# Architecture Logicielle

# Les patrons de responsabilités

Florent Nicart

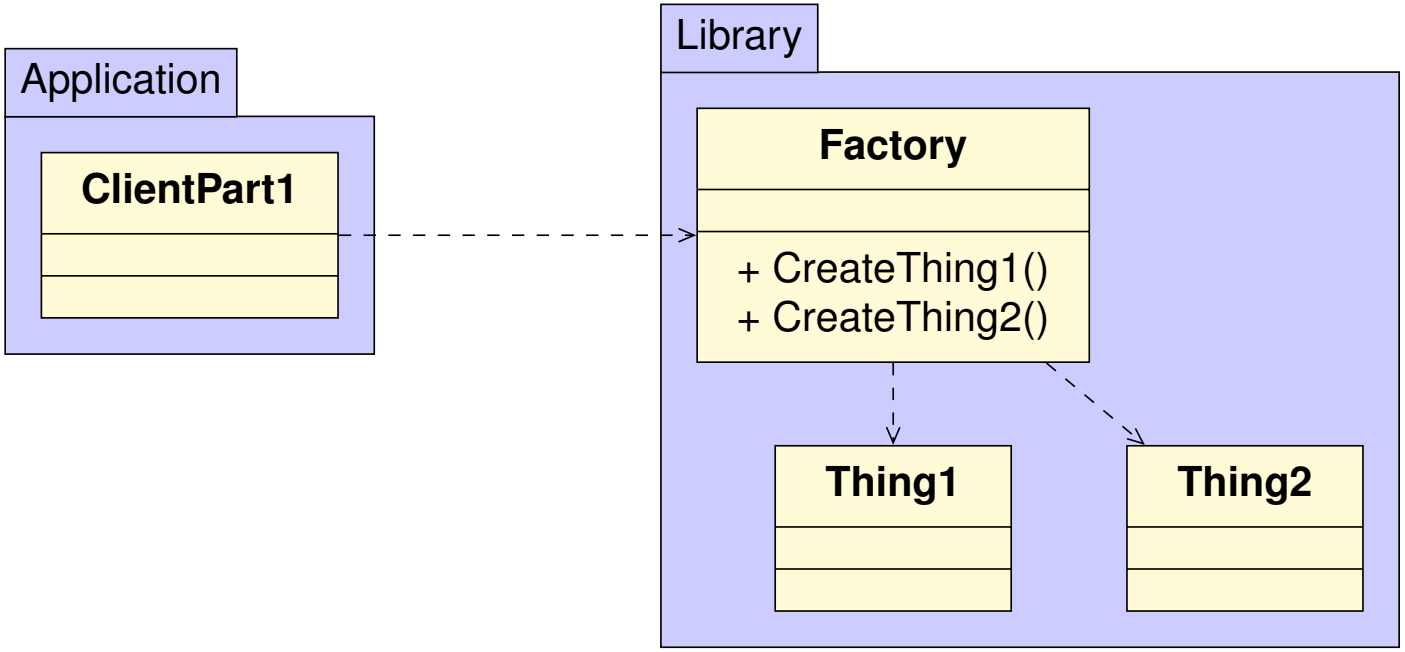
Université de Rouen

*2017–2018*

# Le patron Singleton

Fournir une (hiérarchie de) classe(s) admettant au plus une instance.

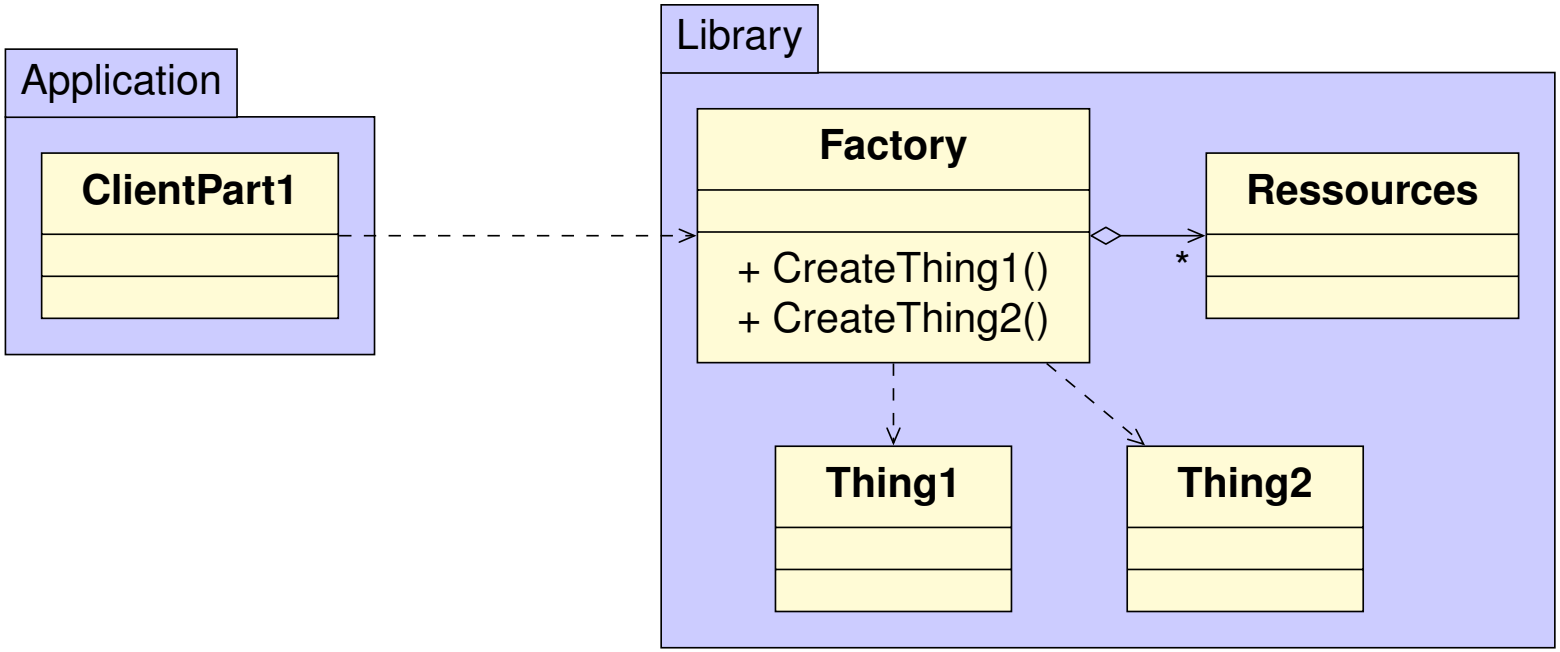
# Situation Initiale




- Une application utilise un service d'une bibliothèque, ici pour produire des objets [1].
- Ce service pourrait-être rendu par des méthodes statiques...

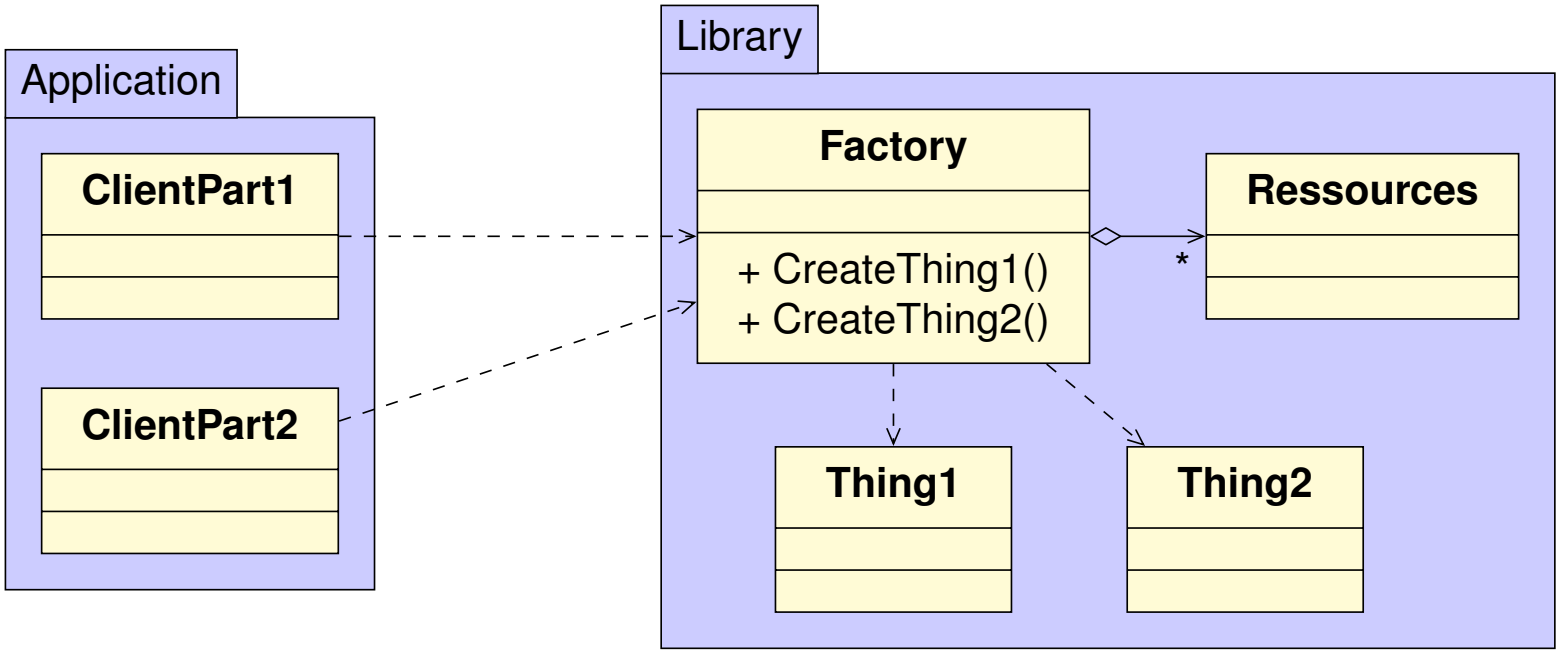
1. Nous verrons les usines au chapitre suivant.

# Situation Initiale



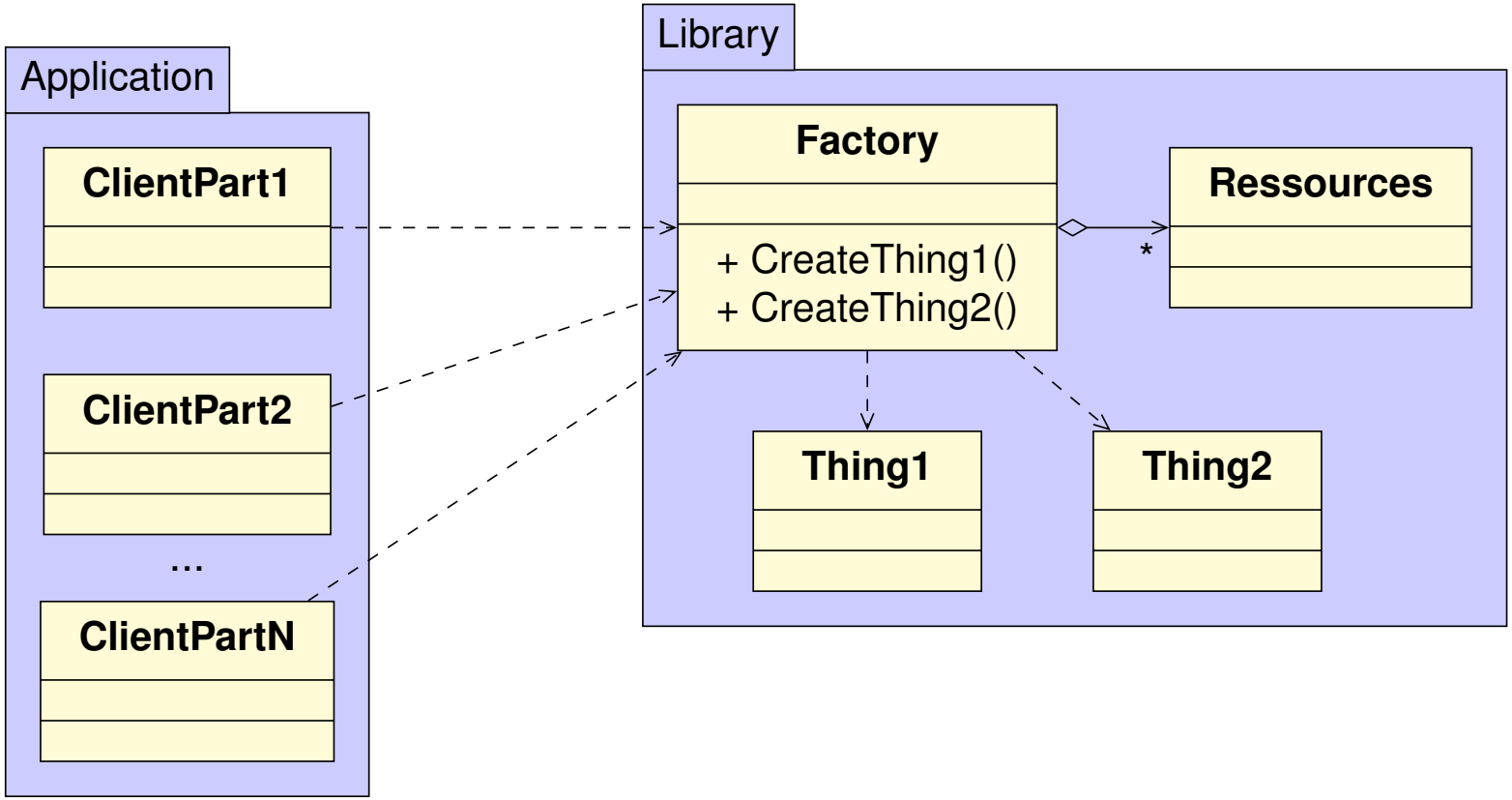

- ... sauf que ce service requiert des ressources allouées dynamiquement.
- C'est donc bien sur une instance que nous souhaitons travailler.
- Ou bien la nature de cette instance peut varier dynamiquement (voir plus loin).

# Situation Initiale



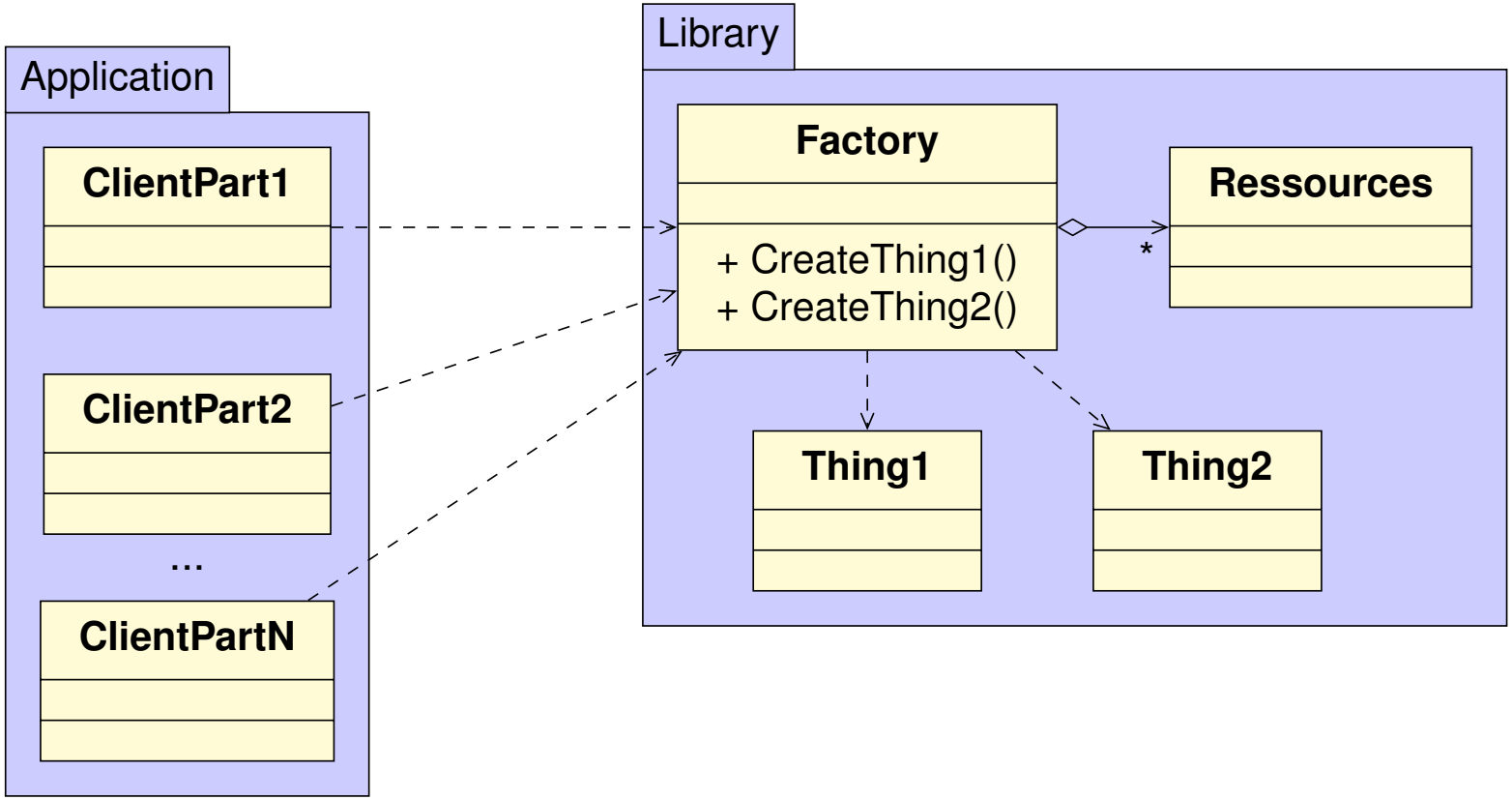

- En général, ces services sont utilisés en plusieurs endroits du code client.

# Situation Initiale




- En général, ces services sont utilisés en plusieurs endroits du code client.
- En général, à beaucoup d'endroits.
- Parfois répartis sur plusieurs autres bibliothèques

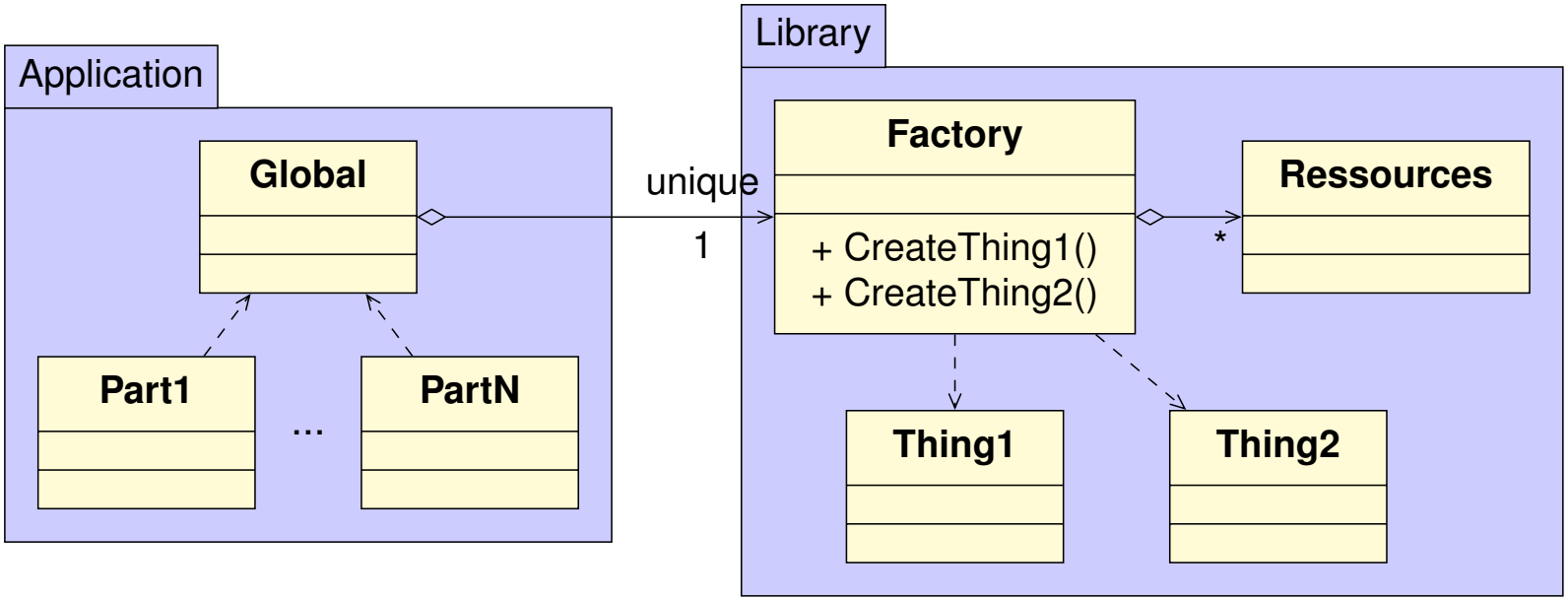
# Situation Initiale




- Des instanciations multiples sont inutiles (gaspillage en espace et en temps),
- et peuvent conduire à travailler sur des espaces de ressources différents (disfonctionnements)

# Une mauvaise solution

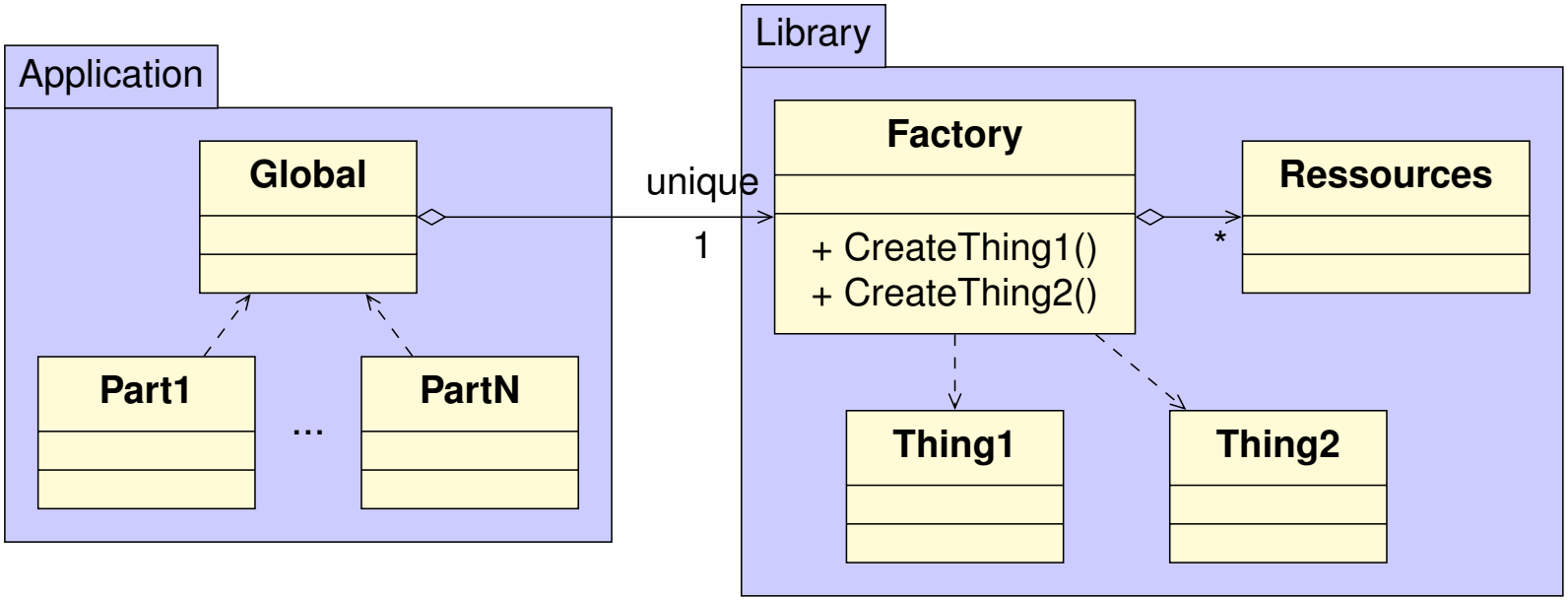
## Gérer le problème au niveau du client




- Problème : le client implémente une solution à un problème imposé par la bibliothèque !
- et ne résout pas le problème pour les biliotèques tierces,
- Solution : faire en sorte que la bibliothèque garantisse elle-même l'unicité grâce au patron **singleton**.

# Une mauvaise solution

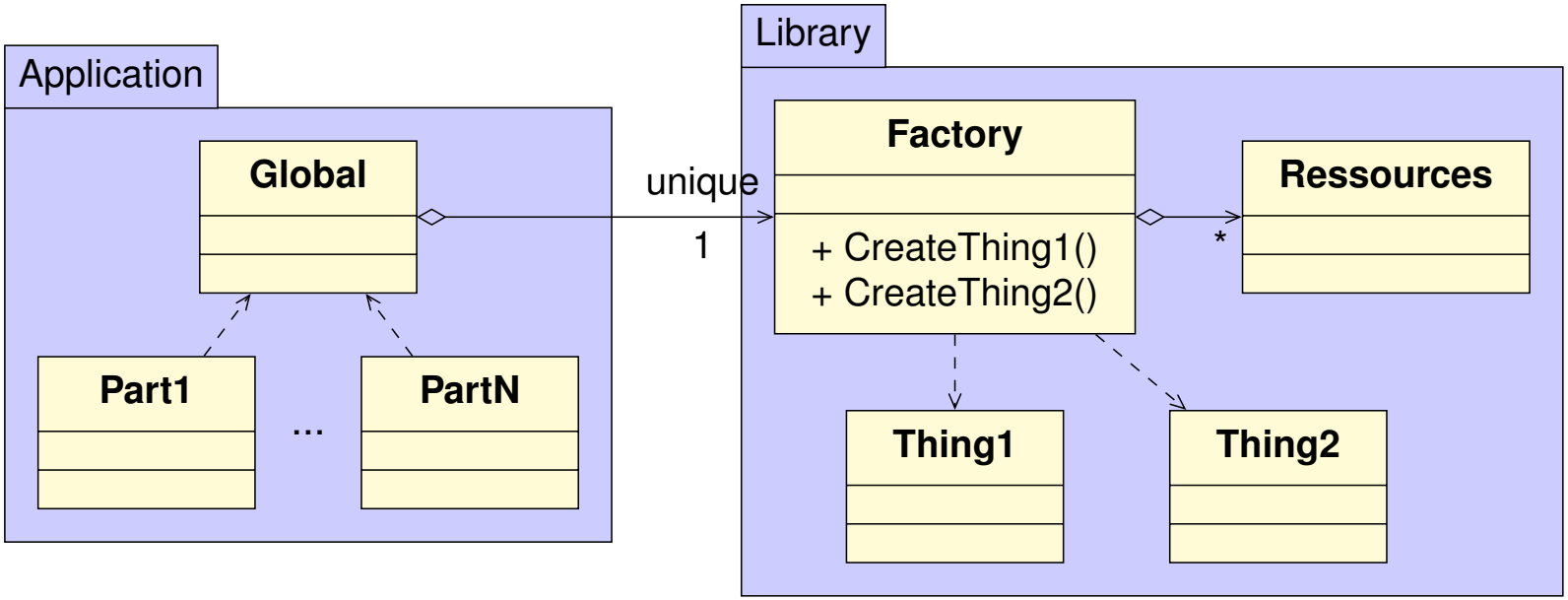
## Gérer le problème au niveau du client




- Problème : le client implémente une solution à un problème imposé par la bibliothèque !
- et ne résout pas le problème pour les biliotèques tierces,
- Solution : faire en sorte que la bibliothèque garantisse elle-même l'unicité grâce au patron **singleton**.

# Une mauvaise solution

## Gérer le problème au niveau du client




- Problème : le client implémente une solution à un problème imposé par la bibliothèque !
- et ne résout pas le problème pour les biliotèques tierces,
- Solution : faire en sorte que la bibliothèque garantisse elle-même l'unicité grâce au patron **singleton**.

# Le patron **Singleton**

## Aussi connu comme

Singleton

## Intention

- Garantir l'unicité de l'instance d'une classe (ou à l'intérieur d'une arborescence de classes).
- Fournir au client un moyen d'accès simple et fiable à cette instance.

## Motivation

- Parfois un service doit être rendu par une instance, mais utiliser Réutilisation d'une boite à outils dont l'interface n'est pas compatible avec celle conçue pour l'application.
- Parfois, il est nécessaire que cette instance soit unique.

# Singleton

## Motivations

Ce que l'on veut :

- Disposer d'une instance maximum d'une classe donnée,
- ET donner un accès global à cet objet dans l'application.

Raisons de ne vouloir qu'une instance au plus d'une classe :

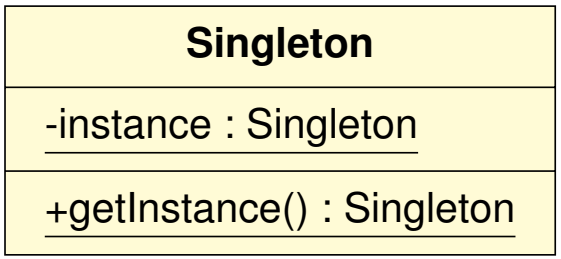
- Éviter à tout prix de travailler sur des instances séparées. Ex : contextes (graphiques, bases de données, ...), variables d'environnement/de session, .. ;
- variables globales,

# Participants du patron **Singleton**

- **Singleton** : Classe (ou arborescence de classes) dont on souhaite avoir une unique instance.

# Singleton

## Schéma de principe



# Différent types de singleton

On distingue deux familles de singletons :

- non dérivables (une seule instance d'une seule classe),
- dérivable (une seule instance d'une arborescence de classes.

pour lesquelles on peut envisager deux grands types d'implémentations :

- instanciation agressive,
- instanciation paresseuse (à la demande).

# Singleton non dérivable

## Instanciation agressive

```
1  /** Class Singleton is an implementation of a class
2  * that only allows one instantiation. */
3  public final class Singleton {
4    // The private reference to the one and only instance.
5    private static Singleton uniqueInstance =new Singleton();
6
7    // An instance attribute.
8    private int data = 0;
9
10   /**
11   * Returns a reference to the single instance.
12   * Creates the instance if it does not yet exist. (This is called lazy
         instantiation.)
13   */
14   public static Singleton instance() {
15     return uniqueInstance;
16   }
17
18   /** The Singleton Constructor.
19   * Note that it is private! No client can instantiate a Singleton object! */
20   private Singleton() {}
21
22   // Accessors and mutators here!
23   ...
24 }
```

# Singleton non dérivable

## Instanciation agressive - test

```
1  public static void main(String[] args) {
2    // Get a reference to the single instance of Singleton.
3    Singleton s = Singleton.instance();
4
5    // Set the data value.
6    s.setData(34);
7    System.out.println("First reference: " + s);
8    System.out.println("Singleton data value is: " + s.getData());
9
10   // Get another reference to the Singleton.
11   Singleton s1 = Singleton.instance();
12   System.out.println("\nSecond reference: " + s1);
13   System.out.println("Singleton data value is: " + s1.getData());
14   System.out.println("Is it the same object? " + (s==s1));
15 }
```

```
First reference:
single.eager.Singleton@19821f
Singleton data value is: 34
Second reference:
single.eager.Singleton@19821f
Singleton data value is: 34
Is it the same object? true
```

# Singleton non dérivable

## Instanciation agressive - test

```
1  public static void main(String[] args) {
2    // Get a reference to the single instance of Singleton.
3    Singleton s = Singleton.instance();
4
5    // Set the data value.
6    s.setData(34);
7    System.out.println("First reference: " + s);
8    System.out.println("Singleton data value is: " + s.getData());
9
10   // Get another reference to the Singleton.
11   Singleton s1 = Singleton.instance();
12   System.out.println("\nSecond reference: " + s1);
13   System.out.println("Singleton data value is: " + s1.getData());
14   System.out.println("Is it the same object? " + (s==s1));
15 }
```

```
First reference:
single.eager.Singleton@19821f
Singleton data value is: 34
Second reference:
single.eager.Singleton@19821f
Singleton data value is: 34
Is it the same object? true
```

# Singleton non dérivable

## Instanciation paresseuse

- Dans certains cas d'utilisation de l'application, le singleton ne sera jamais utilisé.
- Si l'initialisation du singleton est coûteuse (temps/mémoire), on souhaitera créer le singleton uniquement à sa première utilisation.
- On qualifie ce genre de traitements de *à la volé*/*à la demande* (*on the fly*) ou de *paresseux*/*laxiste*[2] (*lazy*).
- Note : ces chargements/calculs à la demande se généralisent et ne sont pas propres au singleton.

Allez, sors-leur ta tirade sur les logiciels mal conçus !

2. Ce qui n'est pas du tout péjoratif, bien au contraire !

# Singleton non dérivable

## Instanciation paresseuse

```
1  /** Class Singleton is an implementation of a class that
2  * only allows one instantiation.
3  */
4  public final class Singleton {
5    // The private reference to the one and only instance.
6    private static Singleton uniqueInstance = null;
7    // The private reference to the current class
8    private static Object lock = Singleton.class;
9    // An instance attribute.
10   private int data = 0;
11   /** The Singleton Constructor. Note that it is private!
12   * No client can instantiate a Singleton object!
13   */
14   private Singleton() {}
15   /**
16   * Returns a reference to the single instance.
17   * Creates the instance if it does not yet exist.
18   * (This is called lazy instantiation.)
19   */
20   public static Singleton instance() {
21     synchronized(lock){
22     if (uniqueInstance == null) {
23       uniqueInstance = new Singleton();
24     }
25       return uniqueInstance;
26     }
27   }
28   // Accessors and mutators here!
29 }
```

# Singleton non dérivable

## Instanciation laxiste - test

```
1  public static void main(String[] args) {
2    // Get a reference to the single instance of Singleton.
3    Singleton s = Singleton.instance();
4
5    // Set the data value.
6    s.setData(34);
7    System.out.println("First reference: " + s);
8    System.out.println("Singleton data value is: " + s.getData());
9
10   // Get another reference to the Singleton.
11   Singleton s1 = Singleton.instance();
12   System.out.println("\nSecond reference: " + s1);
13   System.out.println("Singleton data value is: " + s1.getData());
14   System.out.println("Is it the same object? " + (s==s1));
15 }
```

```
First reference:
single.eager.Singleton@19821f
Singleton data value is: 34
Second reference:
single.eager.Singleton@19821f
Singleton data value is: 34
Is it the same object? true
```

# Singleton non dérivable

## Instanciation laxiste - test

```
1  public static void main(String[] args) {
2    // Get a reference to the single instance of Singleton.
3    Singleton s = Singleton.instance();
4
5    // Set the data value.
6    s.setData(34);
7    System.out.println("First reference: " + s);
8    System.out.println("Singleton data value is: " + s.getData());
9
10   // Get another reference to the Singleton.
11   Singleton s1 = Singleton.instance();
12   System.out.println("\nSecond reference: " + s1);
13   System.out.println("Singleton data value is: " + s1.getData());
14   System.out.println("Is it the same object? " + (s==s1));
15 }
```

```
First reference:
single.eager.Singleton@19821f
Singleton data value is: 34
Second reference:
single.eager.Singleton@19821f
Singleton data value is: 34
Is it the same object? true
```

# Singleton non dérivable

## Note d'implémentation

Il semble que l'instanciation paresseuse est

- préférable dans les cas à coût élevés,
- un peu plus complexe à implémenter,
- difficile à garantir dans son fonctionnement,

et que l'instanciation agressive

- convient dans la plupart des cas,
- et est très simple à implémenter,
- sans soucis de concurrence.

Sauf que ... depuis Java 5 [3] ...

3. environ.

# Singleton non dérivable

## Note d'implémentation

Il semble que l'instanciation paresseuse est

- préférable dans les cas à coût élevés,
- un peu plus complexe à implémenter,
- difficile à garantir dans son fonctionnement,

et que l'instanciation agressive

- convient dans la plupart des cas,
- et est très simple à implémenter,
- sans soucis de concurrence.

Sauf que ... depuis Java 5 [3] ...

3. environ.

# Note d'implémentation en Java

Initialization of a class consists of executing its static initializers and the initializers for static fields declared in the class. [...]
**A class or interface type T will be initialized immediately before the first occurrence of any one of the following :**

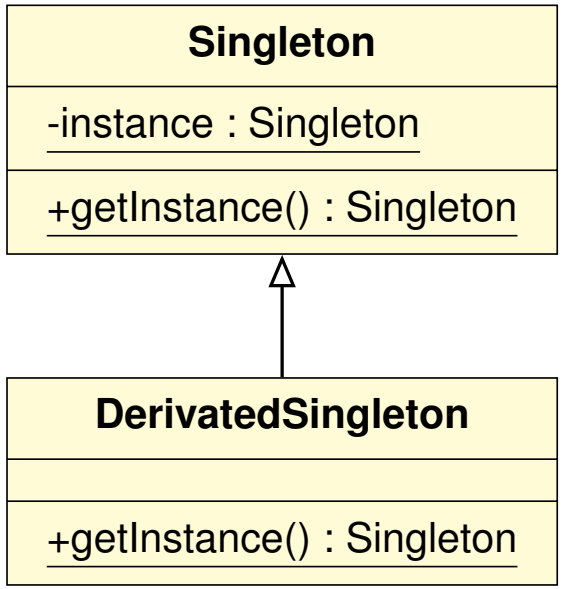
- T is a class and an instance of T is created.
- T is a class and a static method declared by T is invoked.
- A static field declared by T is assigned.
- A static field declared by T is used and the field is not a constant variable (§4.12.4).
- T is a top-level class, and an assert statement (§14.10) lexically nested within T is executed.

Invocation of certain reflective methods in class Class and in package java.lang.reflect also causes class or interface initialization. **A class or interface will not be initialized under any other circumstance.**

# Singleton dérivable

## il ne peut y en avoir qu'un ?

Que se passe-t-il si l'on dérive un singleton ? (On enlève `final`)



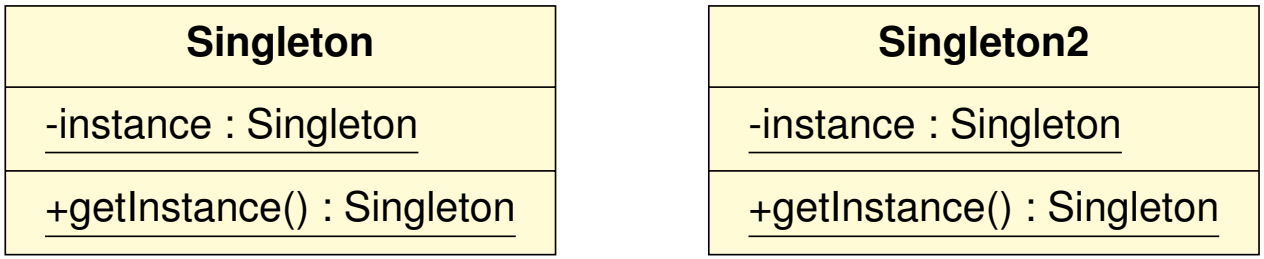

Le principe de singleton porte-t-il dans ce cas sur :

- chacune des classes ?
- sur l'arborescence entière ?

# Singleton dérivable

## il ne peut y en avoir qu'un ?

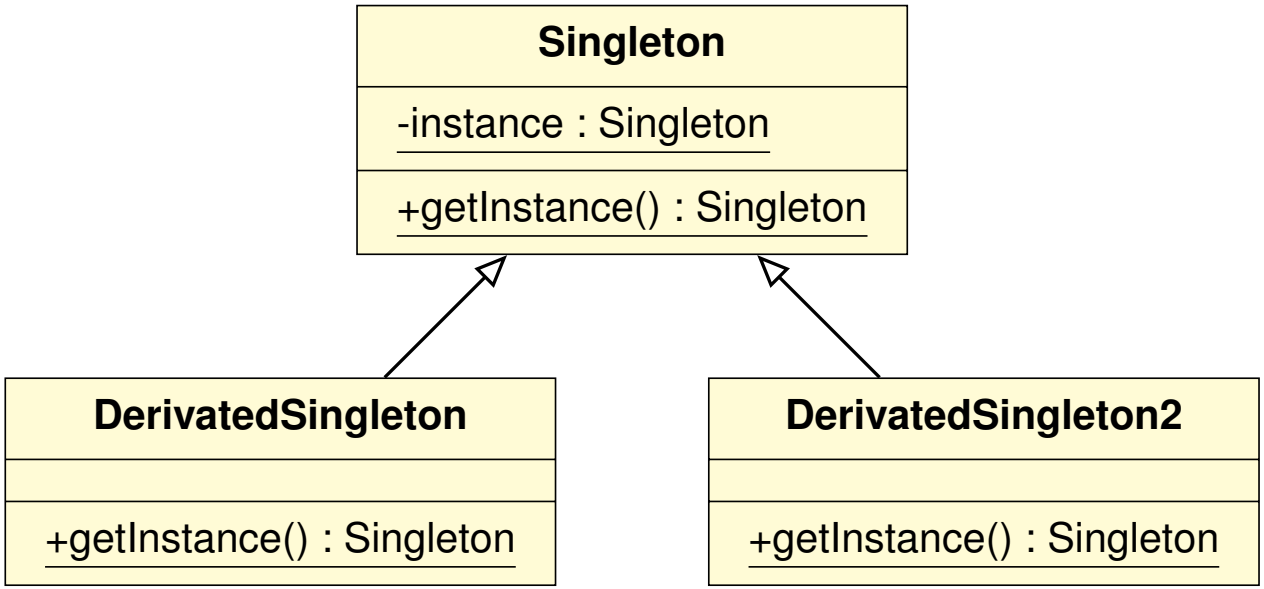
Remarque : si nous voulions deux singletons, aurions pu en écrire un second sans dériver :



# Singleton dérivable

## il ne peut y en avoir qu'un ?

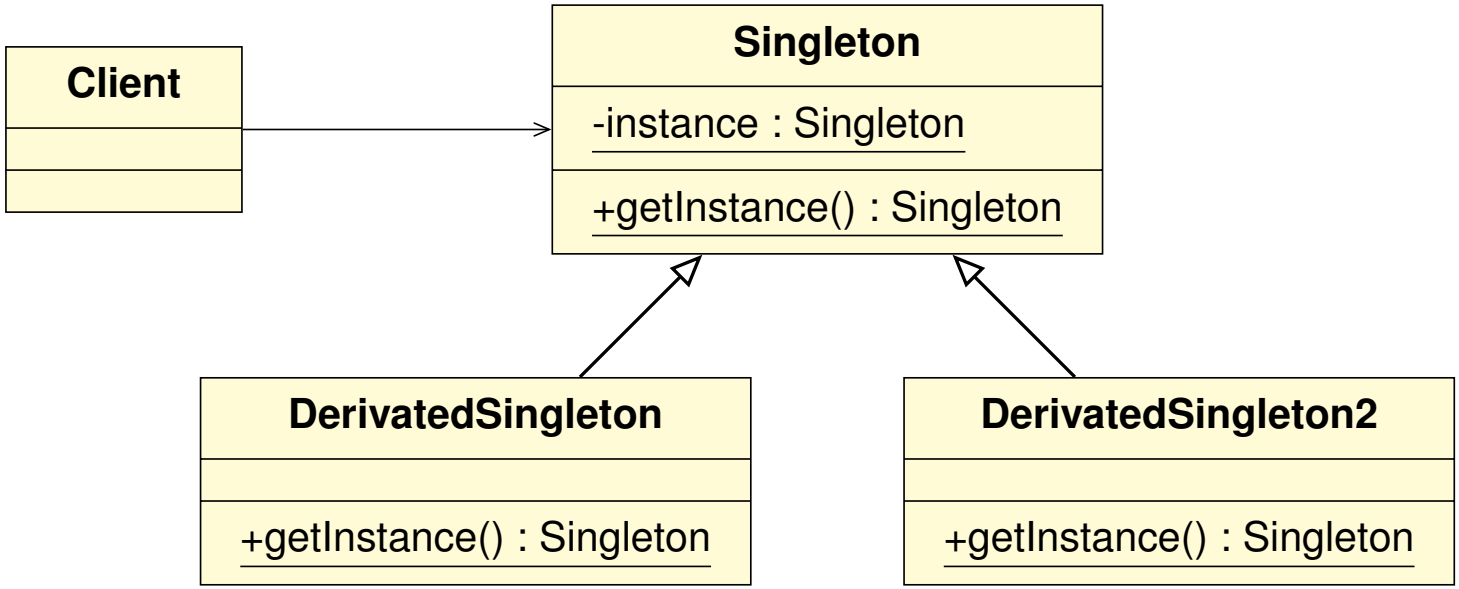
Le singleton porte sur toute l'arborescence d'héritage :




en particulier parce que l'attribut `instance` est partagé par toutes les classes.

# Singleton dérivable

## Objectif ?




- Le but est ici d'obtenir un singleton polymorphe, dont le type sera choisi dynamiquement,
- pour cela, le client continue de travailler avec la classe de base comme type statique.

# Singleton dérivable

## Instanciation

- Où et comment instancier ? Certainement pas comme ceci :

```
1  class Singleton {
2      private static instance=new DerivatedSingletonX();
3      ...
4  }
```

- On évitera également :

```
1  class Singleton {
2      ...
3      synchronized public Singleton getinstance(int type) {
4          switch(type) {
5              0: instance=new Singleton(); break;
6              1: instance=new DerivatedSingleton1(); break;
7              ...
8          }
9      }
10 }
```

car ...

# Singleton dérivable

## Instanciation 2

- Il semble plus raisonnable de s'adresser à la classe en question pour instancier le singleton désiré :

```
1  class DerivatedSingleton1 {
2      ...
3      synchronized public Singleton getinstance() {
4          instance=new DerivatedSingleton1(); return instance;
5      }
6  }
7  ...
8  class DerivatedSingleton2 {
9      ...
10     synchronized public Singleton getinstance() {
11         instance=new DerivatedSingleton2(); return instance;
12     }
13 }
```

# Singleton dérivable

## Utilisation

- Toutefois cette approche peut être confusante pour l'utilisateur :

```
1  public class Client {
2      public void test() {
3          DerivatedSingleton2.getInstance().doSomething();
4              // DerivatedSingleton2
5          Singleton.getInstance().doSomething();
6              // DerivatedSingleton2
7          DerivatedSingleton1.getInstance().doSomething();
8              // DerivatedSingleton2
9      }
10 }
```

- Le premier usage détermine le type pour toutes les demandes suivantes,
- ... ce qui est normale !
- une telle architecture enfreint le ...

# Singleton dérivable

## Utilisation

- Une meilleure approche consisterait à considérer la classe de base comme unique point d'accès au singleton et d'équiper les classes dérivées d'une méthode avec une sémantique de définition uniquement. Par exemple :

```
1  class Singleton {
2      synchronized public static Singleton getInstance() {
3          if (instance==null) throw new UndefinedSingletonException();
4          return instance;
5      }
6      synchronized protected static void setInstance(Singleton ins) {
           instance=ins; }
7  }
8
9  class DerivatedSingleton1()
10     public static void setInstance() {
11         if (instance==null) { instance=new DerivatedSingleton1(); }
12         else
13         if (! instance instanceOf DerivatedSingleton1) {
14             throw new AlreadyDefinedWithOtherTypeException();
15         }
16     }
17 }
18
19 class DerivatedSingleton2() { idem ...
```

# Principes respectés

- **S.R.P.** et **I.S.P.** : dépend des autres fonctionnalités, si l'on considère que le singleton n'est pas une responsabilité.
- **O.C.P.** : il faudra être vigilent sur certains choix d'implémentation. Par exemple, on pourra d'abord choisir un singleton non dérivable puis changer d'avis sans violer l'OCP, le contraire n'est pas vrai.
- **L.S.P.** : ok si on se repose uniquement sur la classe de base pour l'accès.
- **D.I.P.** : dépend du reste.

# Liens avec les autres patrons ?

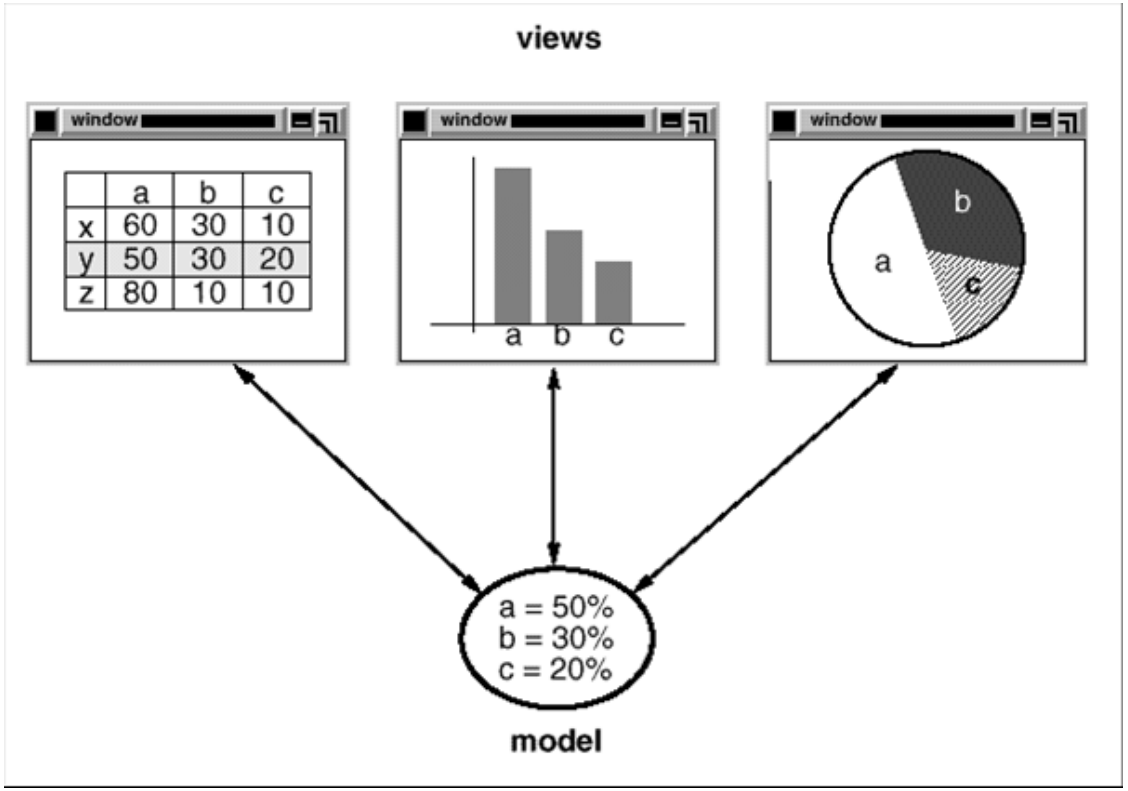
Le singleton est souvent utilisé avec le patron *Abstract Factory* lorsqu'il n'est pas nécessaire de multiplier les instances d'une usine abstraite.

# Le patron Observateur

Permet de synchroniser l'état de plusieurs objets dans une relation un-à-plusieurs.

# Motivation

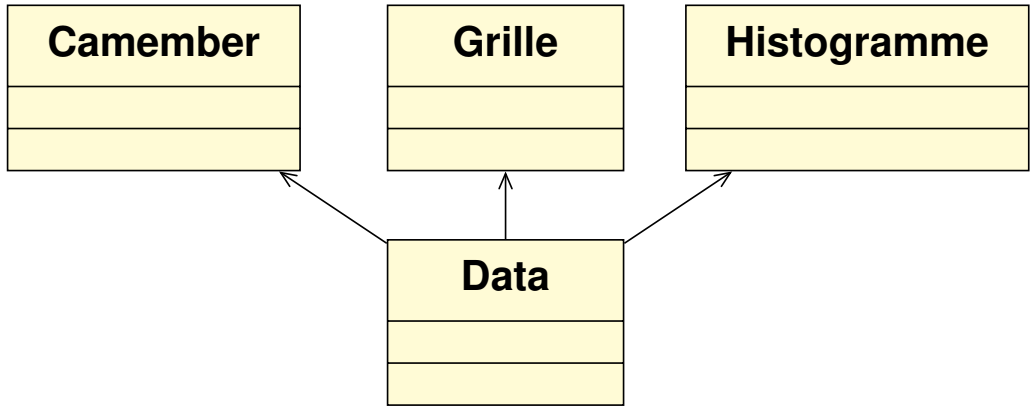
## Usage classique : modèle–vue




- La notion d'observateur est fréquemment utilisé pour synchonizer des vues avec l'état d'un modèle.

# Motivation

## À ne pas faire !

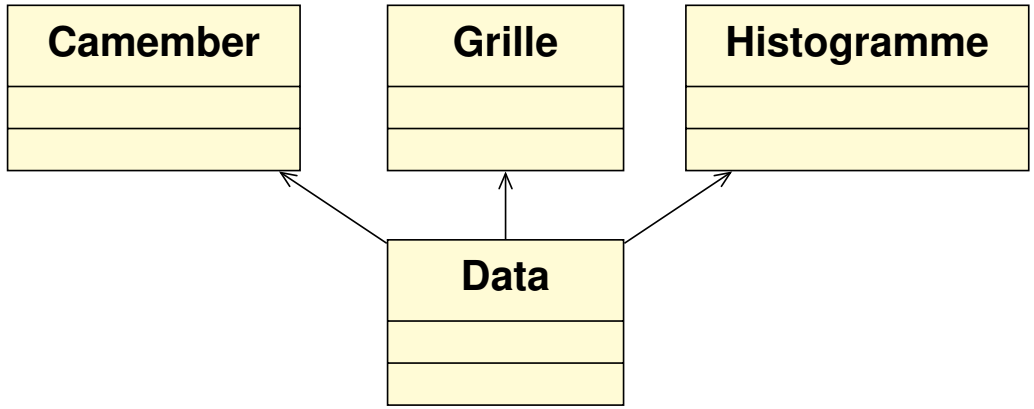



```
1  public class Data {
2      private Camembert cam;
3      private Grille gr;
4      private Histogramme hist;
5      ...
6
7      public setA(float a) {
8          this.a = a;
9          if (cam!=null) cam.updateA(a)
               ;
10         if (gr!=null) gr.updateA(a);
11         if (hist!=null) hist.updateA(
               a);
12     }
13 }
```

- Couplage du code des classe,
- Data devrait être un TDA,
- Configuration dynamique rigide,

# Motivation

## À ne pas faire !

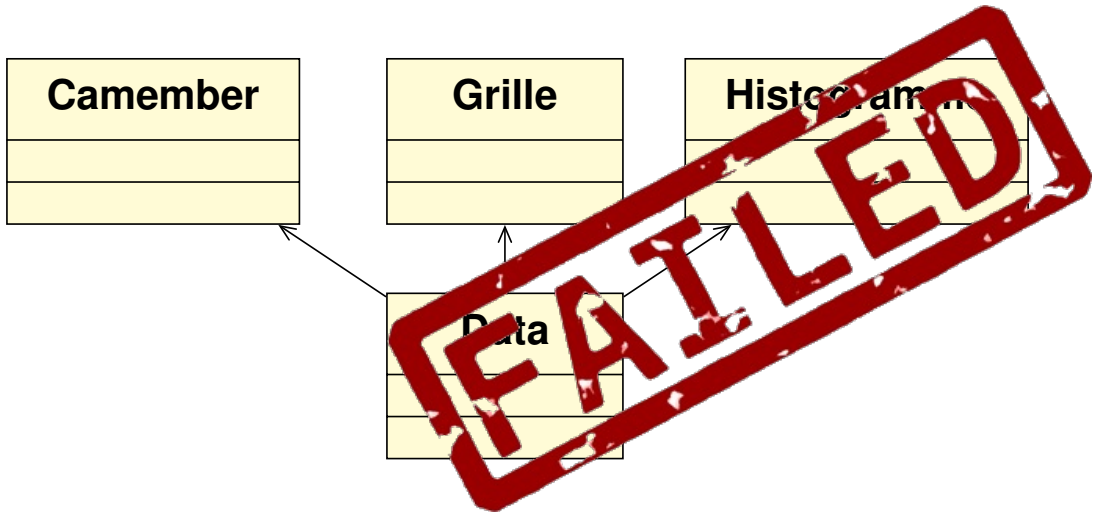



```
1  public class Data {
2      private Camembert cam;
3      private Grille gr;
4      private Histogramme hist;
5      ...
6
7      public setA(float a) {
8          this.a = a;
9          if (cam!=null) cam.updateA(a)
               ;
10         if (gr!=null) gr.updateA(a);
11         if (hist!=null) hist.updateA(
               a);
12     }
13 }
```

- Couplage du code des classe,
- Data devrait être un TDA,
- Configuration dynamique rigide,

# Motivation

## À ne pas faire !




```
1  public class Data {
2      private Camembert cam;
3      private Grille gr;
4      private Histogramme hist;
5      ...
6
7      public setA(float a) {
8          this.a = a;
9          if (cam!=null) cam.updateA(a)
               ;
10         if (gr!=null) gr.updateA(a);
11         if (hist!=null) hist.updateA(
               a);
12     }
13 }
```

- Couplage du code des classe,
- Data devrait être un TDA,
- Configuration dynamique rigide,

# Le patron **Observateur**

## Aussi connu comme

Dependents, Publish-Subscribe, Model-View

## Intention

- Définir une dépendance « un à plusieurs » dynamiquement entre un objet (l'observé) et plusieurs autres objets (observateurs).
- lorsque l'objet observé (le modèle) change d'état, tous les objets dépendants/observateurs (des vues/autres modèles) sont notifiés et mis à jour automatiquement.

## Motivation

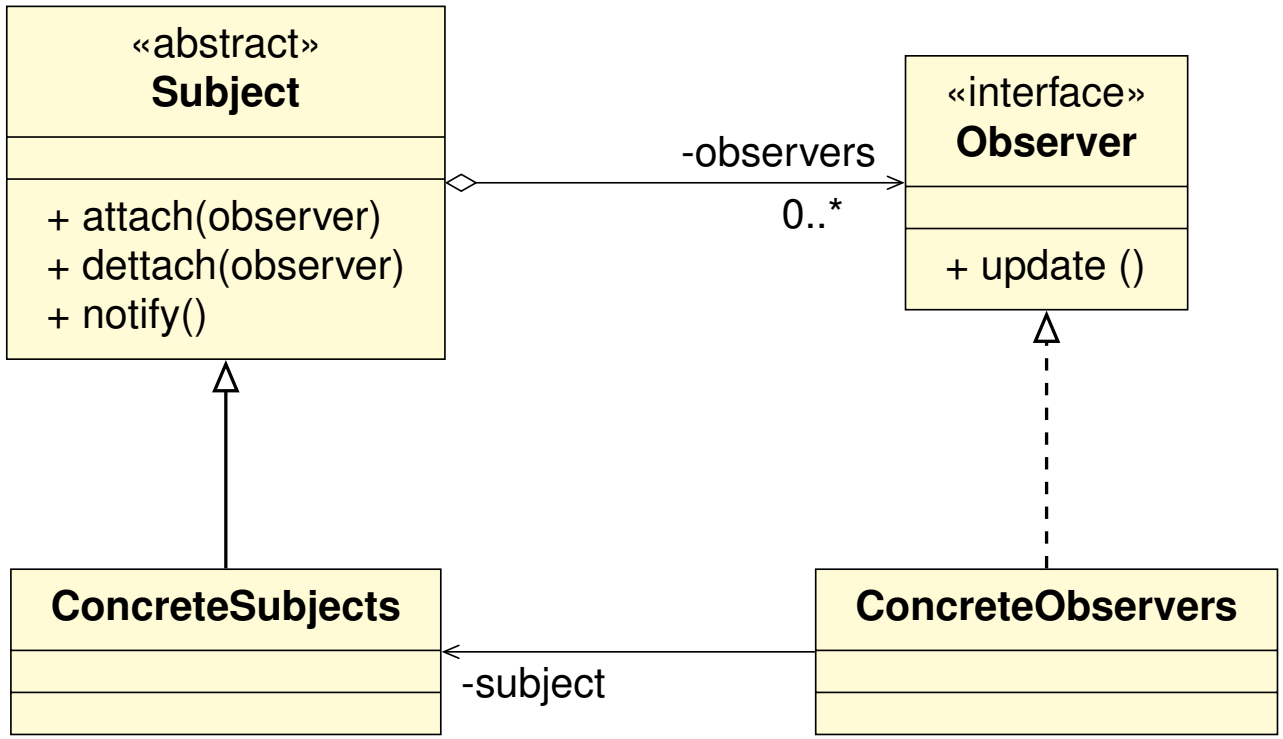
- Maintenir la relation de manière consistante tout en couplant les classes le plus faiblement possible.

# Participants du patron **Observateur**

- **Subject** : Souche de l'observé, maintient le lien avec les observateurs et définit l'interface de notification.
- **ConcreteSubjects** : Sujet(s) réel(s) notifiant les observateurs lors d'un changement d'état.
- **Observer** : Définit l'interface de notification des objets observateurs.
- **ConcreteObservers** : Classes pouvant être observatrices de l'état d'un autre objet.

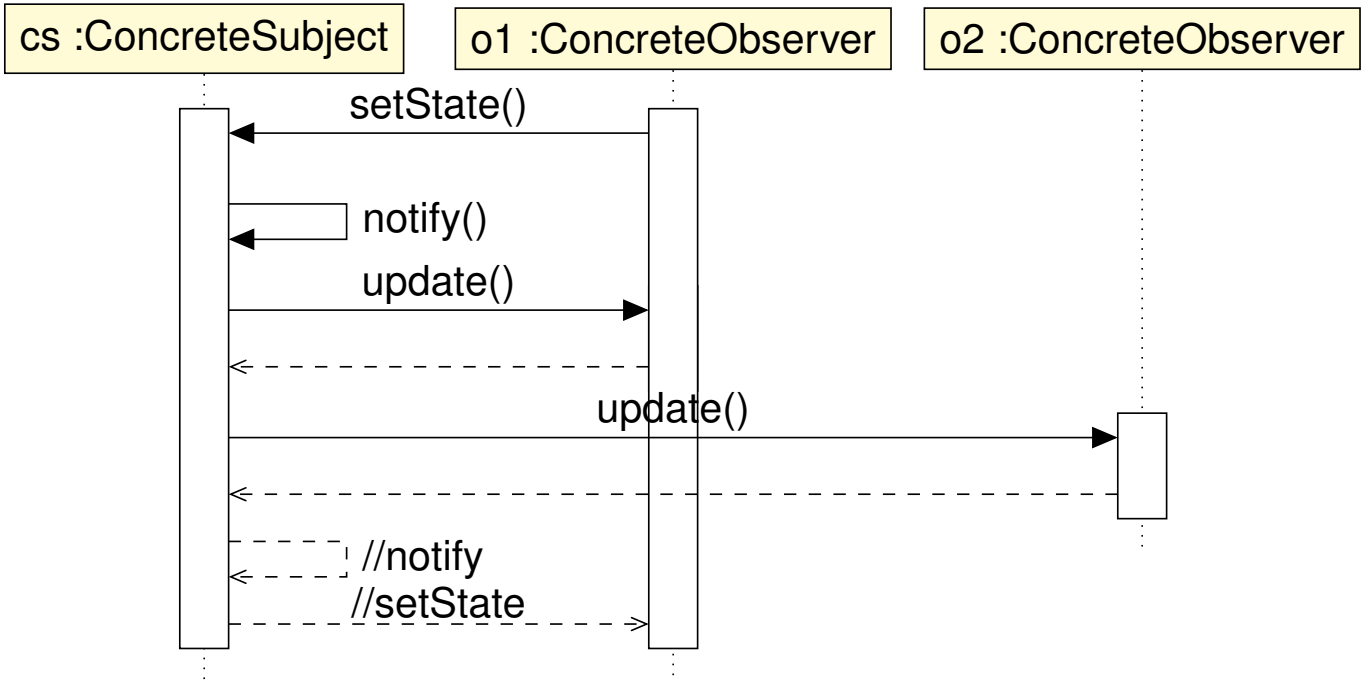
# Structure

## Diagramme de principe (original)



# Structure

## Fonctionnement




- Toute modification (souvent provoquée par un observateur) est notifiée à tous les observateurs enregistrés.

# Avantages

- Couplage minimal entre le modèle Observable et ses Observers :
  - Les modèles sont réutilisables in indépendamment des observateurs.
  - Ajout de nouveaux type d'observateurs sans modifier le modèle.
  - Le seul couplage du modèle concerne l'interface avec la méthode `update`.
  - Modèles et observateurs peuvent appartenir à des couches d'abstraction distinctes.
- Support de la diffusion dévénements :
  - Le modèle envoie des notifications aux observateurs abonnés.
  - Les observateurs peuvent ajoutés/supprimés dynamiquement.

# Inconvénients

- Effet possible dimbrication en cascade des notifications :
  - Les observateurs nont pas à être conscient des autres aussi doivent-ils déclencher avec soin les mises à jour, surtout si lun deux sert également de contrôleur.
  - Une interface de mise à jour trop simple nécessite que les observateurs repère les objets modifiés.
- Éviter de construire des cycles : si un observateur est également observable, il est possible de déclencher une récursion infinie.
- Les sources de mises à jour peuvent être de plusieurs natures :
  - le modèle lui-même,
  - les observateurs,
  - un acteur tiers.

# Réalisation

- Implémentation de la liste des observateurs (ArrayList, LinkedList, BTree) : les opérations d'ajout/suppression doivent elles être plus performantes que la notification ?
- Comment distinguer l'émetteur d'une notification lorsqu'un observateur observe plusieurs modèles ?
- S'assurer que le modèle met à jour son état et le stabilise avant de déclencher la notification.
- Combien d'information le modèle envoie-t-il sur les changements qu'il a subit :
  - *push model* : le modèle transmet ce qui a été modifié,
  - *pull model* : l'observateur vient chercher le nouvel état depuis le modèle.

# Réalisation

- Les observateurs peuvent souscrire aux seuls événements qui les intéressent. Dans ce cas le modèle filtre au moment de la notification.
- Il peut arriver qu'un observateur qui observe plusieurs modèles simultanément ne puisse entreprendre sa tâche qu'une fois que tous les observés ont changé d'état :
  - dans ce cas on utilisera un intermédiaire qui jouera le rôle de médiateur (voir le patron du même nom),
  - les modèles notifient à tour de rôle le médiateur de leurs changements d'état et celui-ci réalise les traitements nécessaires avant d'invoquer l'observateur final.

# Réalisation

## l'API Java

- L'API Java (`java.util`) fournit une souche pour implémenter le patron observateur/observé :

```
1  package java.util;
2
3  public class Observable {
4      // Construct an Observable with zero Observers :
5      public Observable();
6      // Adds an observer to the set of observers of this object :
7      public synchronized void addObserver(Observer o);
8      // Deletes an observer from the set of observers of this object :
9      public synchronized void deleteObserver(Observer o);
10
11     // Indicates that this object has changed since last notification :
12     protected synchronized void setChanged();
13     // Indicates that this object no longer has changed :
14     protected void clearChanged()
15     // Tests if this object has changed :
16     public synchronized boolean hasChanged();
17
18     // Notify all the observers if this object has changed :
19     public void notifyObservers(Object arg);
20     public void notifyObservers();
21     ...
22 }
```

# Réalisation

## l'API Java

- De même pour créer une classe observateur :

```
1 package java.util;
2
3 public interface Observer {
4     public abstract void update(Observable o, Object arg);
5 }
```

- This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.
- Parameters :
  - `o` : the observable object that triggered the notification,
  - `arg` : the optional parameter passed by the observable.

# Exemple 1

## L'observable

```
1  package myobserver;
2  import java.util.Observable;
3
4  public class ConcreteSubject extends Observable {
5      private String name;
6      private float price;
7
8      public ConcreteSubject(String name, float price) {
9          this.name = name;
10         this.price = price;
11         System.out.println("ConcreteSubject created: "+name+" at "+price);
12     }
13
14     public String getName() { return name; }
15
16     public float getPrice() { return price; }
17
18     public void setName(String name) {
19         this.name = name;
20         setChanged();
21         notifyObservers(name);
22     }
23
24     public void setPrice(float price) {
25         this.price = price;
26         setChanged();
27         notifyObservers(new Float(price));
28     }
29 }
```

# Exemple 1

## Observateur de changement nom

```
1  package myobserver;
2  import java.util.Observable;
3  import java.util.Observer;
4
5  public class NameObserver implements Observer {
6      private String name;
7
8      public NameObserver() {
9          name = null;
10         System.out.println("NameObserver created : Name is "+name);
11     }
12
13     public void update(Observable obj, Object arg) {
14         if (arg instanceof String) {  // Beurk !!!!
15             name = (String) arg;
16             System.out.println("NameObserver : Name changed to "+name);
17         } else {
18             System.out.println("NameObserver : Some other change to subject!
                   ");
19         }
20     }
21 }
```

# Exemple 1

## Observateur de changement prix

```
1  package myobserver;
2  import java.util.Observable;
3  import java.util.Observer;
4
5  public class PriceObserver implements Observer {
6      private float price;
7
8      public PriceObserver() {
9          price = 0;
10         System.out.println("PriceObserver created: Price is "+price);
11     }
12
13     public void update(Observable obj, Object arg) {
14         if (arg instanceof Float) { // Beurk !!!!
15             price = ((Float) arg).floatValue();
16             System.out.println("PriceObserver: Price changed to "+price);
17         } else {
18             System.out.println("PriceObserver: Some other change to subject
                   !");
19         }
20     }
21 }
```

# Exemple 1

## Programme de test

```
1  public class Test {
2      public static void main(String args[]) {
3          // Create the Subject and Observers.
4          ConcreteSubject s = new ConcreteSubject("Corn␣Pops", 1.29f);
5          NameObserver nameObs = new NameObserver();
6          PriceObserver priceObs = new PriceObserver();
7          // Add those Observers!
8          s.addObserver(nameObs);
9          s.addObserver(priceObs);
10         // Make changes to the Subject.
11         s.setName("Frosted␣Flakes");
12         s.setPrice(4.57f);
13         s.setPrice(9.22f);
14         s.setName("Sugar␣Crispies");
15     }
16 }
```

Produit :

```
1  ConcreteSubject created: Corn Pops at 1.29
2  NameObserver created: Name is null
3  PriceObserver created: Price is 0.0
4  PriceObserver: Some other change to subject!
5  NameObserver: Name changed to Frosted Flakes
6  PriceObserver: Price changed to 4.57
7  NameObserver: Some other change to subject!
8  PriceObserver: Price changed to 9.22
9  NameObserver: Some other change to subject!
10 PriceObserver: Some other change to subject!
11 NameObserver: Name changed to Sugar Crispies
```

# Exemple 1

## Problème

- Supposons que la classe qui doit devenir observable soit déjà incluse dans une hiérarchie :

```
1 public classe ConcreteSubject extends ParentClass {...
```

- Java ne supportant pas l'héritage multiple de classe [4], `ConcreteSubject` ne peut étendre à la fois les classes `Observable` et `ParentClass` !
- De plus, cela viole la règle de codage 4 : « ne pas hériter dune classe utilitaire » !
- Solution : utiliser la composition [5] pour envelopper un objet observable.

4. Et c'est tant mieux !
5. Quelle surprise !

# Exemple 1

## Problème No 2

- On note justement que la classe `Observable` de l'API Java a le bon goût de ne pas être abstraite,
- malheureusement ses concepteurs ont donné la visibilité *protégée* aux méthodes de gestion du drapeau de modification :

```
1  package java.util;
2  public class Observable {
3      ...
4      // Indicates that this object has changed since last notification :
5      protected synchronized void setChanged();
6      // Indicates that this object no longer has changed :
7      protected void clearChanged()
8      ...
9  }
```

- forçant à dériver cette classe de toute façon ! :-/
- Nous utiliserons une classe supplémentaire qui sera notre observateur délégué.

# Exemple 2

## Observable délégué

```
1  public class ConcreteSubject extends ParentClass {
2       private String name;      private float price;
3       private Observable obs;
4
5       public ConcreteSubject(String name, float price) {
6            this.name = name;      this.price = price;
7            System.out.println("ConcreteSubject created : "+name+" at "+price);
8            obs = new Observable() { // Utilisation d'une classe anonyme
9                 public void setChanged() { super.setChanged(); }
10                public void clearChanged() { super.clearChanged(); }
11           }
12      }
13
14      public String getName() { return name; }
15      public float getPrice() { return price; }
16
17      public void setName(String name) {
18           this.name = name;
19           obs.setChanged();
20           obs.notifyObservers(name);
21      }
22      public void setPrice(float price) {
23           this.price = price;
24           obs.setChanged();
25           obs.notifyObservers(new Float(price));
26      }
27
28      public void addObserver(Observer o) { obs.addObserver(o); }
29      public void deleteObserver(Observer o) { obs.deleteObserver(o); }
30 }
```

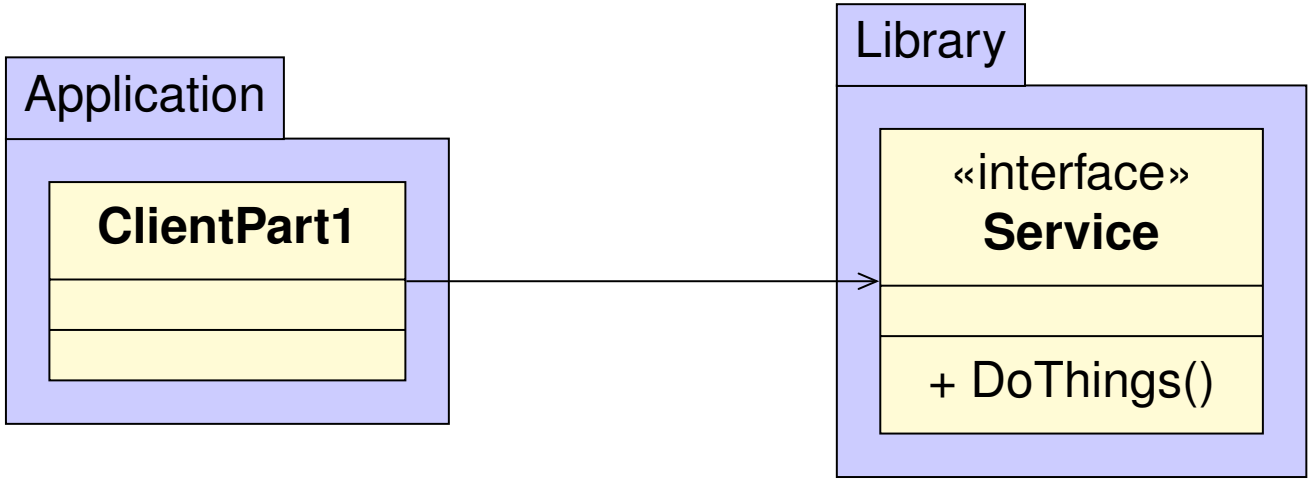
# Principes respectés

- **S.R.P.** : ?
- **O.C.P.** : ?
- **L.S.P.** : ?
- **I.S.P.** : XXX.
- **D.I.P.** : ?

# Le patron Proxy

Contrôle l'accès à un objet au moyen d'un intermédiaire.

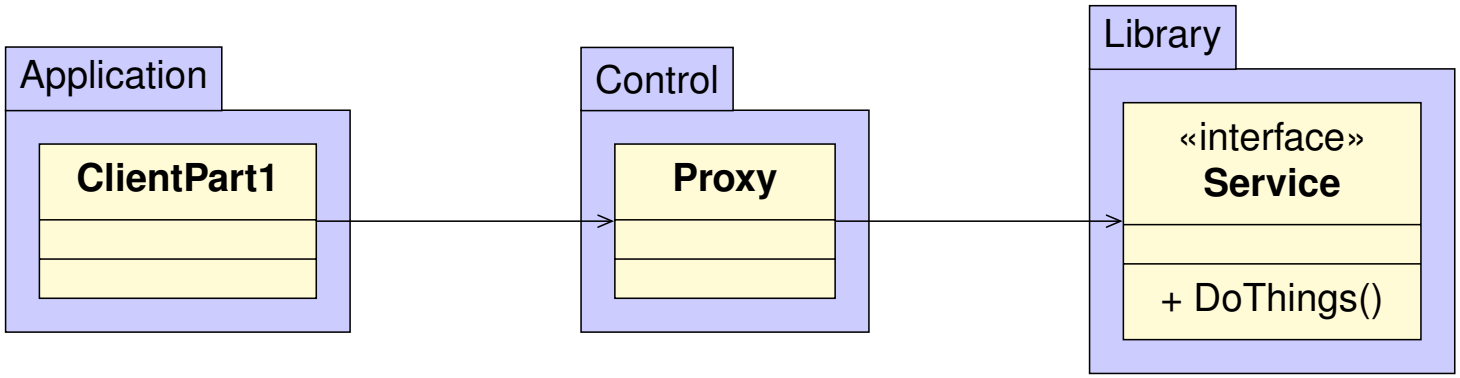
# situation initiale




- Une application (existante) utilise les services d'une bibliothèque (existante).
- On souhaite ajouter des contrôles ou des comportements dans une relation entre un composant existant et son code client.

# situation initiale

- On ne peut modifier ni l'un ni l'autre, mais l'on peut toutefois modifier le lien associatif entre-eux.
- Solution : intercaler un objet intermédiaire qui ajoutera les nouveaux comportements sans que le client "s'en rende compte" [6] et déléguera à la bibliothèque.




6. Sans que le client ait besoin d'être modifié.

# Le patron **Proxy/Mandataire**

## Aussi connu comme

Surrogate (subrogé)

## Intention

Fournir un intermédiaire qui permette de contrôler l'accès à un objet.

## Motivation

- Ajouter un contrôle ou des traitements lorsqu'un client accède à un objet.
- Le client ne "sait" pas qu'il s'adresse à un proxy.
- le proxy délègue les traitements à l'objet réel.

# Proxy/mandataire

## Différents types/motivations

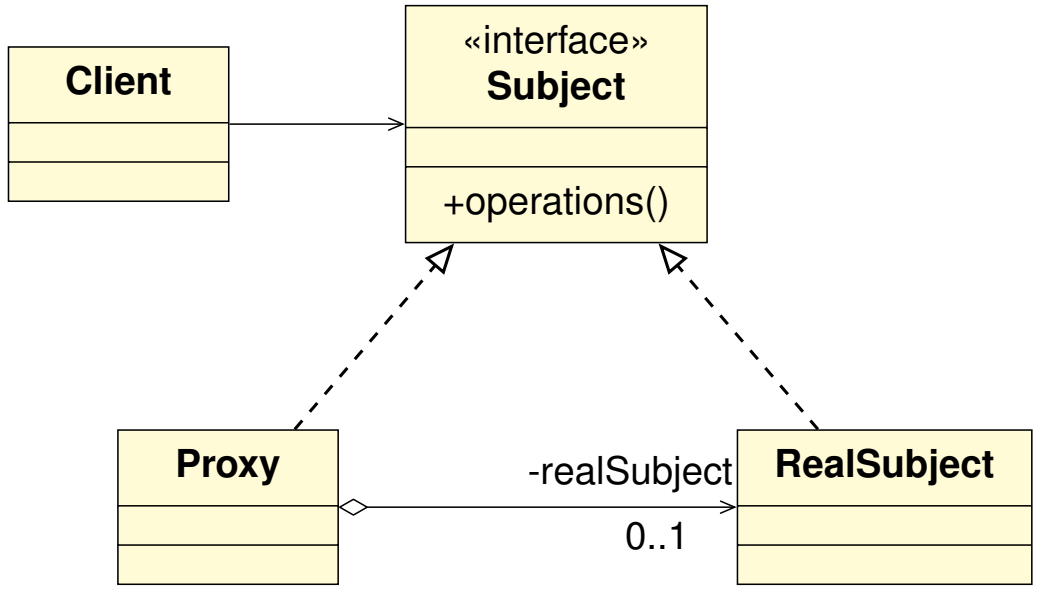
- **Remote Proxy :** référence un objet situé dans un espace d'adresse différent (voir java.rmi)
- **Virtual Proxy :** permet la création laxiste d'une copie en mémoire rapide d'un objet.
- **Copy-On-Write Proxy :** diffère le clonage d'objets tant qu'aucune opération de modification ne les différencie. C'est une forme de virtual proxy.
- **Protection (Access) Proxy :** fournit aux clients des niveaux d'accès différents à l'objet cible.
- **Cache Proxy :** permet de mémoriser et partager les résultats d'opérations coûteuses .
- **Firewall Proxy :** protège la cible des mauvais clients (ou vice versa)
- **Synchronization Proxy :** gère les accès concurrens à la cible.

# Participants du patron **Proxy**

- **Subject** : L'interface définissant le comportement des classes dont les objets seront contrôlés
- **RealSubject** : Les classes dont les instances pourront être contrôlées par le Proxy.
- **Proxy** :
  - implémente l'interface Sujet afin de pouvoir se substituer au sujet réel ;
  - délègue les actions au sujet réel (maintien une référence vers celui-ci) ;
  - implémente le contrôle à effectuer ou les comportements à ajouter.
- **Client** : Code utilisateur qui manipulera indifféremment le proxy ou le sujet réel

# Patron Proxy

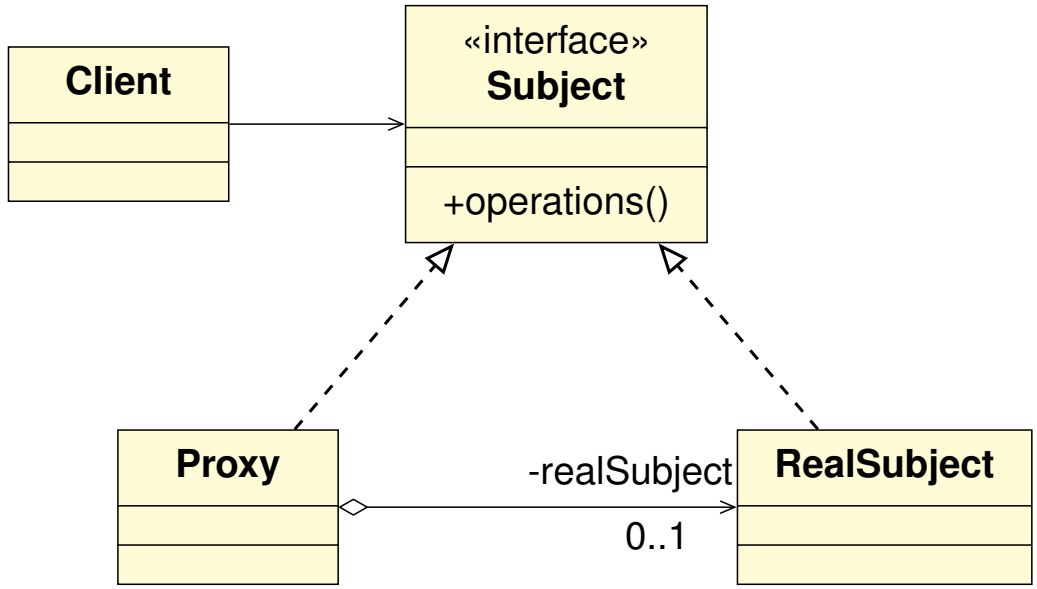
## Schéma de principe



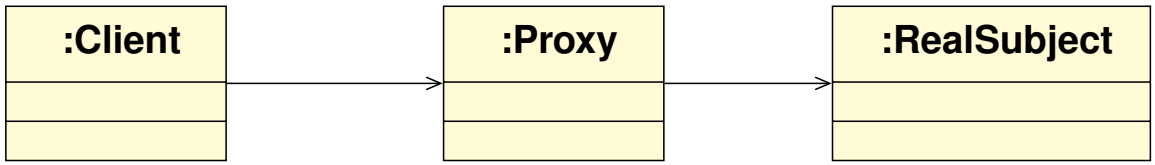

- Le client manipule le sujet indirectement à travers le proxy :

# Patron Proxy

## Schéma de principe



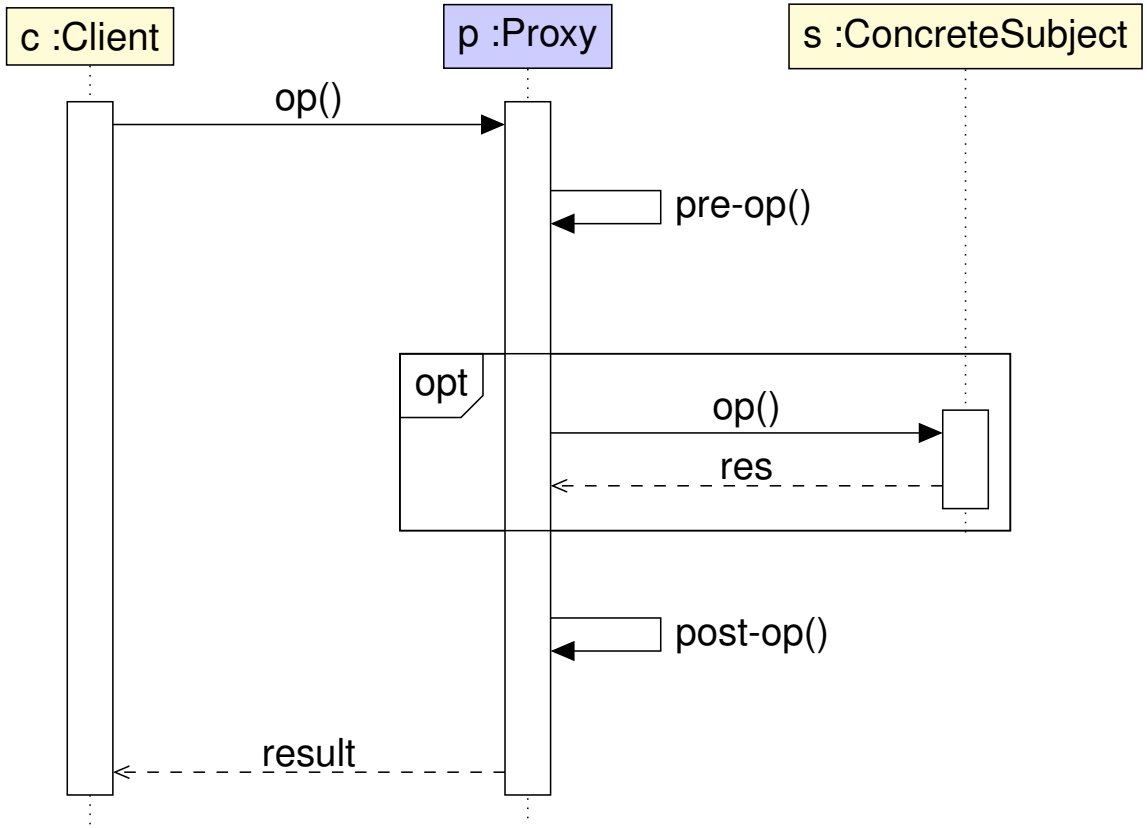

- Le client manipule le sujet indirectement à travers le proxy :



# Patron Proxy

## Schéma de principe

- Le proxy ajoute des opérations des comportements ou des contrôles :



# Ex1 : Proxy de synchronisation

## Situation de départ

- Une bibliothèque fournit une implémentation d'une table :

```
1  public interface ITable<T> {
2      public T getElementAt(int row, int column);
3      public void setElementAt(T element, int row, int column);
4      public int getNumberOfRows();
5  }
6  ...
7  public class Table<T> implements ITable<T> {
8      // ...
9      int numrows;
10     public T getElementAt(int row, int column) {
11         // Get the element.
12         return ...;
13     }
14     public void setElementAt(T element, int row, int column) {
15         // Set the element.
16     }
17     public int getNumberOfRows() {
18         return numrows;
19     }
20 }
```

# Ex1 : Proxy de synchronisation

## Problème

- L'auteur de la bibliothèque originale n'a pas prévu un usage en environnement concurrent (*thread safety*).
- Notre application comporte plusieurs fils d'exécution et nécessite des garanties de synchronisation.
- Peut-on modifier la bibliothèque originale ?
- Techniquement, l'ajout de la synchronisation ne change pas le contrat, toutefois :
  - changerions nous le comportement ? [7]
  - Sommes nous les auteurs de la bibliothèque ? (mise-à-jours/patches)
- Nous allons ici préférer ajouter le nouveau comportement au dessus de l'implémentation existante [8] à l'aide d'un proxy.

7. Ici ajout d'un coût pour les autres clients.
8. Solution proposée par Roger Whitney

# Ex1 : Proxy de synchronisation

## Problème

- L'auteur de la bibliothèque originale n'a pas prévu un usage en environnement concurrent (*thread safety*).
- Notre application comporte plusieurs fils d'exécution et nécessite des garanties de synchronisation.
- Peut-on modifier la bibliothèque originale ?
- Techniquement, l'ajout de la synchronisation ne change pas le contrat, toutefois :
  - changerions nous le comportement ? [7]
  - Sommes nous les auteurs de la bibliothèque ? (mise-à-jours/patches)
- Nous allons ici préférer ajouter le nouveau comportement au dessus de l'implémentation existante [8] à l'aide d'un proxy.

7. Ici ajout d'un coût pour les autres clients.
8. Solution proposée par Roger Whitney

# Ex1 : Proxy de synchronisation

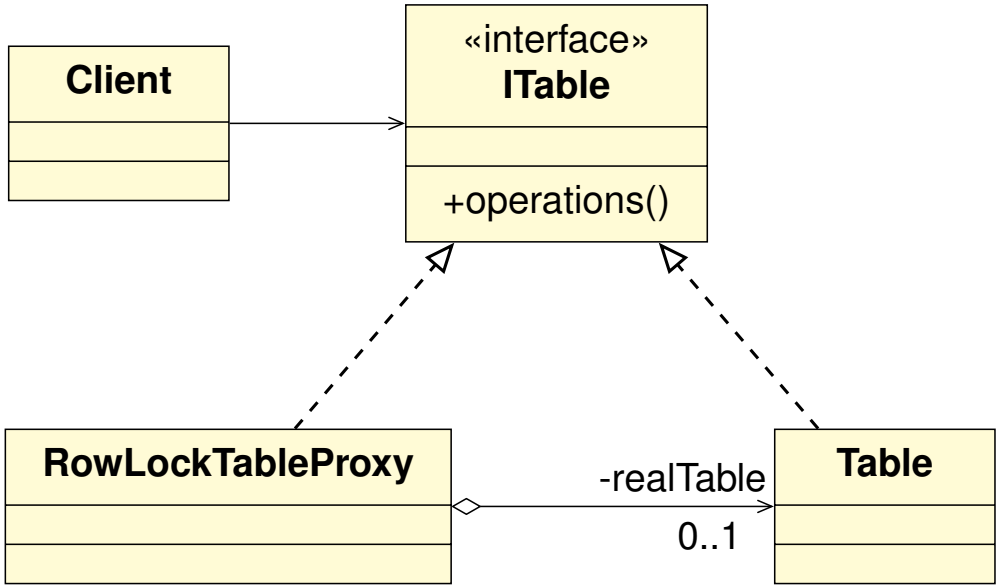
## Problème

- L'auteur de la bibliothèque originale n'a pas prévu un usage en environnement concurrent (*thread safety*).
- Notre application comporte plusieurs fils d'exécution et nécessite des garanties de synchronisation.
- Peut-on modifier la bibliothèque originale ?
- Techniquement, l'ajout de la synchronisation ne change pas le contrat, toutefois :
  - changerions nous le comportement ? [7]
  - Sommes nous les auteurs de la bibliothèque ? (mise-à-jours/patches)
- Nous allons ici préférer ajouter le nouveau comportement au dessus de l'implémentation existante [8] à l'aide d'un proxy.

7. Ici ajout d'un coût pour les autres clients.
8. Solution proposée par Roger Whitney

# Ex1 : Proxy de synchronisation

## Solution




- À l'utilisation, on écrira plutôt :

```
1  public class Client {
2      public ITable getTable() {
3          return new RowLockTableProxy<Integer>(new Table<Integer>(...));
4      }
5  }
```

# Ex1 : Proxy de synchronisation

## Solution

- On propose un verrou par ligne :

```
1  public class RowLockTableProxy<T> implements ITable<T> {
2      Table<T> realTable;
3      Integer[] locks;
4
5      public RowLockTableProxy(Table<T> toLock) {
6          realTable = toLock;
7          locks = new Integer[toLock.getNumberOfRows()];
8          for (int row = 0; row < toLock.getNumberOfRows(); row++)
9          locks[row] = new Integer(row);
10     }
11
12     public T getElementAt(int row, int column) {
13         synchronized (locks[row]) {
14             return realTable.getElementAt(row, column);
15         }
16     }
17     public void setElementAt(T element, int row, int column) {
18         synchronized (locks[row]) {
19             realTable.setElementAt(element, row, column);
20         }
21     }
22     public int getNumberOfRows() {
23         return realTable.getNumberOfRows();
24     }
25 }
```

# Exemple 2 : Copy-on-Write

## Situation initiale

- Un programme opère sur des copies partagées et/ou séparées d'une même structure de données :

```
1  ...
2  Integer algorithm(HashTable<Integer> h) {
3      // Sous-algorithmes :
4      subAlgo1(h);                          // modifications globale.
5      int r1=subAlgo2(h.clone());           // Modif. locales (ou pas!)
6      int r2=subAlgo4(h.clone());           // Modif. locales (ou pas!)
7
8      int theBigResult=subAlgo3(h, r1, r2); // pas de modification.
9
10     return theBigResult;
11 }
12 ...
```

- certains sous programmes travaillant sur des copies peuvent ne pas modifier ces dernières.
- On considère que la table est assez grosse et que les appels sans modification sont nombreux.
- Dans ce cas la copie induit un coût inutile.

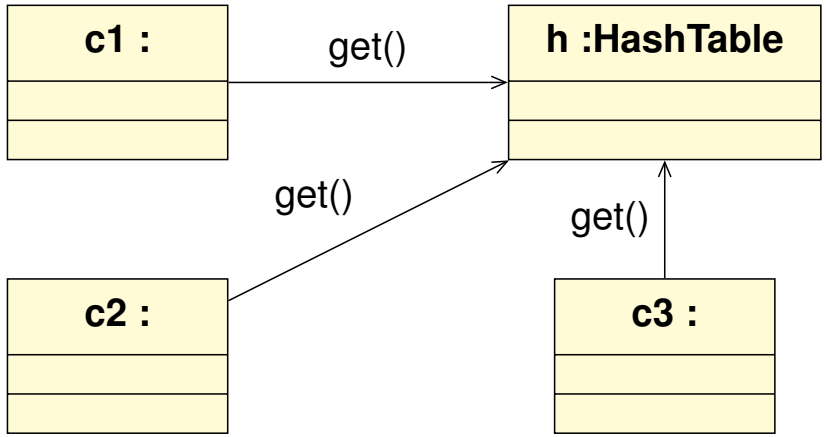
# Exemple 2 : Copy-on-Write

## Solution

- On se propose d'utiliser la technique de *copie à l'écriture* (*copy-on-write*).
- Cette technique est largement utilisée, notamment dans les implémentations de systèmes de fichiers (ZFS, BtrFS, LVM, etc.)
- Là aussi, nous allons souhaiter conserver l'implémentation initiale intacte et ajouter une nouvelle version, sans toutefois tout réécrire...

# Exemple 2 : Copy-on-Write

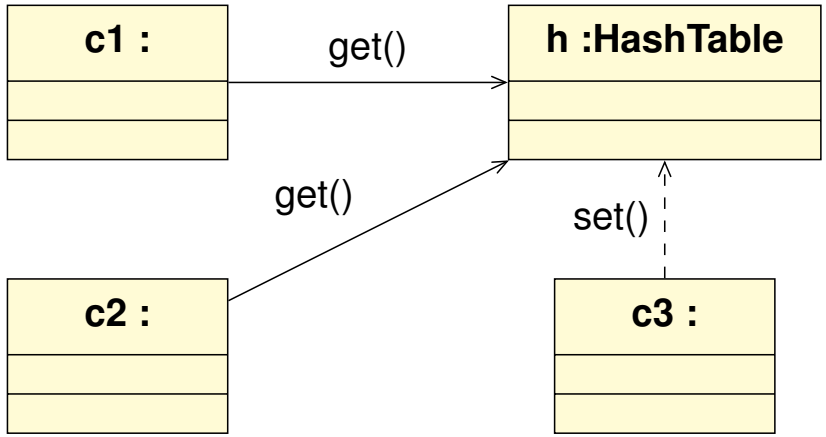
## Fonctionnement




- Tous les clients travaillent sur la même copie,
- le premier client qui effectue une modification,
- provoque une duplication de la structure, il sera le seul concerné par sa modification et continuera de travailler sur sa propre copie...

# Exemple 2 : Copy-on-Write

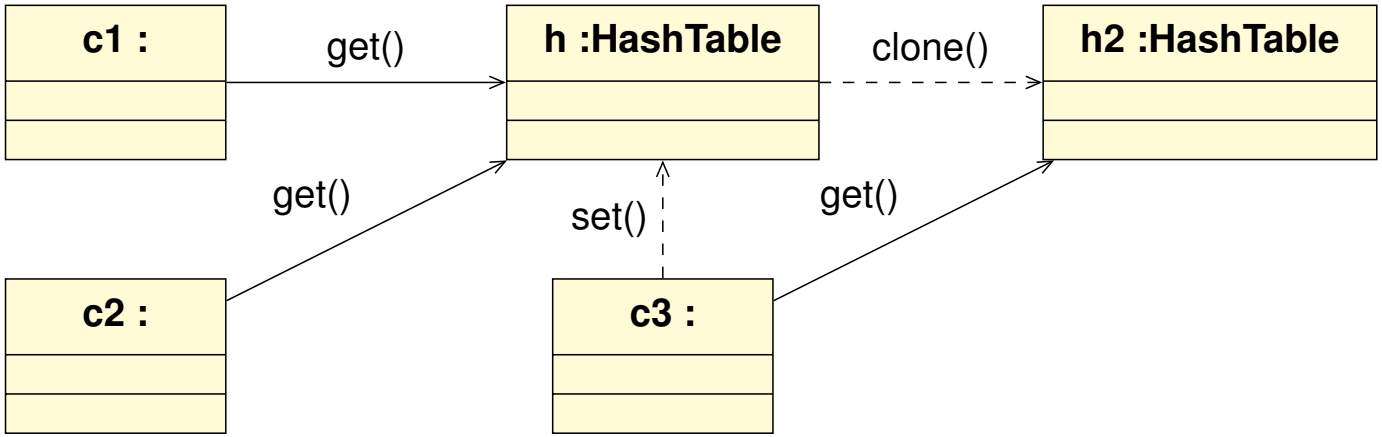
## Fonctionnement




- Tous les clients travaillent sur la même copie,
- le premier client qui effectue une modification,
- provoque une duplication de la structure, il sera le seul concerné par sa modification et continuera de travailler sur sa propre copie...

# Exemple 2 : Copy-on-Write

## Fonctionnement

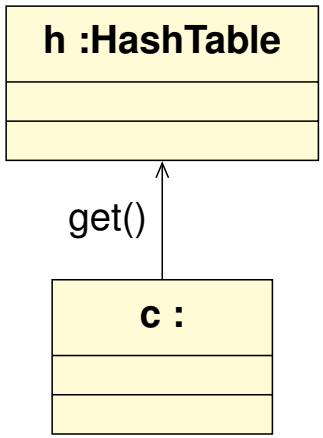



- Tous les clients travaillent sur la même copie,
- le premier client qui effectue une modification,
- provoque une duplication de la structure, il sera le seul concerné par sa modification et continuera de travailler sur sa propre copie...

# Exemple 2 : Copy-on-Write

## Raffinement

- On souhaite bien sûr éviter ce cas :



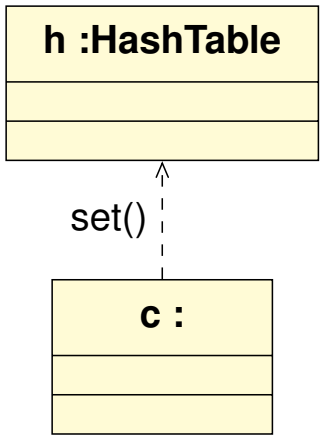

- Un seul client manipule la structure,
- s'il opère une modification,
- alors la duplication n'est pas nécessaire...

$\rightarrow$ il faut donc procéder au clonage seulement lorsque plus d'un client utilise la table.

# Exemple 2 : Copy-on-Write

## Raffinement

- On souhaite bien sûr éviter ce cas :



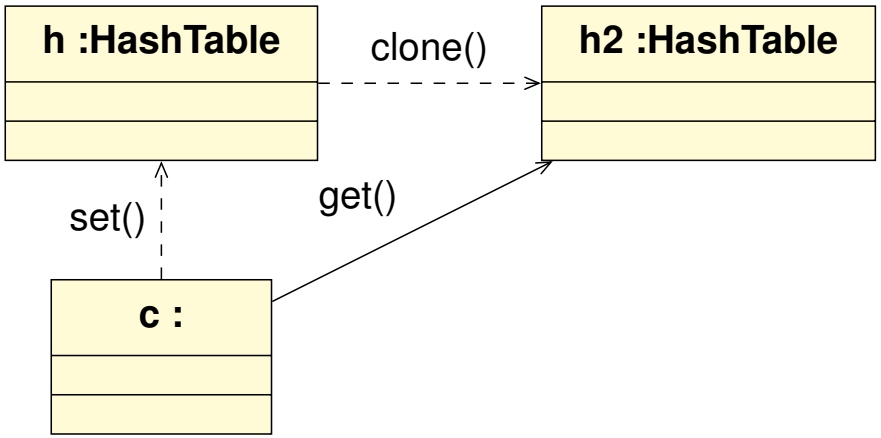

- Un seul client manipule la structure,
- s'il opère une modification,
- alors la duplication n'est pas nécessaire...

$\rightarrow$ il faut donc procéder au clonage seulement lorsque plus d'un client utilise la table.

# Exemple 2 : Copy-on-Write

## Raffinement

- On souhaite bien sûr éviter ce cas :

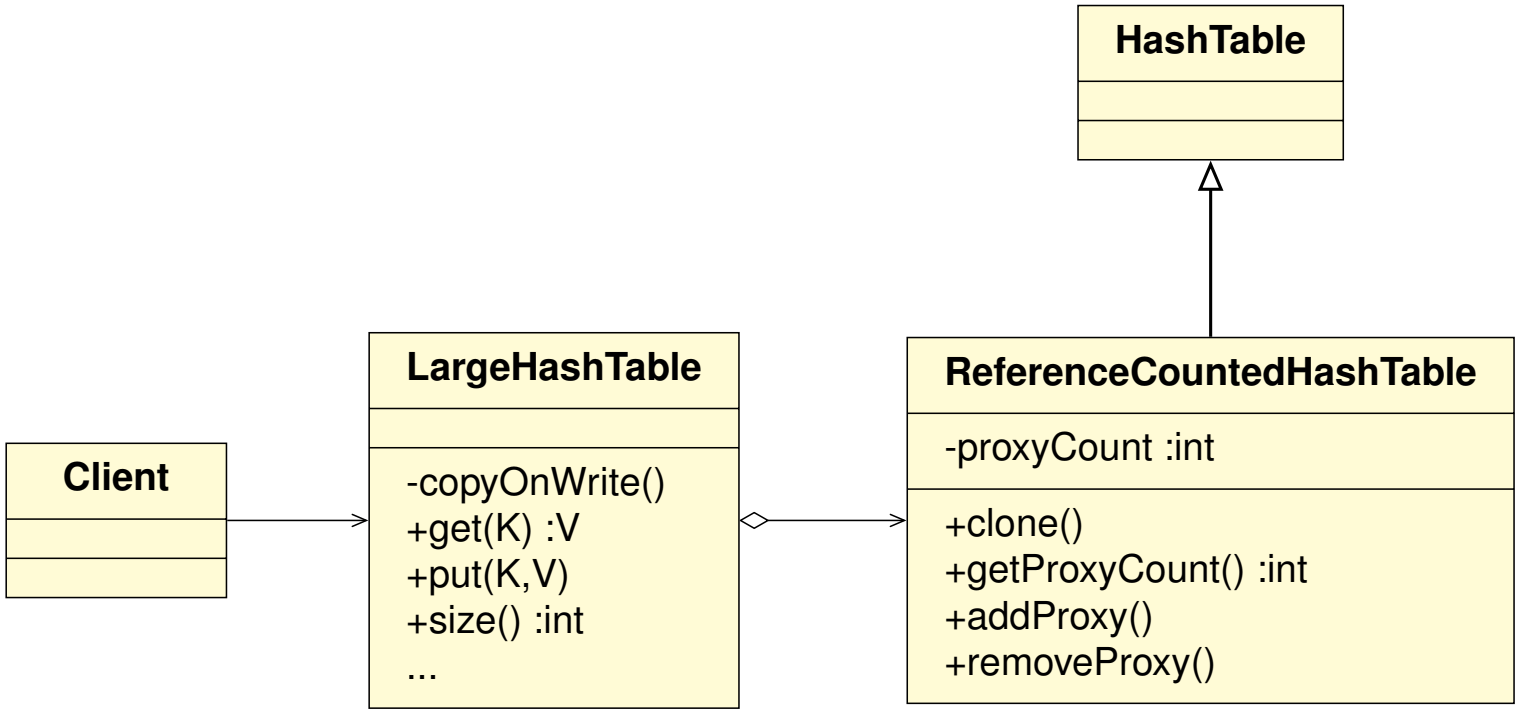



- Un seul client manipule la structure,
- s'il opère une modification,
- alors la duplication n'est pas nécessaire...

$\rightarrow$ il faut donc procéder au clonage seulement lorsque plus d'un client utilise la table.

# Exemple 2 : Copy-on-Write

## Solution




- **LargeHashTable** est le proxy,
- **ReferenceCountedHashTable** est une classe équipant la classe *HashTable* d'un comptage de référence.

# Exemple 2 : Copy-on-Write

## Implémentation 1/2

```
1  private class ReferenceCountedHashTable<K,V> extends Hashtable<K,V> {
2      private int proxyCount = 1;
3      // Constructor
4      public ReferenceCountedHashTable() {
5          super();
6      }
7      // Return a copy of this object with proxyCount set back to 1.
8      public synchronized Object clone() {
9          ReferenceCountedHashTable<K,V> copy;
10         copy = (ReferenceCountedHashTable<K,V>) super.clone();
11         copy.proxyCount = 1;
12         return copy;
13     }
14     // Return the number of proxies using this object.
15     synchronized int getProxyCount() {
16         return proxyCount;
17     }
18     // Increment the number of proxies using this object by one.
19     synchronized void addProxy() {
20         proxyCount++;
21     }
22     // Decrement the number of proxies using this object by one.
23     synchronized void removeProxy() {
24         proxyCount--;
25     }
26 }
```

# Exemple 2 : Copy-on-Write

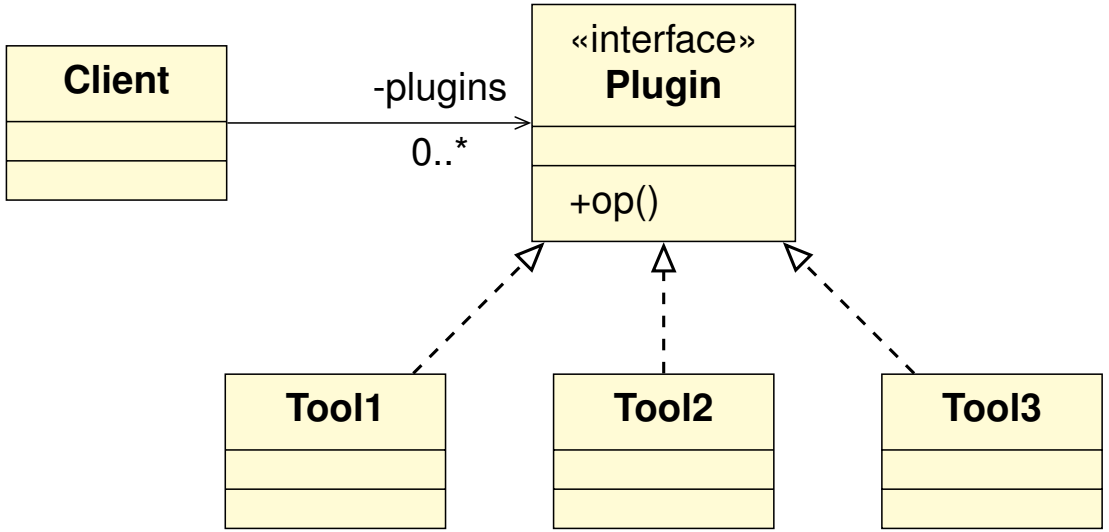
## Implémentation 2/2

```
1  public class LargeHashtable<K,V> implements Map<K,V> {
2     private ReferenceCountedHashTable<K,V> theHashTable;
3
4     public LargeHashtable() {
5        theHashTable = new ReferenceCountedHashTable<K,V>();
6     }
7     public int size() { return theHashTable.size(); }
8     public synchronized V get(K key) { return theHashTable.get(key); }
9     public synchronized V put(K key, V value) {
10       copyOnWrite();
11       return theHashTable.put(key, value);
12    }
13    public synchronized Object clone() {
14       Object copy = super.clone();
15       theHashTable.addProxy();
16       return copy;
17    }
18    private void copyOnWrite() {
19       if (theHashTable.getProxyCount() > 1) {
20          synchronized (theHashTable) {
21             theHashTable.removeProxy();
22             theHashTable =(ReferenceCountedHashTable) theHashTable.clone();
23          }
24       }
25    }
26    ...
27 }
```

# Exemple 3 : Proxy virtuel

## Situation initiale

- Un programme instancie un certain nombre d'objets lors de son démarrage :

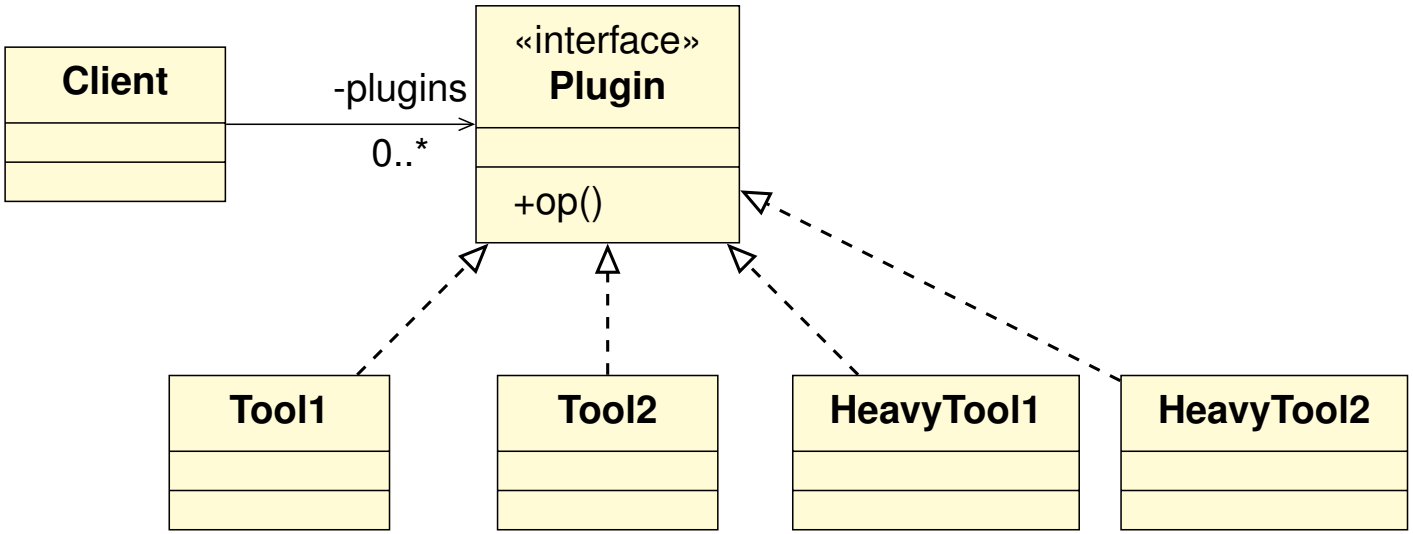



- Ces objets représentent des fonctionnalités qui ne sont pas nécessairement utiles par la suite,
- cependant le coût de cette solution au départ semble acceptable.

# Exemple 3 : Proxy virtuel

## Et le prévisible arriva ...

- Cependant, l'application reçoit au fil du temps de nombreuses contributions,
- dont des modules assez lourds devenus standards :




- Conséquence : l'application devient très longue à démarrer, même pour les utilisateurs souhaitant en faire l'usage le plus basique.

# Exemple 3 : Proxy virtuel

## Situation initiale

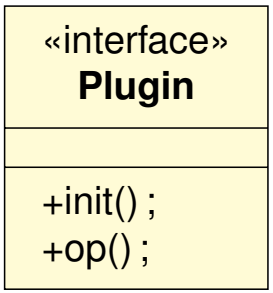
- Bien-sur, cette situation aurait pu être évitée en anticipant le problème et en concevant une architecture adaptée, utilisant par exemple une initialisation paresseuse.
- Mais ce n'est pas le cas et nous devons faire avec l'existant :
- L'architecture impose que l'application, et tous ses modules, considère que les modules sont disponibles à tout moment après l'initialisation :

```
1  void initApp() {...
2     Map<Plugin> plugins=Plugin.load("dossier/plugins"); // Builder
3     ...
4  }
5
6  ...
7  Y calcul(X x) {
8     return plugin.get("calcul").op(x); // Utilisation du plugin "calcul".
9  }
```

# Exemple 3 : Proxy virtuel

## Solutions ?

- Modifier l'application pour initialiser les objets de manière paresseuse : il faut alors déplacer du code du constructeur vers une méthode d'initialisation qui elle ne sera pas appelée systématiquement :

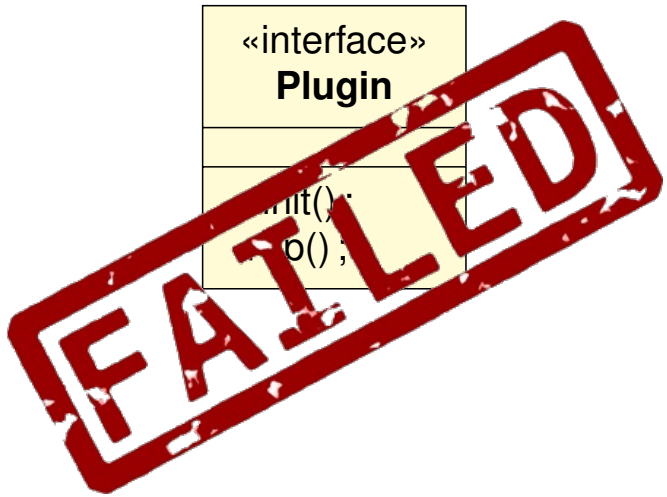



- ... mais nous changeons un contrat déjà établi publiquement !
- et le client doit être modifié pour prendre en charge le mécanisme paresseux !

# Exemple 3 : Proxy virtuel

## Solutions ?

- Modifier l'application pour initialiser les objets de manière paresseuse : il faut alors déplacer du code du constructeur vers une méthode d'initialisation qui elle ne sera pas appelée systématiquement :




- ... mais nous changeons un contrat déjà établi publiquement !
- et le client doit être modifié pour prendre en charge le mécanisme paresseux !

# Exemple 3 : Proxy virtuel

## Solutions ?

- Spécifier dans le manuel de programmation des greffons que l'initialisation doit se faire de manière paresseuse ?
- Le comportement dépendra de la bonne volonté des développeurs de greffons.
- impossible de garantir ce comportement pour les greffons qui ne sont pas empaquetés avec l'application (relecture de code).

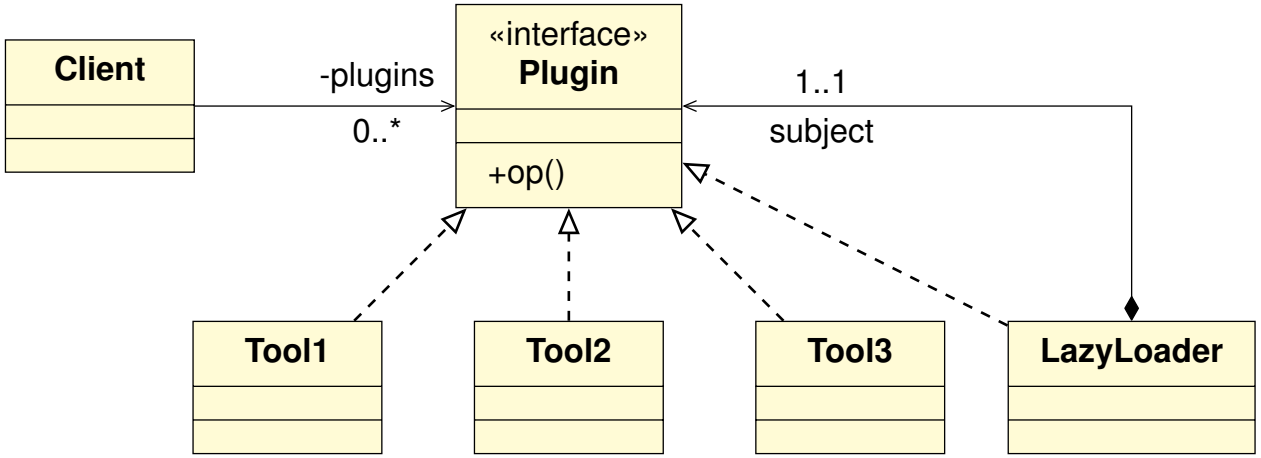
La version *proxy virtuel* du patron proxy nous permet d'ajouter de manière uniforme un comportement à des composants sans modifier l'application [9].

9. À l'exception de l'initialisation bien-sur, c'est-à-dire ici du builder.

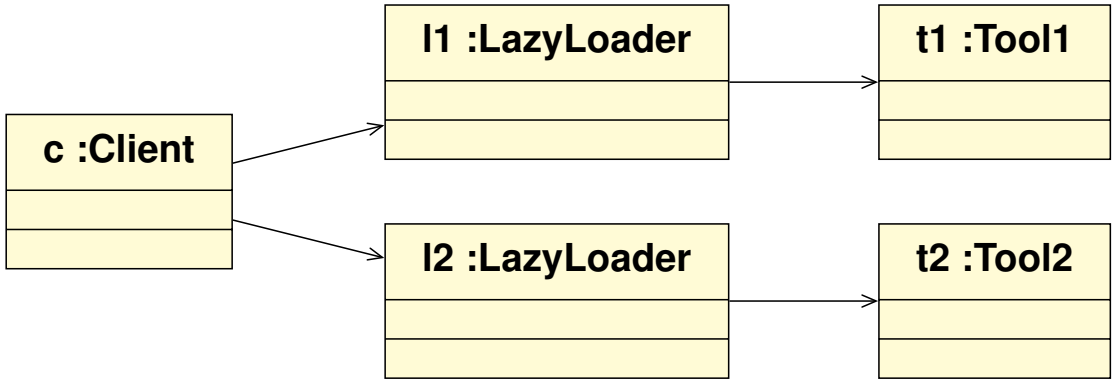
# Exemple 3 : Proxy virtuel

## Ajout du proxy dans l'infrastructure

- L'ajout n'induit aucune modification de code :




- Chaque greffon est accédé via son objet proxy :



# Exemple 3 : Proxy virtuel

## Réalisation

- Dans le builder, on remplace :

```
1    plugins["pluginXName"] = new PluginXName();
```

par

```
1    plugins["pluginXName"] = new LazyLoader("pluginXName");
```

- On ne modifie rien d'autre dans l'application.

# Exemple 3 : Proxy virtuel

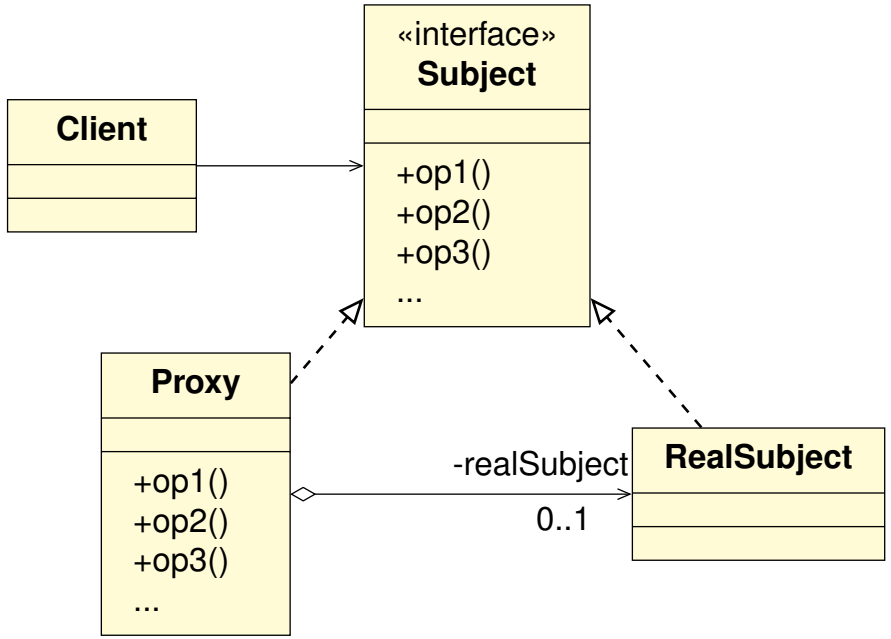
## Réalisation

- Le proxy instancie dynamiquement le greffon à sa première utilisation :

```
1  public class LazyLoader() implements Plugin {
2     private String pluginName;
3     private Plugin plugin;
4
5     public LazyLoader(String PluginName) {
6        this.PluginName=PluginName;
7     }
8
9     private synchronized Plugin getPlugin()
10     throws ClassNotFoundException, NoSuchMethodException, ... {
11       if (plugin==null) {
12          Class c=Class.forName(PluginName);
13          Constructor cons=c.getConstructor(new Class[]{});
14          plugin=(Plugin)cons.newInstance(new Object[]{});
15       }
16       return plugin;
17    }
18
19    public String operation(String par) {
20       return getPlugin().operation(par);
21    }
22 }
```

# Proxy Dynamique

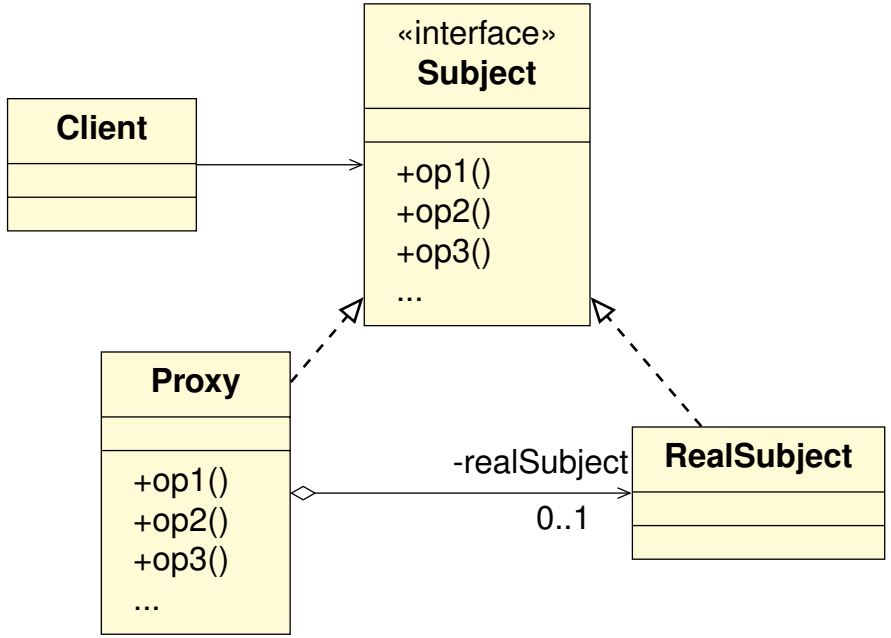
## Situation initiale




- La délégation (par composition) induit l'écriture exhaustive de toutes les méthodes de l'interface **subject** dans la classe du proxy,
- Le proxy est générique si le type utilisé pour la composition est une interface,

# Proxy Dynamique

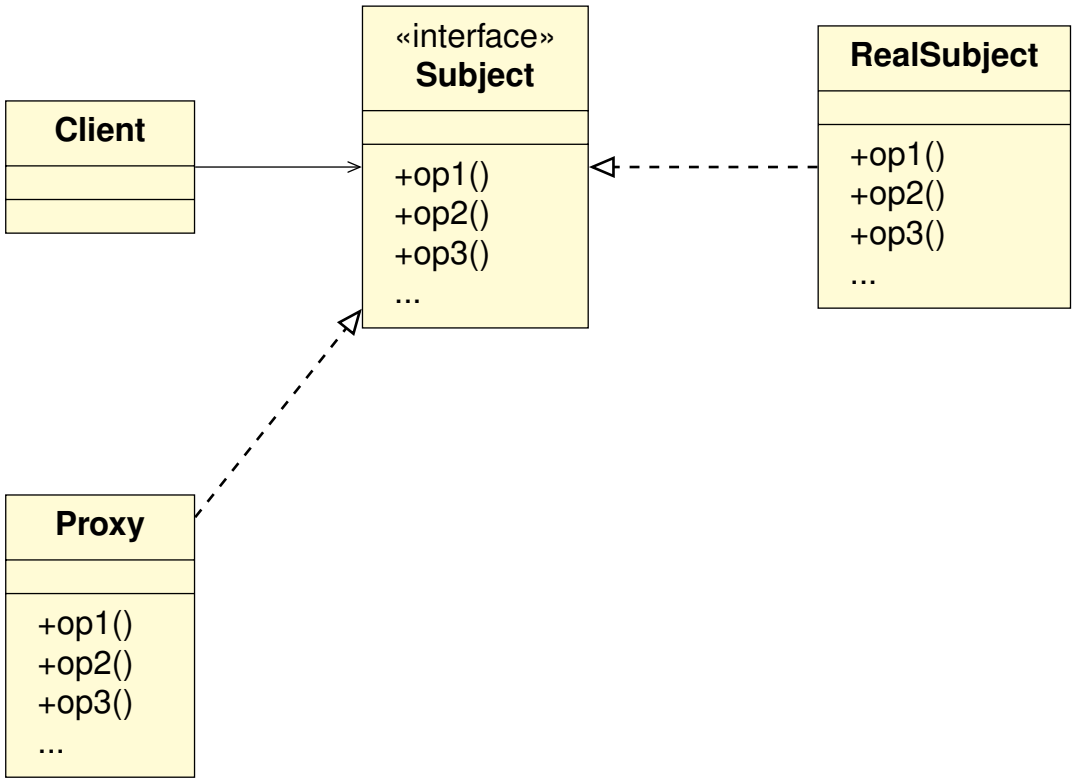
## Situation initiale




- La délégation (par composition) induit l'écriture exhaustive de toutes les méthodes de l'interface **subject** dans la classe du proxy,
- Le proxy est générique si le type utilisé pour la composition est une interface,
- le proxy peut-il être indépendant de toute interface ?

# Proxy Dynamique

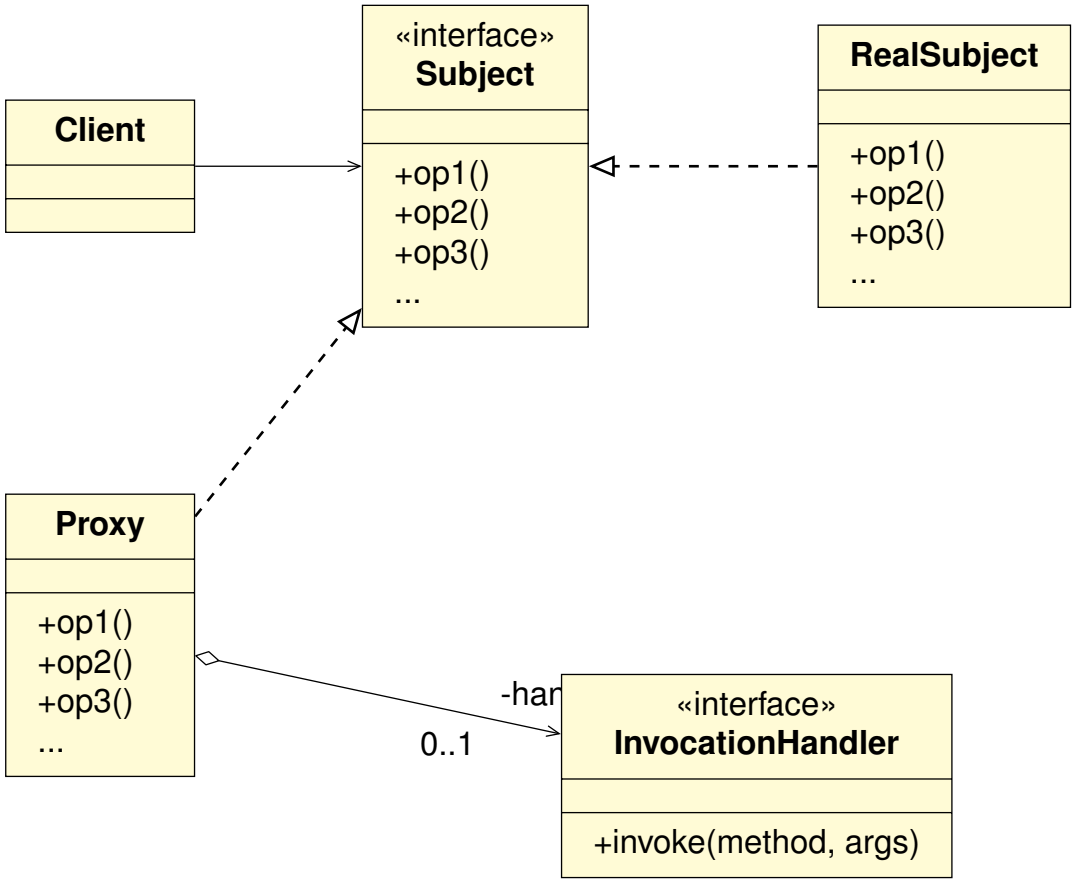
## principe




- L'API `reflection` de Java génère un proxy implémentant l'interface `Subject`.
- Quel code dans les méthodes ?

# Proxy Dynamique

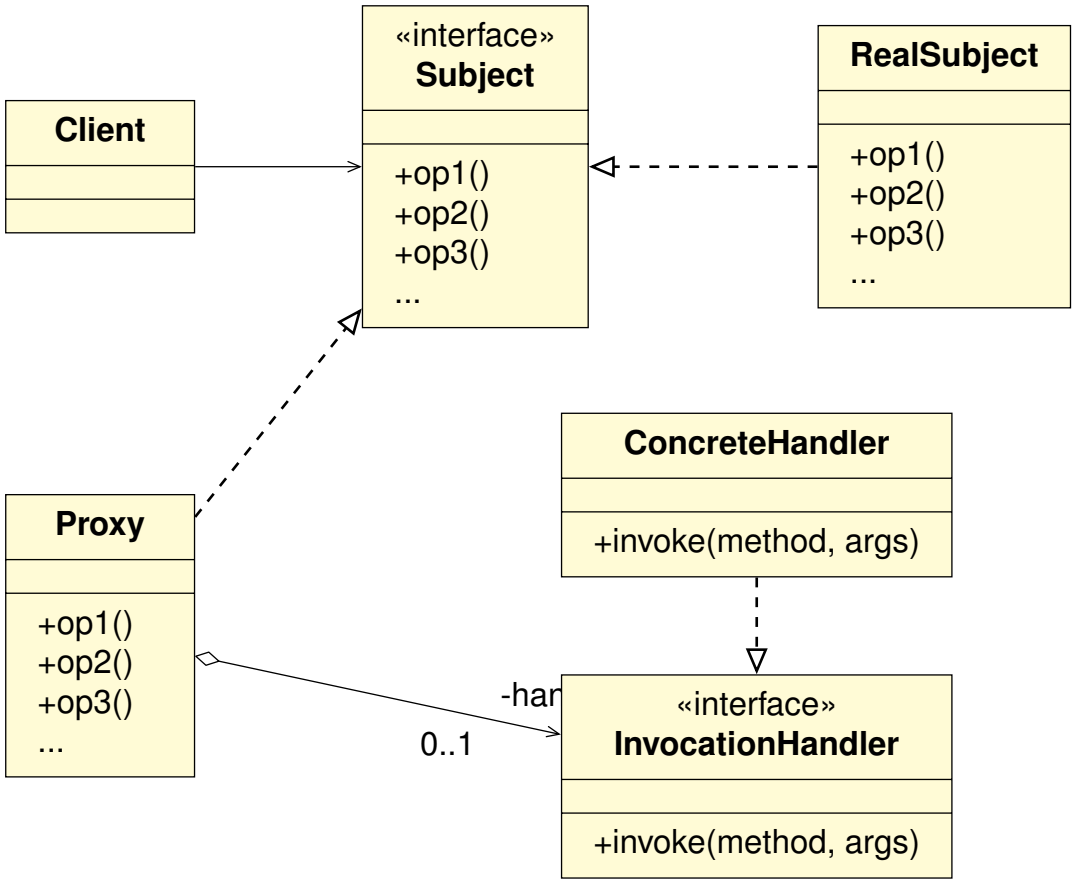
## principe




- Le code généré est générique et appelle la méthode `invoke` d'un gestionnaire d'invocation associé au proxy.

# Proxy Dynamique

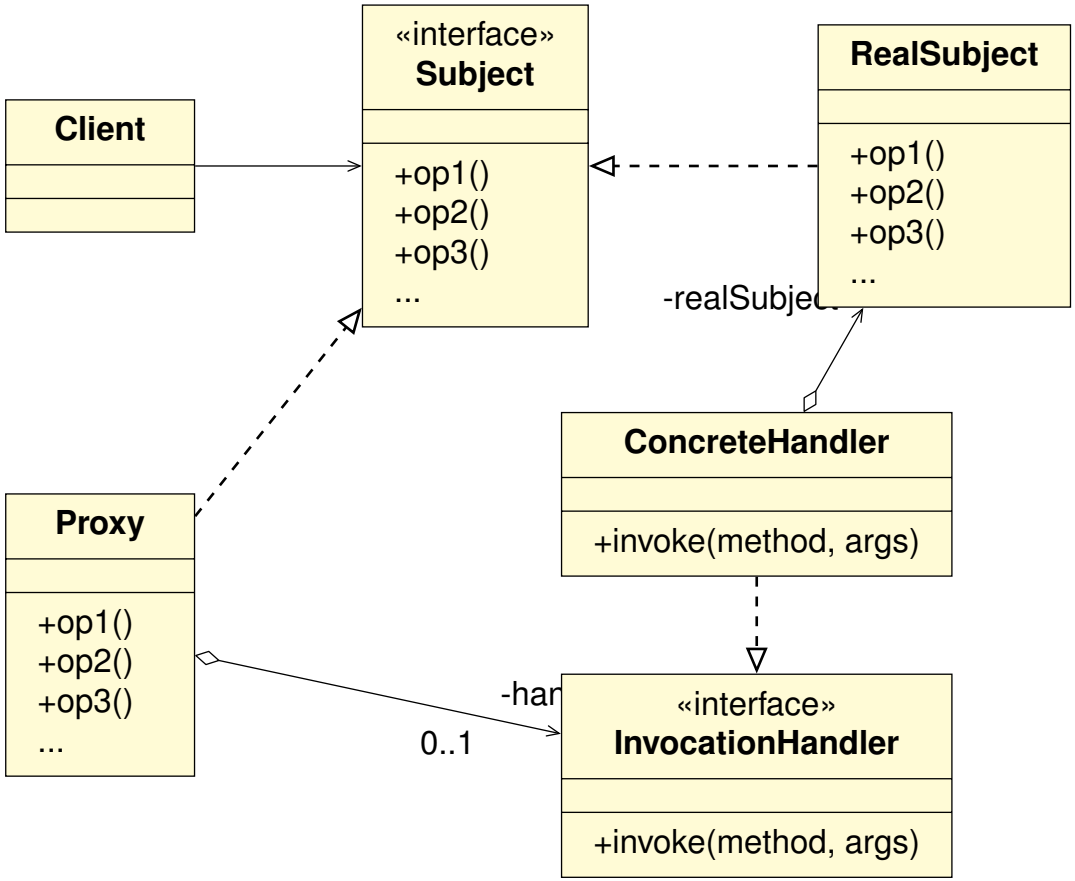
## principe




- `ConcreteHandler` est la classe que nous devons écrire.
- Mais comment appeler les méthodes du sujet réel ?

# Proxy Dynamique

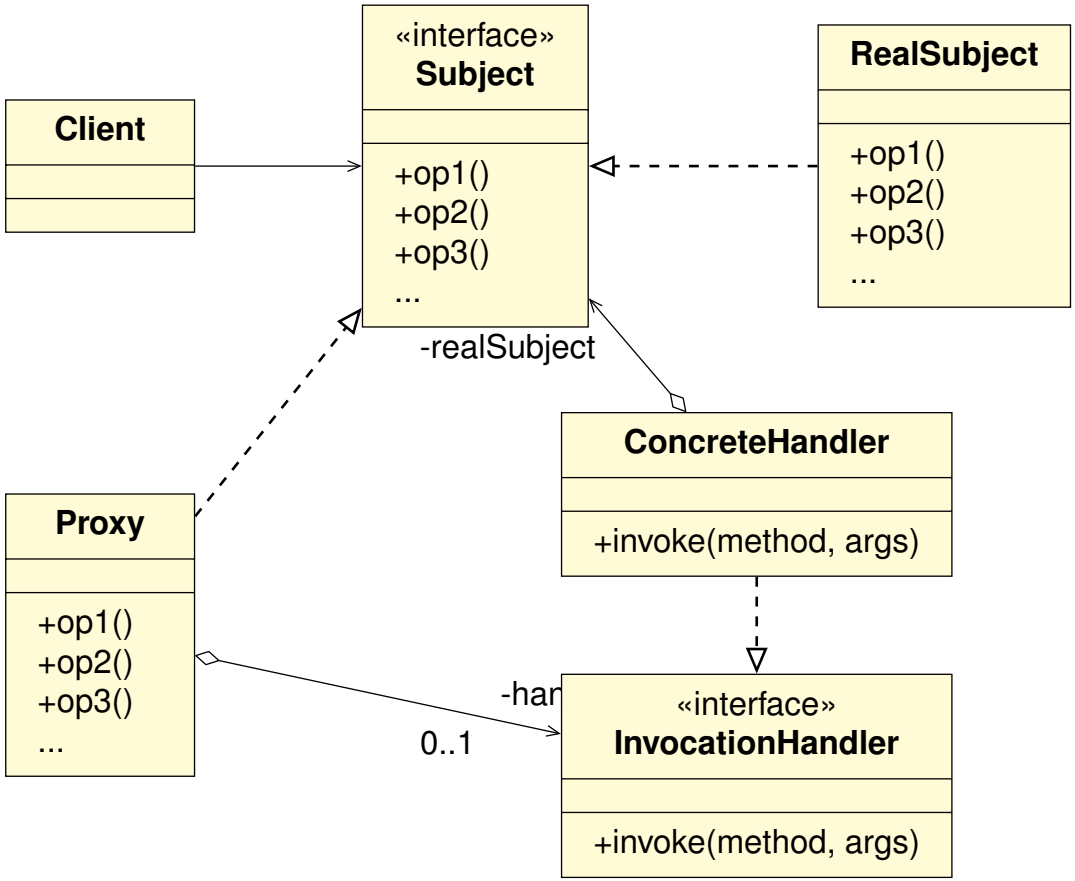
## principe




- Composer avec le sujet réel ? Pas générique du tout !

# Proxy Dynamique

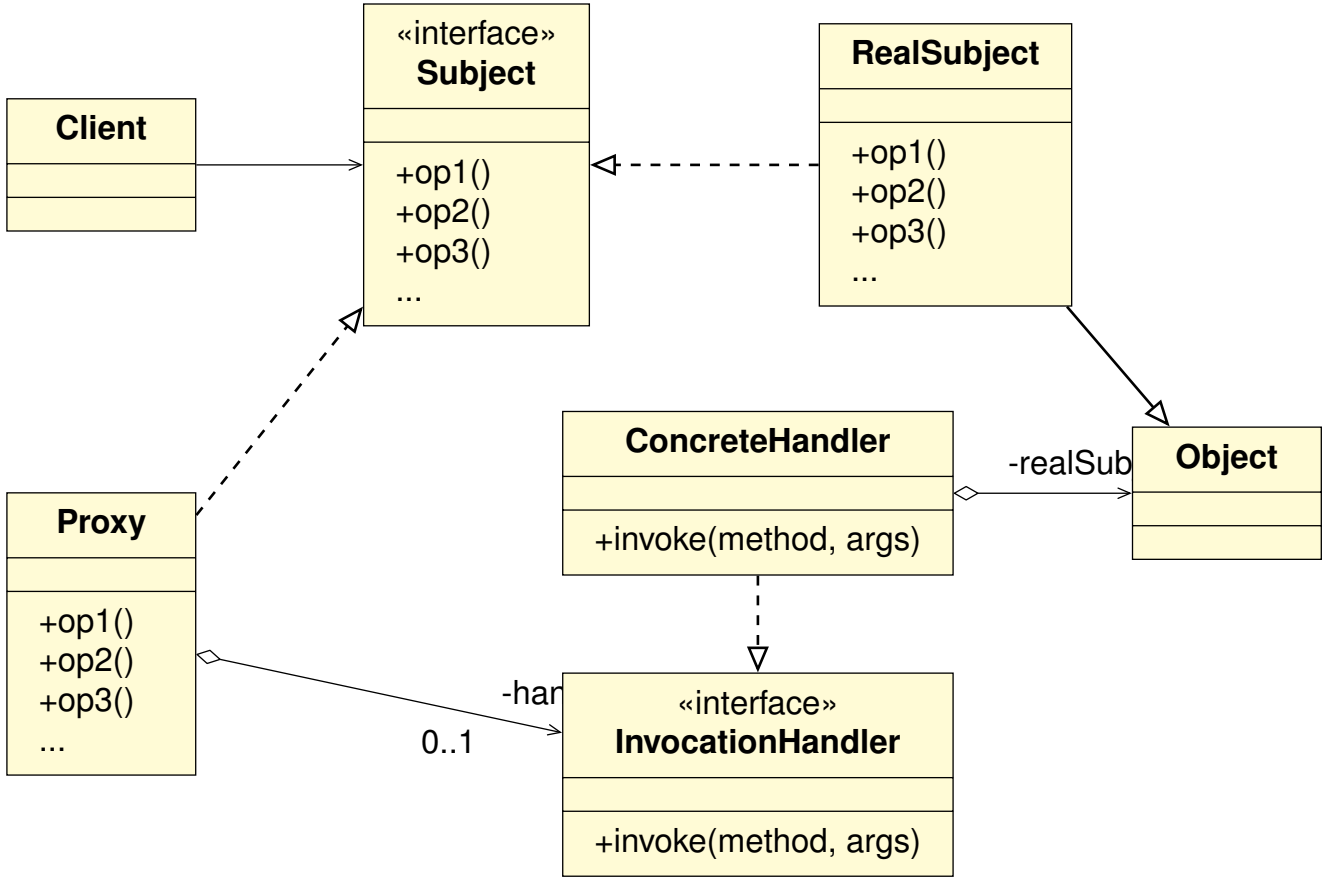
## principe




- Composer avec le sujet ? Mieux, mais nous aurions pu nous en sortir avec un simple paramètre générique `T` !

# Proxy Dynamique

## principe




- Il ne nous reste que le type `Object`.
- L'appel se fera à travers un object de la classe `Method`...

# Proxy Dynamique

## Élements de l'API Reflection

- Les classes proxy sont crées grâce à la classe `java.lang.reflect.Proxy`.
- Les classes proxy sont des classes concrètes publiques et finales, dérivées de `java.lang.reflect.Proxy`.
- Le nom d'une classe proxy n'est pas précisé. Toutefois les noms commençant par `"$Proxy"` sont réservés.
- Une classe proxy réalise exactement les interfaces spécifiées lors de sa création.

# Proxy Dynamique

## Élements de l'API Reflection

- Chaque classe proxy possède un constructeur public qui admet pour argument une réalisation de l'interface `InvocationHandler`.
- S'il est possible d'utiliser l'API `reflection` pour accéder à ce constructeur, il est plus simple d'utiliser la méthode statique `Proxy.newInstance()` qui combine la création dynamique de la classe et de l'instance .

# Proxy Dynamique

## Élements de l'API Reflection

Quelques méthodes de la classe `java.lang.reflect.Proxy` :

- Création d'une classe proxy :

```
1 public static Class getProxyClass(ClassLoader loader, Class[] interfaces
      )
2    throws IllegalArgumentException
```

- Constructeur de la classe générée :

```
1 protected Proxy(InvocationHandler ih)
```

- Tester si une classe est un proxy :

```
1 public static boolean isProxyClass(Class c)
```

# Proxy Dynamique

## Élements de l'API Reflection

Raccourci pour la création d'un objet proxy :

- La méthode :

```
1 public static Object newProxyInstance(ClassLoader loader,
2     Class[] interfaces,
3     InvocationHandler ih)
4     throws IllegalArgumentException
```

- Construit une classe proxy et retourne une instance.
- `Proxy.newProxyInstance(cl, interfaces, ih);` est équivalent à :

```
1 Proxy.getProxyClass(cl, interfaces).getConstructor(
2     new Class[] { InvocationHandler.class }
3 ).newInstance(new Object[] {ih});
```

# Proxy Dynamique

## Élements de l'API Reflection

```
1  package java.lang.reflect;
2
3  public Interface InvocationHandler {
4      Object invoke(Object proxy,
5          Method method,
6          Object[] args
7      ) throws Throwable
8  }
```

- **proxy** : une référence à l'objet proxy responsable de l'invocation.
- **method** : la classe `Method` décrit une méthode avec une signature donnée et permet d'effectuer un appel.
- **args** : un tableau d'objets représentant les paramètres effectifs.

# Exemple Proxy Dynamique

## Situation initiale

- Dans le cadre de profilage de programmes on souhaite pouvoir mesurer les temps d'exécution des méthodes de certaines classes pour détecter où le programme consomme le plus de temps.
- On ne souhaite évidemment pas modifier le code des classes existantes,
- On souhaite pouvoir réutiliser notre instrument de mesure pour toute classe existante ou à venir.
- On se propose de créer un proxy dynamique capable de s'intercaler entre un client et une classe quelconque.

# Exemple Proxy Dynamique

## Réalisation

```
1  package proxy.dynamic;
2  import java.lang.reflect.*; // {InvocationHandler, Method, Proxy}
3
4  public class ImpatientHandler implements InvocationHandler {
5      private Object target;
6      private Impatient Handler(Object target) { this.target = target; }
7
8      public static Object newInstance(Object target) {
9          ClassLoader loader = target.getClass().getClassLoader();
10         Class[] interfaces = target.getClass().getInterfaces();
11         return Proxy.newProxyInstance(loader, interfaces, new Impatient
               Handler (target));
12     }
13
14     public Object invoke(Object proxy, Method m, Object[] args) throws
           Throwable {
15         Object result;
16         long t1 = System.currentTimeMillis();
17         result = m.invoke(target, args);
18         long t2 = System.currentTimeMillis();
19         if (t2 - t1 > 10) {
20             System.out.println(">␣It␣takes␣" + (t2 - t1) + "␣millis␣to␣invoke␣
                   "
21                 + m.getName()+"()␣with");
22             for (int i = 0; i < args.length; i++)
23                 System.out.println(">␣arg[" + i + "]:␣" + args[i]);
24         }
25         return result;
26     }
27 }
```

# Exemple Proxy Dynamique

## Réalisation

### Utilisation :

```
1  package proxy.dynamic;
2  import java.util.HashSet;
3  import java.util.Set;
4
5  public class TestDynamicProxy {
6     // Client 2 :
7     public static Set<Apple> setup() {
8        Set<Apple> realSubject = new HashSet<Apple>();
9        // Le proxy+gest. d'invocation :
10       Set<Apple> proxy = (Set<Apple>)ImpatientHandler.newInstance(realSubject)
               ;
11       return proxy; // Ni vu ni connu !
12    }
13
14    // Client 1 :
15    public static void main(String[] args) {
16       // Le sujet :
17       Set<Apple> set = setup();
18
19       set.add(new GoodApple("Cox␣Orange"));
20       set.add(new BadApple("Lemon"));
21       set.add(new GoodApple("Prems"));
22       System.out.println("The␣set␣contains␣" + set.size() + "␣things.");
23    }
24 }
```

# Conclusion

- Les implémentations du pattern PROXY produisent un objet intermédiaire qui gère l'accès à un objet cible.
- Un objet proxy peut dissimuler aux clients les changements d'état d'un objet cible, comme dans le cas d'une image qui nécessite un certain temps pour se charger.
- Le problème est que ce pattern s'appuie habituellement sur un couplage étroit entre l'intermédiaire et l'objet cible.
- Dans certains cas, la solution consiste à utiliser un proxy dynamique : lorsque la classe d'un objet implémente des interfaces pour les méthodes que vous voulez intercepter, vous pouvez envelopper l'objet dans un proxy dynamique et faire en sorte que votre code s'exécute avant/après le code de l'objet enveloppé ou à sa place.

# Principes respectés

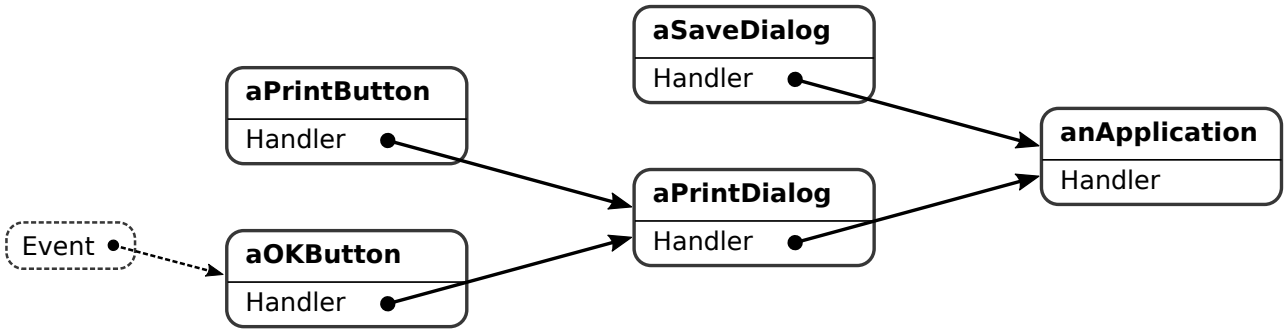
- **S.R.P.** ; ?
- **O.C.P.** : ?
- **L.S.P.** : ?
- **I.S.P.** : ?
- **D.I.P.** : ?

# Le patron Chaîne de responsabilité

Découple l'émetteur d'une requête du récepteur et permet à plus d'un objet de participer à son traitement.

# Chaîne de responsabilités

## Motivations - Exemple 1




- Dans une interface graphique, tout composant peut déclencher l'affichage de l'aide contextuelle (en pressant `F1` par exemple)
- Le composant qui a le focus reçoit la demande,
- s'il ne possède pas d'aide contextuelle, il transmet la demande à son conteneur,
- le conteneur fait de même jusqu'au niveau le plus haut de l'application ou jusqu'à ce que l'un des composants dispose d'une aide.

# Chaîne de responsabilités

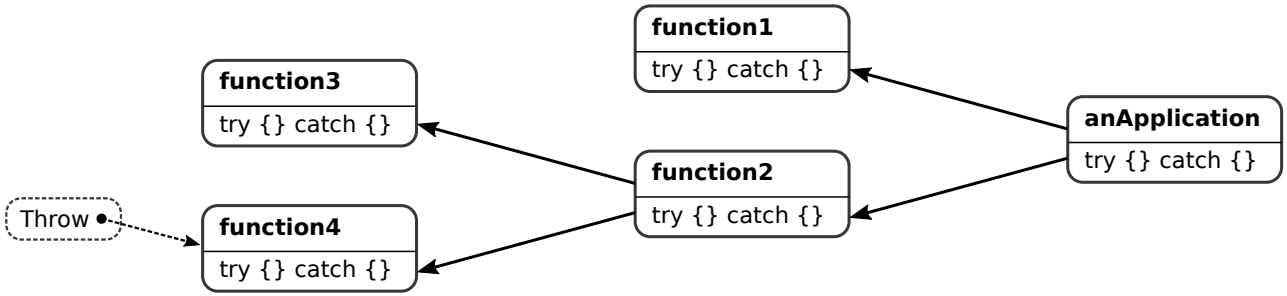
## Motivations - Exemple 2

- Le fonctionnement correspondant à celui d'un mecanisme implicite utilisé en permanence en programmation...

# Chaîne de responsabilités

## Motivations - Exemple 2

- Le fonctionnement correspondant à celui d'un mecanisme implicite utilisé en permanence en programmation...
- Celui des exceptions :




- L'exception remonte la pile d'appel jusqu'à rencontrer un bloc try/catch qui la traite (arrêt) ou la relance (poursuite).

# Chaîne de responsabilités

## Motivations

- Dans les exemples précédents, la configuration de la chaîne de traitement est déterminée par l'état de l'application.
- Le patron autorise une configuration déterminable par les choix de l'utilisateur, par ex. : chaînage de filtre de traitement vidéo.
- Des exemples concrets incluent : pile d'authentification PAM [10] (fichier de configuration), les chaînes de traitement de requêtes serveur JEE (javax.servlet.filter), etc.

10. *cf* votre cours d'administration système au S1.

# Participants du patron **Chaîne de responsabilités**

- **Client** : Initie la requête sur un objet *ConcreteHandler* de la chaîne.
- **Handler** :
  - Fournit l'interface standardisant le vocabulaire d'opérations (listes des requêtes possibles) que le client peut demander et que les objets *ConcreteHandler* peuvent se transférer entre-eux.
  - Fournit également le lien de chaînage [11].
- **ContreteHandler** : Traite une requête pour laquelle il est responsable et peut déléguer à ses successeurs.

11. Une interface et un lien à la fois... hum.

# Le patron **Chaîne de responsabilités**

## Aussi connu comme

–

## Intention

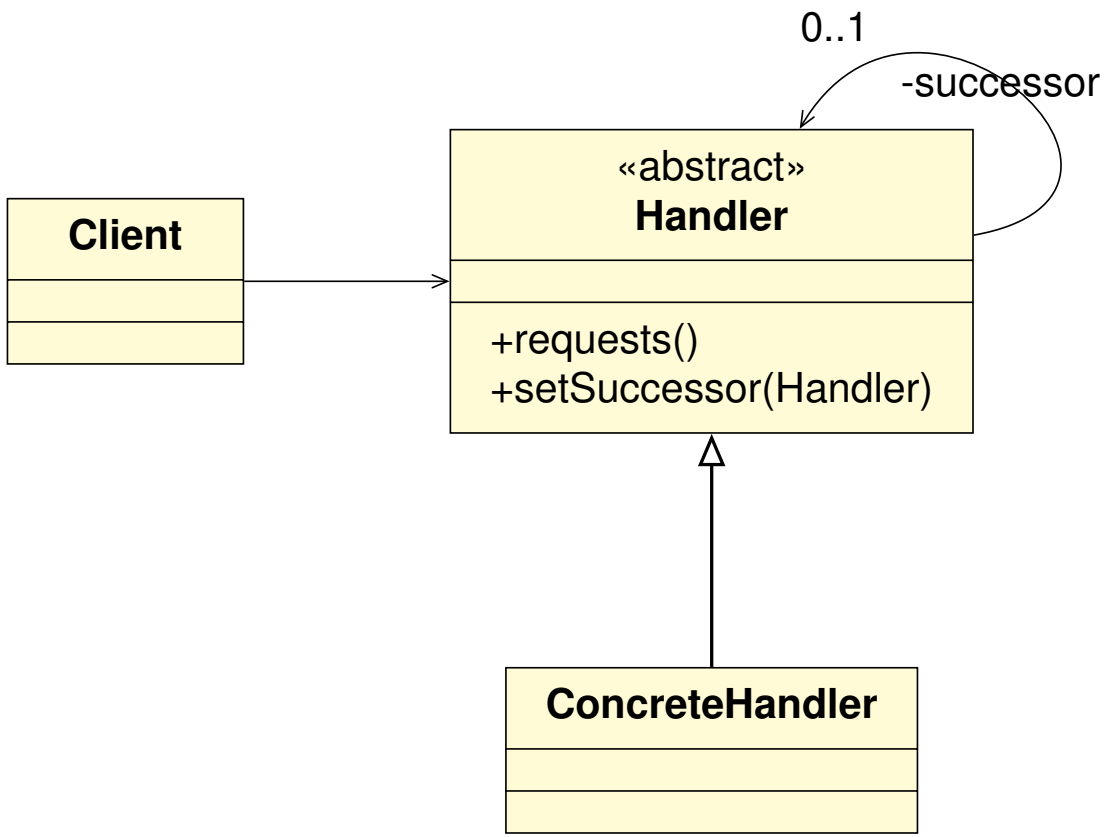
- Permettre à un ou plusieurs objets de traiter une requête séquentiellement (par chaînage).
- Éviter le couplage entre l'émetteur et les récepteurs.

## Motivation

- Service rendu par une chaîne de traitement dont la configuration est indéterminée (car dynamique).
- Le client (émetteur de la requête) ne doit pas avoir à tenir compte de cette structure pour trouver l'objet responsable du service.

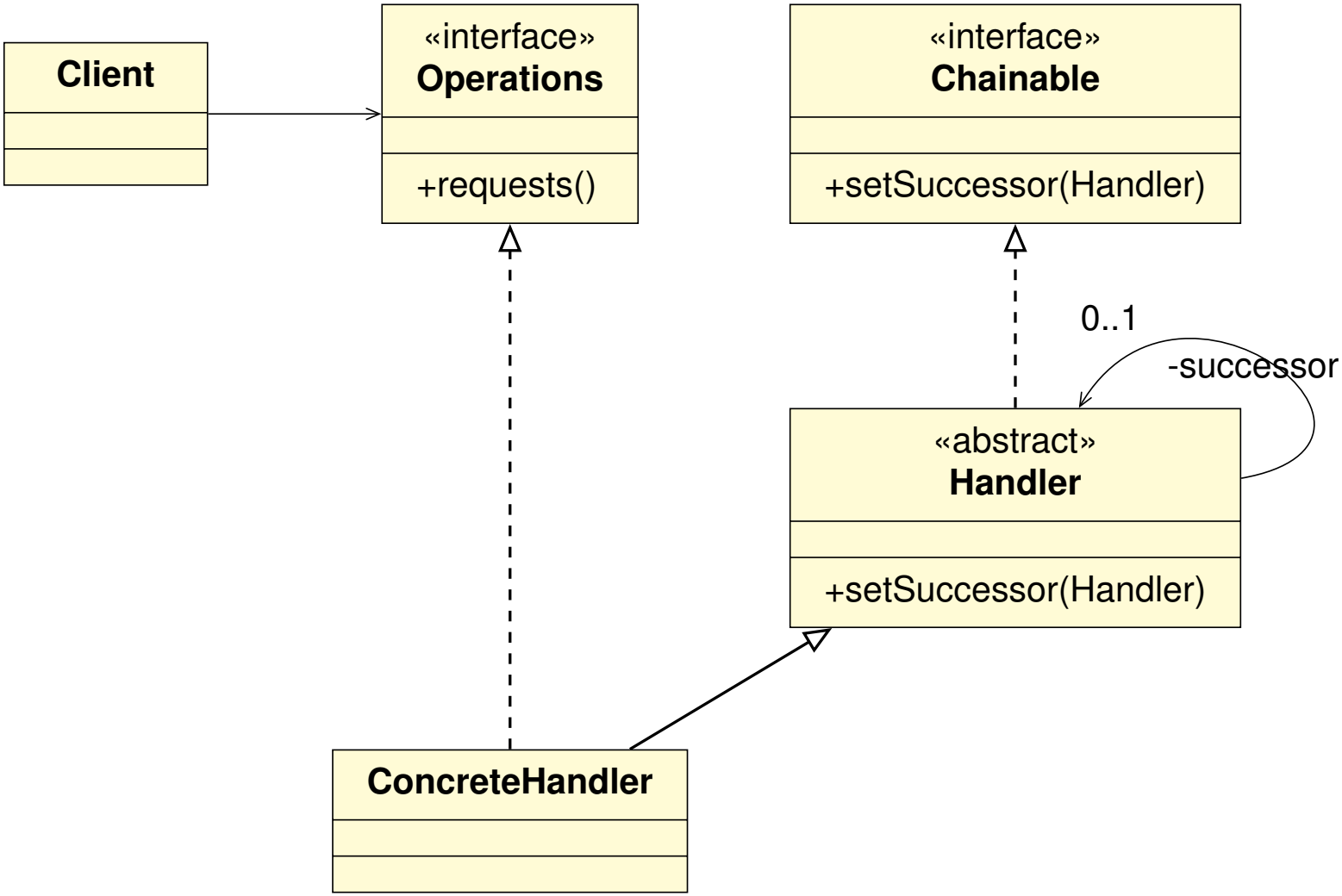
# Chaîne de responsabilités

## Schéma de principe V1



# Chaîne de responsabilités

## Schéma de principe V2



# Chaîne de responsabilités

## Exemple API AWT (JDK 1.0, obsolète)

```
1  public boolean action(Event event, Object obj) {
2     if (event.target == test_button)
3        doTestButtonAction();
4     else if (event.target == exit_button)
5        doExitButtonAction();
6     else
7        return super.action(event,obj);
8     return true;
9     // Return true to indicate the event has been
10    // handled and should not be propagated further.
11 }
```

- Propagation d'événements vue précédement.
- PB d'efficacité : beaucoup d'évènements générés, peu traités (ex : MOUSE_MOVE) ;
- PB flexibilité : configuré par la hiérarchie des composants, les événements ne peuvent être traités que par des composants graphiques.
- → Patron *Observateur* plus approprié.

# Chaîne de responsabilités

## Exemple API AWT (JDK 1.0, obsolète)

```
1  public boolean action(Event event, Object obj) {
2     if (event.target == test_button)
3        doTestButtonAction();
4     else if (event.target == exit_button)
5        doExitButtonAction();
6     else
7        return super.action(event,obj);
8     return true;
9     // Return true to indicate the event has been
10    // handled and should not be propagated further.
11 }
```

- Propagation d'événements vue précédement.
- PB d'efficacité : beaucoup d'évènements générés, peu traités (ex : MOUSE_MOVE) ;
- PB flexibilité : configuré par la hiérarchie des composants, les événements ne peuvent être traités que par des composants graphiques.
- $\rightarrow$ Patron *Observateur* plus approprié.

# Chaîne de responsabilités

## Exemple 2 : système de capteurs

- Conception d'un logiciel pour l'audit (monitorage) d'un système de securité utilisant divers capteurs (fumée, feu, déplacement, etc.), chacun transmettant son état à un ordinateur central.
- Pour chaque capteur, un objet `Sensor` sera instancié qui connaît les valeurs limites ç ne pas dépasser.
- Lorsqu'une valeur mesurée sort des limites, une action doit être déclenchée.
- Mais certains types d'alerte peuvent être conditionnés par des combinaisons de capteurs (fumée+infrarouge pour distinguer un incendie d'un fumeur ou autre...)

# Chaîne de responsabilités

## Exemple 2 : système de capteurs

Solution :

- Utiliser le patron Chaîne de Responsabilité. Regrouper les objets dans une arborescence hiérarchique miroir des zones physiques à protéger. Définir les murs, plancher, ouverture, salle, ouvrages, sites et autres objets comme composant d'un conteneur hiérarchique.
- L'alarme générée par un senseur remonte cette hiérarchie jusqu'à ce qu'un des objets le traite ou qu'elle soit annulée...

# Chaîne de responsabilités

## Exemple 3 : système d'approbation d'achats

- Nous devons concevoir un logiciel pour un système d'approbation d'achats où l'approbation dépend du montant total de l'achat.
- L'instance (au sens juridique) qui donne l'approbation en fonction du montant total peut changer à tout moment, ainsi que les conditions d'acceptation.
- Le logiciel doit être assez flexible pour tenir compte de ces éventuels changements.

# Chaîne de responsabilités

## Exemple 3 : système d'approbation d'achats

Solution :

- Utiliser une chaîne de responsabilité ou les objets représentant une demande d'achat font suivre la demande de validation à un objet de type Approbation. En fonction du montant de la commande cet objet peut valider la requête ou la faire remonter à l'instance de validation suivante de la chaîne.
- L'instance d'approbation à tous les niveaux de la chaîne peut être modifiée sans qu'en soit affectée (le code de) la demande initiale.

# Chaîne de responsabilités

## Considérations techniques : multiples requêtes

- Le patron présente une requête unique :

```
1 public interface Handler {
2     public void handleRequest();
3 }
```

- Mais rien n'empêche une CdR de traiter plusieurs types de requêtes. Ex. :

```
1 public interface Handler {
2     public void handleHelp();
3     public void handlePrint();
4     public void handleFormat();
5 }
```

# Chaîne de responsabilités

## Considérations techniques : multiples requêtes

- Chaque maillon pouvant choisir de traiter lui-même une requête d'un type donné ou bien laisser passer :

```
1  public class ConcreteHandler implements Handler {
2     private Handler successor;
3     public ConcreteHandler(Handler successor) {
4        this.successor = successor; // Not me !
5     }
6     public void handleHelp() {
7        // We handle help ourselves, so help code is here.
8     }
9     public void handlePrint() {
10       successor.handlePrint(); // Not me !
11    }
12    public void handleFormat() {
13       successor.handleFormat(); // Not me !
14    }
15 }
```

- ... mais l'ajout d'un type de requête oblige à modifier tous les maillons.

# Chaîne de responsabilités

## Considérations techniques : multiples requêtes

Solution 2 :

- Chaque maillon pouvant choisir de traiter lui-même une requête d'un type donné ou bien laisser passer :

```
1  public interface HelpHandler {
2      public void handleHelp();
3  }
4  public interface PrintHandler {
5      public void handlePrint();
6  }
7  public interface FormatHandler {
8      public void handleFormat();
9  }
```

- Un handler concret peut maintenant implanter une (ou plusieurs de ces) interface(s).
- Mais un successeur par type de requête ou bien une/des interface(s) chapeau.

# Chaîne de responsabilités

## Considérations techniques : multiples requêtes

Autres solutions (non développées) :

- Passer le type de la requête sous forme de paramètre :

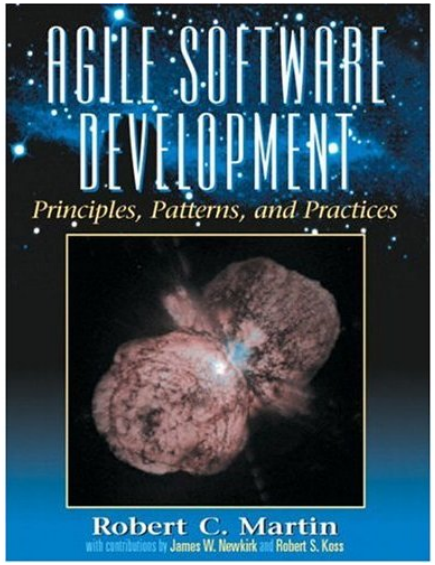
```
1 public interface Handler {
2     public void handleRequest(String request);
3 }
```

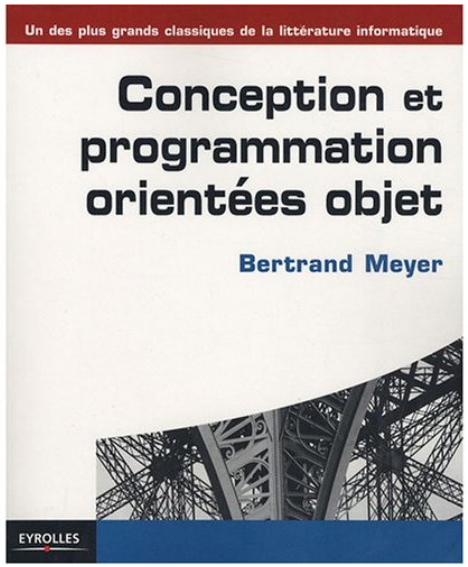
- ... bof !
- Utiliser un visiteur ? (si les types de traitement sont sous contrôle d'une seule bibliothèque).

# Principes respectés

- **S.R.P.** : facilité par la répartition modulaire des traitements possibles,
- **O.C.P.** : forcément car couplage faible et configuration dynamique.
- **L.S.P.** : repose sur le respect de ce principe, d'où l'importance de l'interface de contrat.
- **I.S.P.** : petit plus si l'on sépare les interfaces de service et l'interface de maintenance du patron.
- **D.I.P.** : si l'interface de service est conçue sans couplage avec les problématiques de modules.

# Quelques références




**Agile Software Development : Principles, Patterns, and Practices.**, *Robert C. Martin*, Prentice Hall (2002).
ISBN-13 : 978-0135974445.

**Conception et programmation orientées objets**, *Bertand Meyer*, Eyrolles (3 janvier 2008).
ISBN-13 : 978-2212122701.