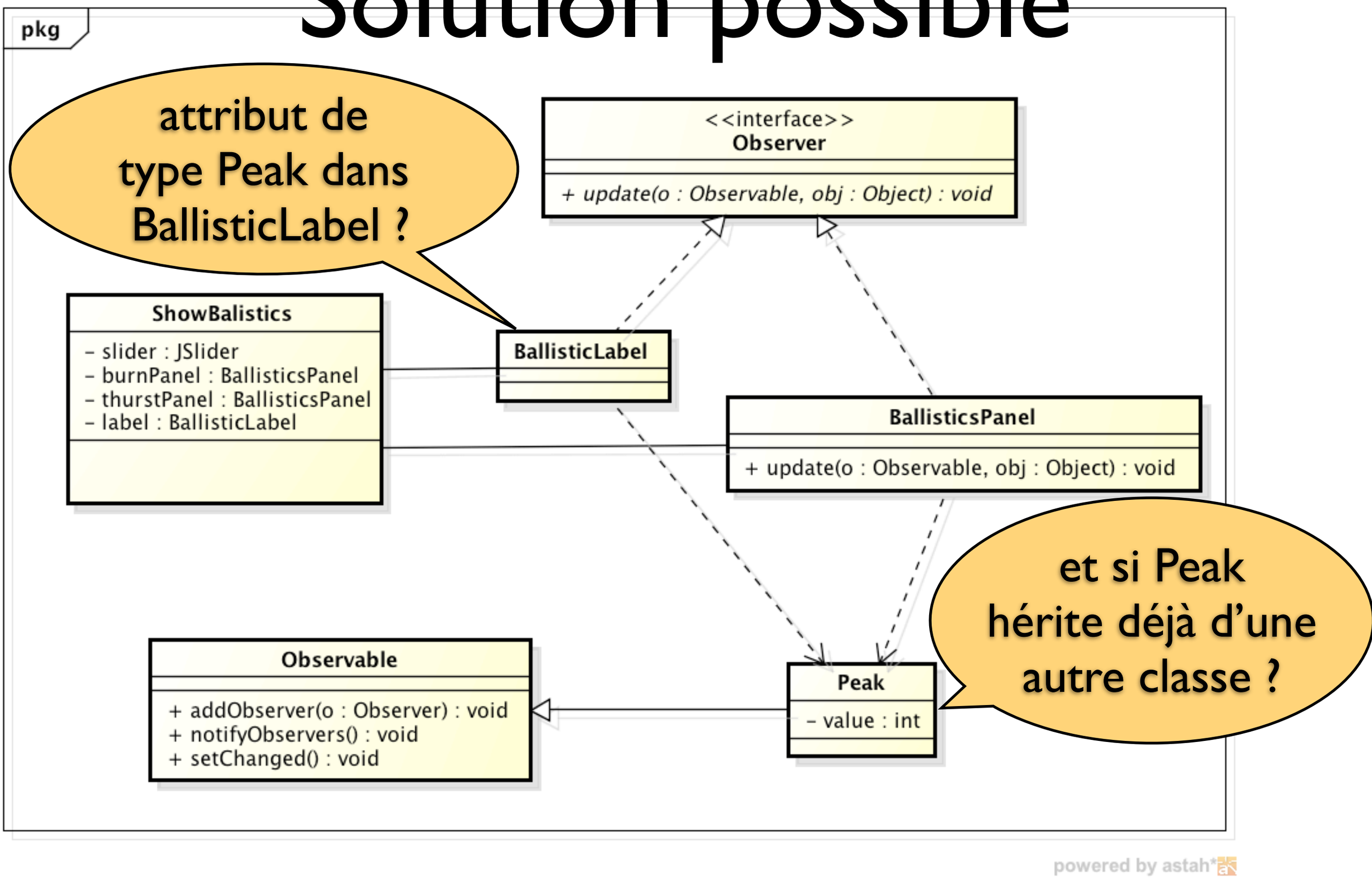
hard-coded

```
public JSlider getSlider(){
   if (slider == null){
      slider = new JSlider();
      slider.addChangeListener(this);
   }
   return slider;
}
```

calculé une fois

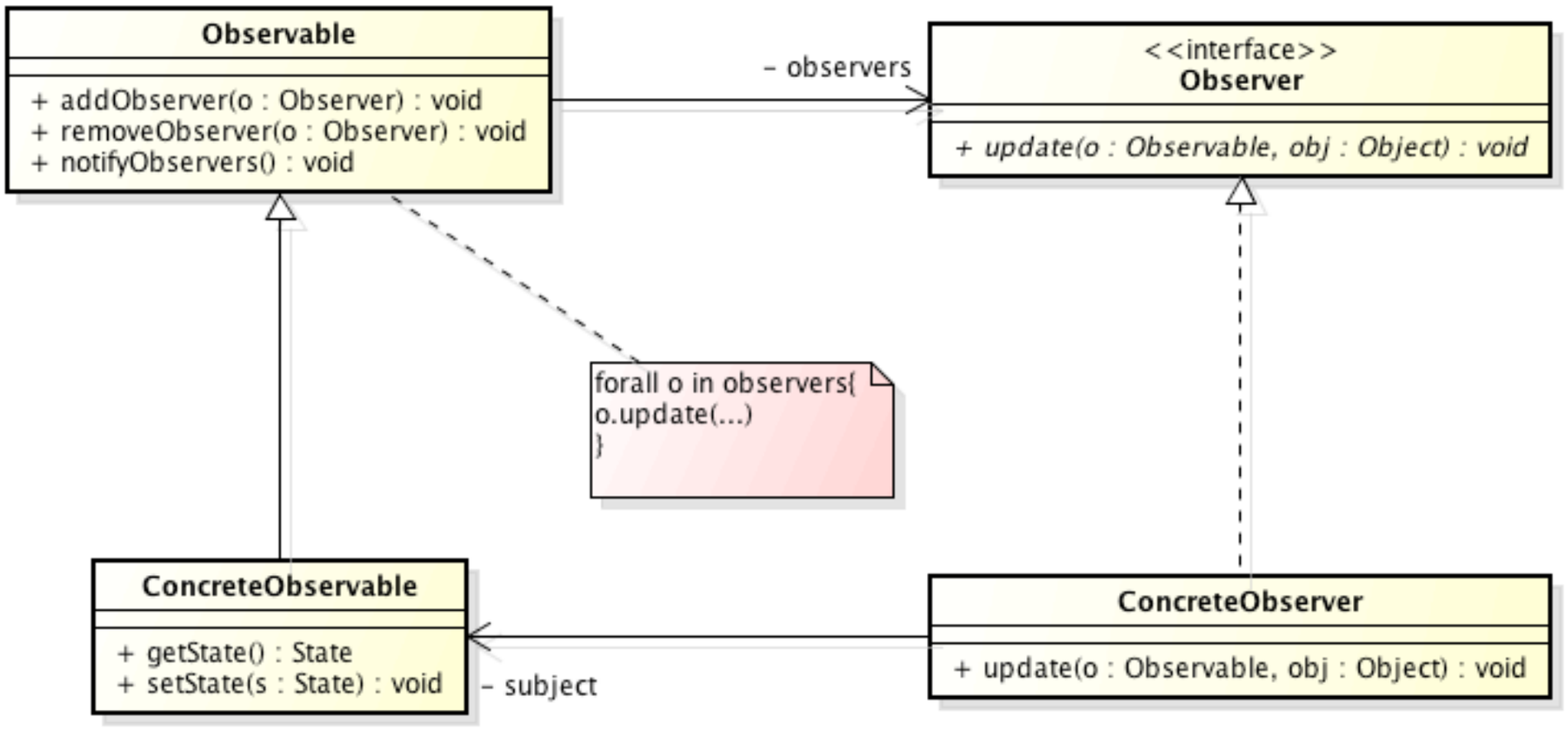
```
public void stateChanged() {
   double val = comp(slider.getValue());
   burnPanel().setTPeak(val);
   thrustPanel().setTPeak(val);
}
```

Solution possible



Observer

Structure



Observer

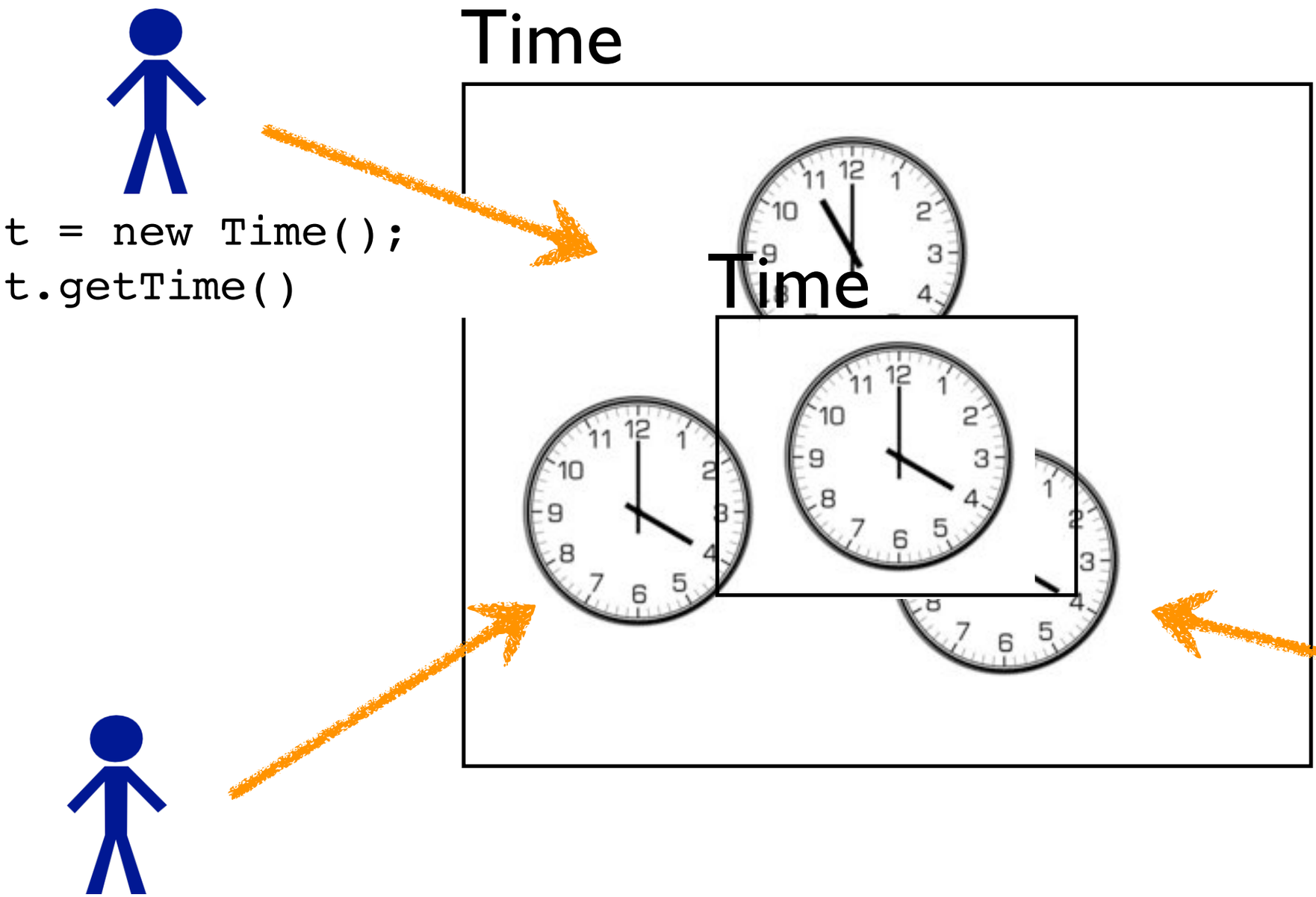
Indications d'utilisation

- une abstraction a plusieurs représentations non indépendantes
- une modification d'un objet nécessite la mise à jour d'autres objets
- un objet doit faire une notification à d'autres sans faire d'hypothèse sur la nature de ceux-ci

Singleton

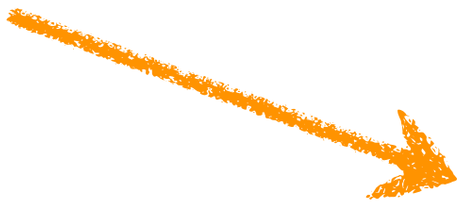
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Problème


Time
Time
t = new Time();
t.getTime()
t = new Time();
t.getTime()
t = new Time();
t.getTime()

Problème

```
t = ?;
t.getTime()
```



Time


```
t = ?;
t.getTime()
```

```
t = ?;
t.getTime()
```

Problème

- Comment identifier un objet (unique) avec des responsabilités concernant potentiellement un ensemble d'objets ?

Exemple : Window Manager, Print Server, point d'accès à une BDD, ...

Singleton

Intention

- Garantir qu'une classe n'a qu'une seule instance et fournir un point d'accès global à cette instance
- encapsulation de l'initialisation « just-in-time » ou « on first use »
- ressemblance avec une « variable globale »

Singleton

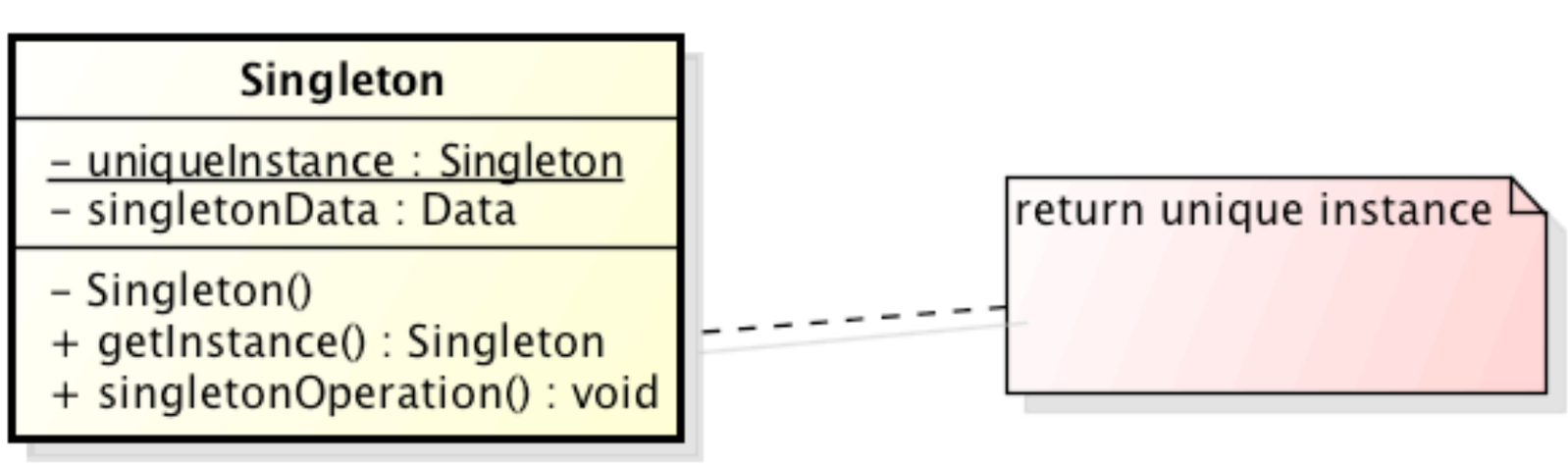
Indications d'utilisation

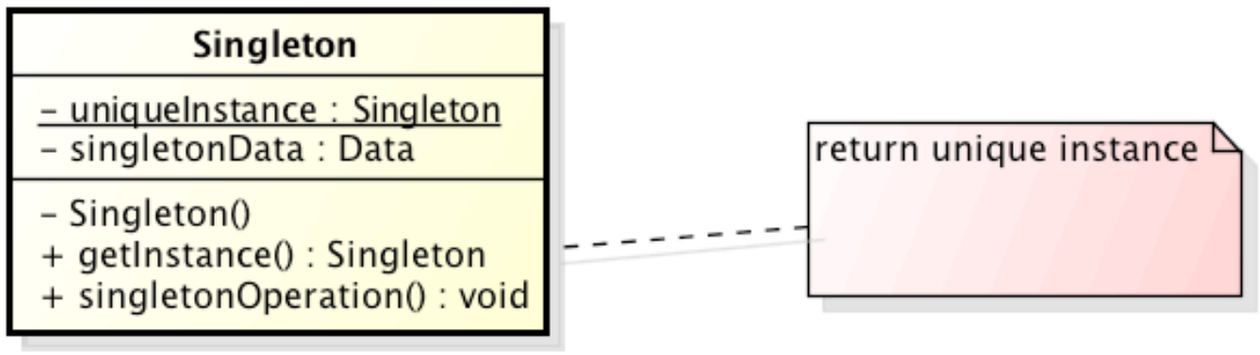
- il doit y avoir exactement une instance d'une classe et cette instance est accessible globalement
- l'instance unique doit être extensible par héritage et les clients doivent pouvoir utiliser cette instance étendue sans modifier leur code

Singleton

pkg

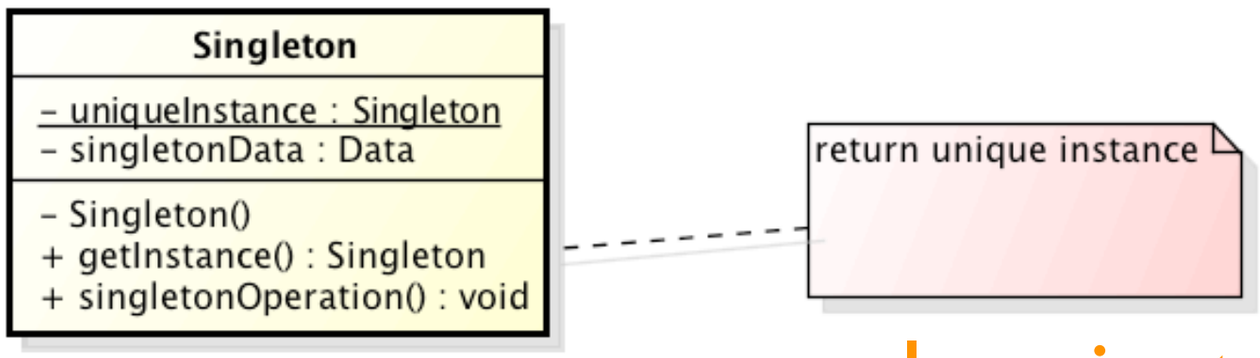
Structure





eager instantiation

```
public class Singleton {
   private static Singleton unique = new Singleton();
   private Singleton() { singletonData = 3; };
   private static int singletonData;
   public static Singleton getInstance(){
      return unique;
   }
   // getter & setter
}
```



lazy instantiation

```
public class Singleton {
   private static Singleton unique = null;
   private Singleton() { singletonData = 3; };
   private static int singletonData;
   public static Singleton getInstance(){
      if (unique == null)
            unique = new Singleton();
      return unique;
   }
   // getter & setter
}
```

```
public class Singleton {
   private static Singleton unique = null;
   private Singleton() { singletonData = 3; };
   private static int singletonData;
   public static Singleton getInstance(){
      if (unique == null)
            unique = new Singleton();
      return unique;
   }
   // getter & setter
}
```

thread-safe ?

```
public class Singleton {
   private volatile static Singleton unique = null;
   private Singleton() { singletonData = 3; };
   private static int singletonData;
   public static Singleton getInstance(){
      if (unique == null)
         synchronized (Singleton.class) {
            unique = new Singleton();
         }
      }
      return unique;
   }
   // getter & setter
}
```

T1 et T2 arrivent en même temps

⇓

2 instances

```
public class Singleton {
  private volatile static Singleton unique = null;
  private Singleton() { singletonData = 3; };
  private static int singletonData;
  public static Singleton getInstance(){
    if (unique == null)
      synchronized (Singleton.class) {
            if (unique == null){
              unique = new Singleton();
            }
      }
    }
    return unique;
  }
  // getter & setter
}
```

double-checked locking

Initialization-on-demand holder idiom

```
public class Singleton {
  private Singleton() { singletonData = 3; };
  private static int singletonData;

  private static class LazyHolder {
     private static final Singleton unique =
                                        new Singleton();
  }
  public static Singleton getInstance(){
    return LazyHolder.unique;
  }
  ...
}
```

Plusieurs instances

```
public class SinglePool {
   private static SinglePool[] instances =
                                    new SinglePool[5];
   private static int count = -1;
   private SinglePool() { ... };
   private static int singletonData;
   public static SinglePool getInstance(){
      count++;
      if(instances[count % instances.length] == null))
         instances[count % instances.length]
                  = new Singleton();
      return instances[count % instances.length];
   }
}
```

Quiz

- Singleton ou non ?
  - BestStudent
  - java.lang.Math
  - TopManager
  - File Manager
  - Print Spooler

Résumé Singleton

- A utiliser quand
  - le propriétaire d'un Singleton ne peut pas être identifié clairement
  - l'accès global n'est pas fourni d'une autre manière

Homework:

Comment sous-classer un Singleton?

Questions ?

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor