
NFP121, Cnam/Paris

Cours 7

Introspection

jean-michel Douin, douin au cnam point fr
version : 13 Novembre 2019

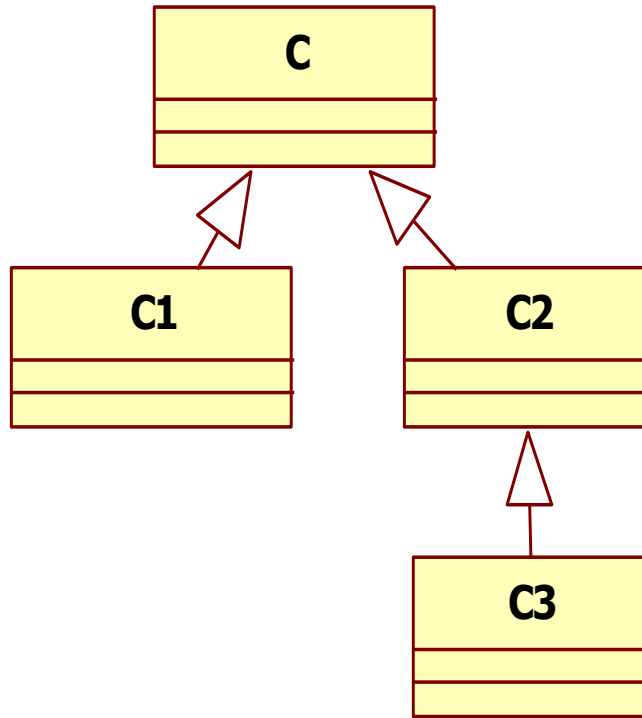
Notes de cours

Bibliographie

- **Cours de J-L Dewez NFP121**
- **Reflection in Java**
 - <http://download.oracle.com/javase/tutorial/reflect/index.html>
- **DynamicProxy**
 - <http://download.oracle.com/javase/1.3/docs/guide/reflection/proxy.html>
- **JavaBeans**
 - <http://java.sun.com/developer/onlineTraining/Beans/JBeansAPI/index.html>
- **JavaBeans + BDK**
 - <http://java.sun.com/developer/onlineTraining/Beans/JBShortCourse/exercises/Wayne/help.html>
 - <http://docs.cs.cf.ac.uk/pdfs/615.pdf>

- **La classe Class<?>**
 - Pourquoi faire ?
 - **JavaBean**
 - **Le BDK**
- **Le paquetage java.lang.reflect**
 - **Field**
 - **Constructor**
 - **Method**
- **Deux patrons et introspection**
 - **Visiteur**
 - **Procuration**
 - **Issu d'un fichier de configuration**
- **Vers une séparation de la configuration/utilisation**

Avant compilation: des classes !



Des classes et des relations
entre classes sont définies
par une édition de texte

Ce texte sera compilé

Après compilation:des objets !

v0 : C

v : C1

Au 'runtime'

Il n'existe que des objets de ces classes

```
C0 v0 = new C1();
```

```
C1 v;
```

Dont un objet spécifique pour chaque classe chargée !

v0 : C

C.class

v : C1

C1.class

C2.class

C3.class

Parmi les objets figure un objet-Class
pour chaque classe chargée.

Cela vaut pour

String.class par exemple

ou

Object.class

Object.class

Cet objet est de classe Class

Class

```
+toString(): String
+forName(className: String): Class
+newInstance(): Object
+isInstance(obj: Object): boolean
+isInterface(): boolean
+isArray(): boolean
+isPrimitive(): boolean
+getName(): String
+getClassLoader(): ClassLoader
+getSuperclass(): Class
+getPackage(): Package
+getInterfaces(): Class
+getComponentType(): Class
+getModifiers(): int
+getSigners(): Object
+getDeclaringClass(): Class
+getClasses(): Class
+getFields(): Field
+getMethods(): Method
+getConstructors(): Constructor
+getField(name: String): Field
+getMethod(name: String, parameterTypes: Class): Method
+getConstructor(parameterTypes: Class): Constructor
+getDeclaredClasses(): Class
+getDeclaredFields(): Field
+getDeclaredMethods(): Method
```

Il est construit
au chargement d'une classe
dans la JVM
Dans un certain classLoader

Une méthode de classe pour charger de nouvelles classes

Class

+toString(): String
+forName(className: String): Class
+newInstance(): Object
+isInstance(obj: Object): boolean
+isInterface(): boolean
+isArray(): boolean
+isPrimitive(): boolean
+getName(): String
+getClassLoader(): ClassLoader
+getSuperclass(): Class
+getPackage(): Package
+getInterfaces(): Class
+getComponentType(): Class
+getModifiers(): int
+getSigners(): Object
+getDeclaringClass(): Class
+getClasses(): Class
+getFields(): Field
+getMethods(): Method
+getConstructors(): Constructor
+getField(name: String): Field
+getMethod(name: String, parameterTypes: Class): Method
+getConstructor(parameterTypes: Class): Constructor
+getDeclaredClasses(): Class
+getDeclaredFields(): Field
+getDeclaredMethods(): Method

public static Class **forName**(String className)
throws ClassNotFoundException

Retourne l'objet-Class associé au type dont
le nom est fourni.
(nom qualifié)

Exemple

Class c = Class.forName("cours7.Exemple")

ClassLoader cl = c.getClassLoader()

Tout objet peut être interrogé sur sa classe

Object

+getClass(): Class
+hashCode(): int
+equals(obj: Object): boolean
#clone(): Object
+toString(): String
+notify()
+notifyAll()
+wait(timeout: long)
+wait(timeout: long, nanos: int)
+wait()
#finalize()

A l'exécution il est possible de demander à tout objet quelle est sa classe, i.e. la classe constatée

Exemple

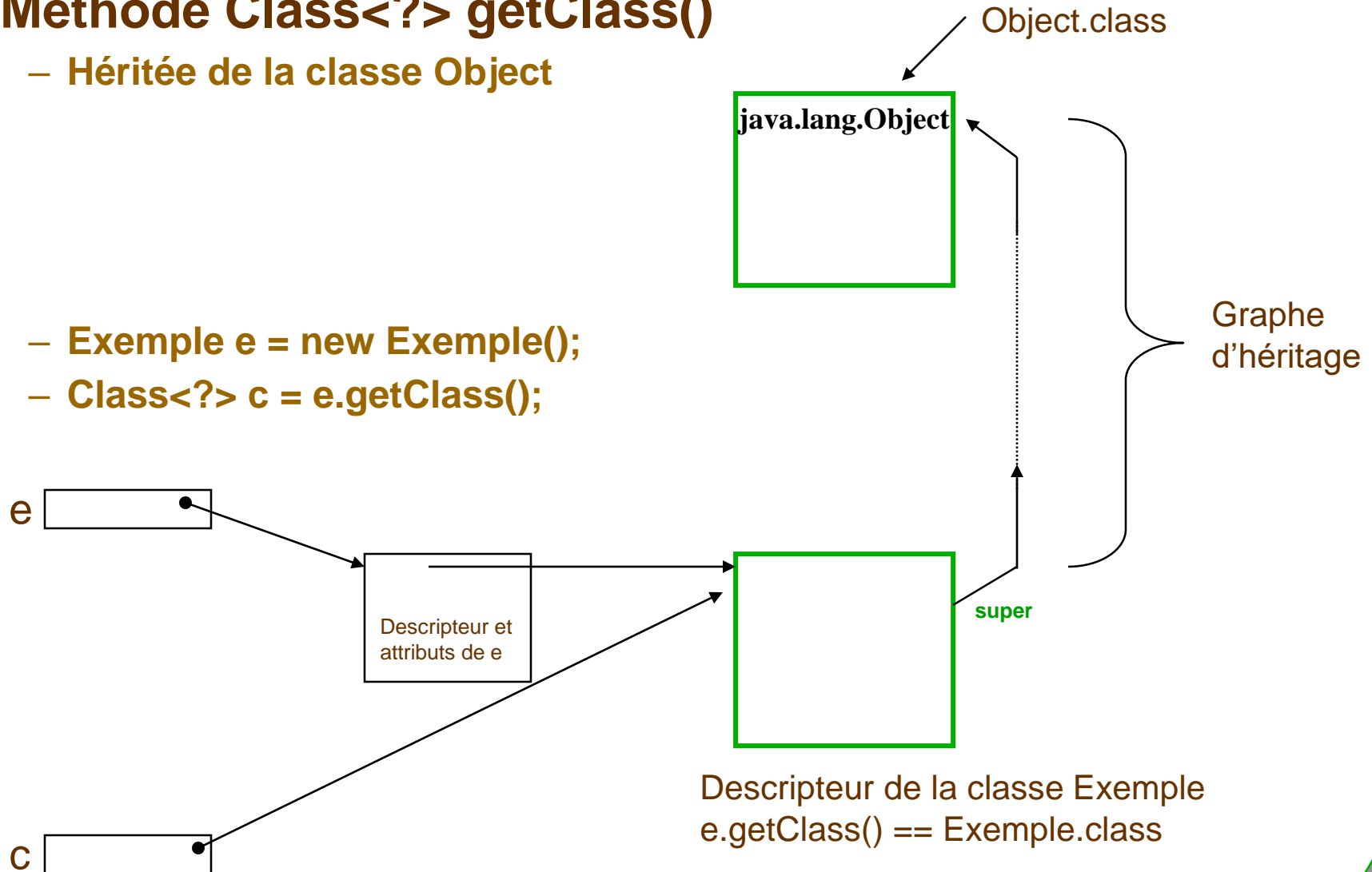
`v.getClass()==C1.class`

`String.class.getClass() == Class.class`

Classe Objet, Class, getClass, super

- Méthode `Class<?> getClass()`
 - Héritée de la classe `Object`

- Exemple `e = new Exemple();`
- `Class<?> c = e.getClass();`



L'objet de classe Class lui-même peut-être interrogé

Class
<ul style="list-style-type: none">+toString(): String+forName(className: String): Class+newInstance(): Object+isInstance(obj: Object): boolean ←+isInterface(): boolean+isArray(): boolean+isPrimitive(): boolean+getName(): String+getClassLoader(): ClassLoader+getSuperclass(): Class+getPackage(): Package+getInterfaces(): Class+getComponentType(): Class+getModifiers(): int+getSigners(): Object+getDeclaringClass(): Class+getClasses(): Class+getFields(): Field+getMethods(): Method+getConstructors(): Constructor+getField(name: String): Field+getMethod(name: String, parameterTypes: Class): Method+getConstructor(parameterTypes: Class): Constructor+getDeclaredClasses(): Class+getDeclaredFields(): Field+getDeclaredMethods(): Method

boolean **isInstance**(Object obj)

Cette méthode permet de savoir si obj est d'une sous-classe de la classe dont l'objet-Class est ici interrogé.

Exemple

v0.getClass().isInstance(v)

Il peut aussi servir à créer dynamiquement un nouvel objet

Class
<ul style="list-style-type: none">+toString(): String+forName(className: String): Class+newInstance(): Object+isInstance(obj: Object): boolean+isInterface(): boolean+isArray(): boolean+isPrimitive(): boolean+getName(): String+getClassLoader(): ClassLoader+getSuperclass(): Class+getPackage(): Package+getInterfaces(): Class+getComponentType(): Class+getModifiers(): int+getSigners(): Object+getDeclaringClass(): Class+getClasses(): Class+getFields(): Field+getMethods(): Method+getConstructors(): Constructor+getField(name: String): Field+getMethod(name: String, parameterTypes: Class): Method+getConstructor(parameterTypes: Class): Constructor+getDeclaredClasses(): Class+getDeclaredFields(): Field+getDeclaredMethods(): Method

public Object newInstance()
throws InstantiationException,
IllegalAccessException

Informations sur le type à l'exécution

- **instanceof**

```
if (g instanceof Graphics2D)
{

}
```

- **isInstance**

```
if ((Graphics2D.class).isInstance(g))
{

}
```

- **getClass**

```
if (g.getClass() == Graphics2D.class)
{

}
```

Simuler l'opérateur instanceof

```
class A {}

public class instance1 {

    public static void main(String args[]) {
        try {
            Class<?> cls = Class.forName("A");
            boolean b1 = cls.isInstance(new Integer(37));
            System.out.println(b1);
            boolean b2 = cls.isInstance(new A());
            System.out.println(b2);
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

NFP121

Class getSuperclass()

— retourne l'objet class
représentant la super classe (class,
interface, primitive type or void)

```
TextField t = new TextField();  
Class c = t.getClass(); // TextField  
Class s = c.getSuperclass(); //TextComponent
```

Récupérer les interfaces implémentées

Class

```
+toString(): String  
+forName(className: String): Class  
+newInstance(): Object  
+isInstance(obj: Object): boolean  
+isInterface(): boolean  
+isArray(): boolean  
+isPrimitive(): boolean  
+getName(): String  
+getClassLoader(): ClassLoader  
+getSuperclass(): Class  
+getPackage(): Package  
+getInterfaces(): Class  
+getComponentType(): Class  
+getModifiers(): int  
+getSigners(): Object  
+getDeclaringClass(): Class  
+getClasses(): Class  
+getFields(): Field  
+getMethods(): Method  
+getConstructors(): Constructor  
+getField(name: String): Field  
+getMethod(name: String, parameterTypes: Class): Method  
+getConstructor(parameterTypes: Class): Constructor  
+getDeclaredClasses(): Class  
+getDeclaredFields(): Field  
+getDeclaredMethods(): Method
```

Class[] getInterfaces()

Quelles sont les interfaces
implémentées par cette classe

`s.getClass().getInterfaces()[0]`

Afficher les interfaces implémentées par un objet

```
static void printInterfaceNames(Object o) {  
    Class<?> c = o.getClass();  
    for (Class<?> i : c.getInterfaces() ) {  
        String interfaceName = i.getName();  
        System.out.println(interfaceName);  
    }  
}
```

pourquoi introduire plus de capacités introspectives ?

- **Une industrie du composant logiciel**
- **Des ateliers d'intégrations de composants**
- **Des outils de 'customisation' d'applications**
- **Une application créée à partir d'un fichier texte**
 - Une variabilité du logiciel induite

Qu'est-ce que la réflexivité ?

C'est ce qui permet à un objet d'obtenir des informations sur sa propre structure et sur le traitement qu'il subit.

C'est un outil puissant permettant d'assembler du code à l'exécution sans que soit nécessaire la disposition du code source.

Démonstration ... Assemblage à l'exécution

- **JavaBean**

- Un JavaBean est un composant logiciel,

- une classe Java respectant certaines règles simples d'écriture du source :

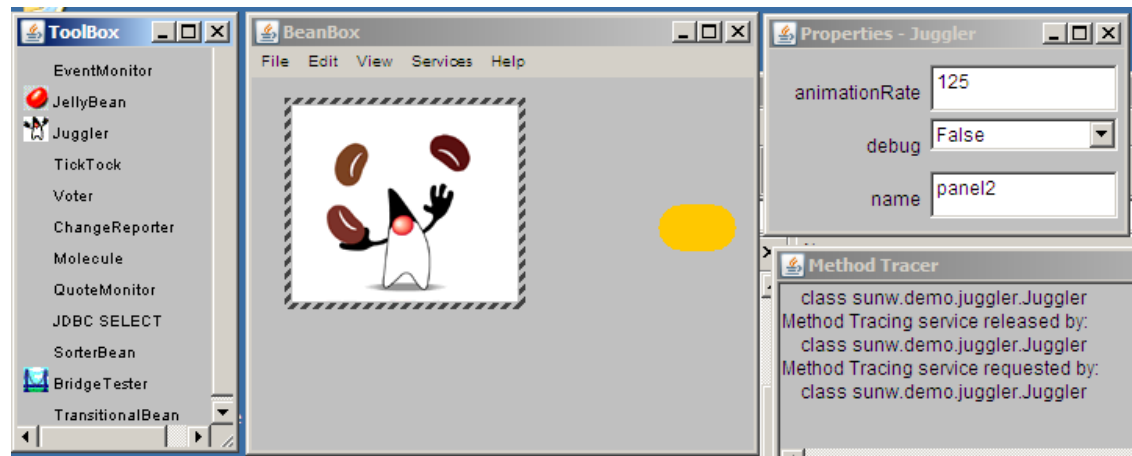
- setter, getter, properties, listener...

- Le BDK 1.1 est un « vieil » outil, du siècle dernier...

- <http://jfod.cnam.fr/NFP121/beans/>

- Dézipper le BDK1.1.zip puis D:\beans\beanbox\run.bat

- » (les boutons ne fonctionnent plus le j2SE 1.6 ... mais en 1.3)



- EJB, JavaEE, comme son nom l'indique c.f. GLG203-GLG204

Démonstration

- Deux composants logiciels en cours d'exécution

- (Deux objets introspectés par un outil : le bdk)

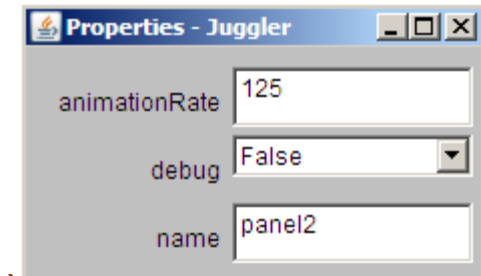
- Un jongleur

- Lecture de l'« état » du jongleur les getter

- `getAnimationRate()`, `getDebug()`, `getName()`

- Changement d'état, les setters

- `setAnimationRate(...)`, `setDebug(...)`, `setName(...)`

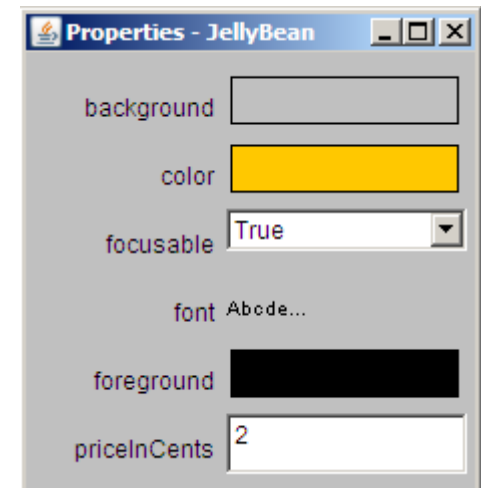


- Fenêtre Properties - Juggler

- Un « bouton »

- idem

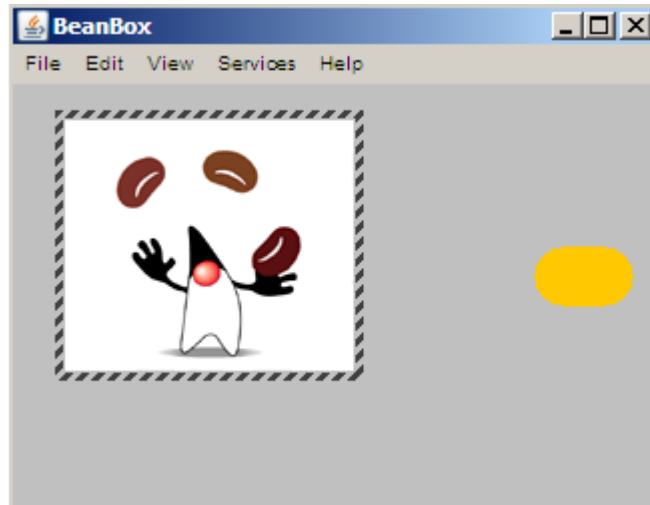
- Fenêtre Properties-JellyBean



Démonstration

- **Ajout d'une fonctionnalité**

- A chaque fois que la souris passe sur le bouton orange, le jongleur arrête de jongler ...



- Deux notifications « capturées »

- MouseEntered par le bouton puis appel de stopJuggling

- MouseExited par le bouton puis startJuggling

- Alors que ces deux composants sont des objets internes

- Notion de composant ici...

- discussion

Un bean (cf. NFP121 4-2_swing)

- **Un Bean est avant tout une classe ...**
 - *Un bean est un (extends) POJO, une classe quelconque*

Avec

- **Le respect de certaines conventions**
 - implements **Serializable**
 - Un constructeur par défaut
 - Un *getter* et/ou un *setter* pour chaque variable d'instance
 - `firePropertyChange` au sein du setter
- Simple n'est-ce pas ?
- **La suite**
 - Nombre devient Bean
 - La BeanBox, l'outil d'assemblage de Beans : un outil historique
 - Introspection systématique

La classe Nombre devient Bean : NombreBean (cf. NFP121 4-2_swing)

```
public class NombreBean implements Serializable{

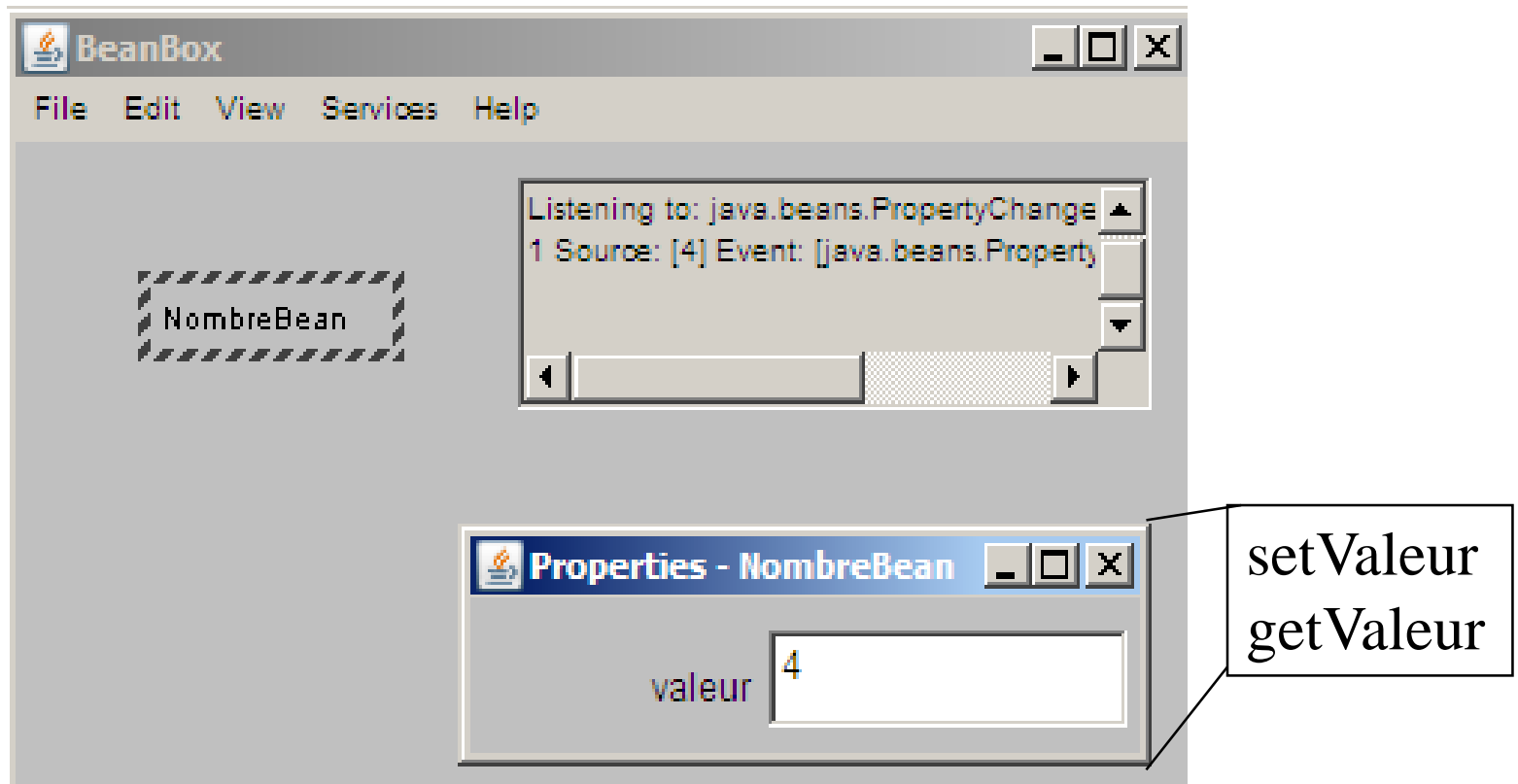
    public final int VALEUR_MIN;
    public final int VALEUR_MAX;
    private int valeur;
    private PropertyChangeSupport propertySupport;

    public NombreBean(){
        this.VALEUR_MIN = this.valeur = 0;
        this.VALEUR_MAX = 10;
        this.propertySupport = new PropertyChangeSupport(this);
    }

    public void inc(){
        if(valeur < VALEUR_MAX){
            int old = valeur;
            this.valeur++;
            propertySupport.firePropertyChange("valeur",old,valeur);
        }
    }

    public void addPropertyChangeListener(PropertyChangeListener l)
    {
        propertySupport.addPropertyChangeListener(l);
    } // public int getValeur() et setValeur()
```

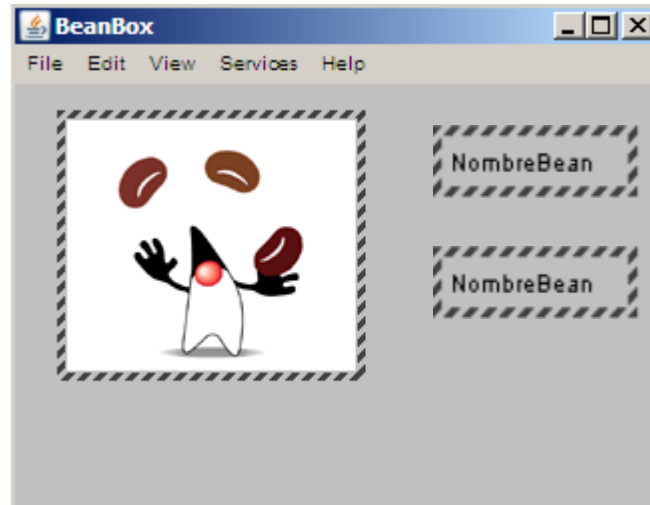

NombreBean intègre la BeanBox



- Ajout par l'outil BeanBox d'un listener (EventMonitor) à chaque changement de valeur de *valeur* ...

Démonstration

- Deux instances de NombreBean
- Un jongleur



- La première instance de NombreBean
 - à l'occurrence d'un changement de valeur demande au jongleur d'arrêter de jongler
- La seconde instance de NombreBean
 - à l'occurrence d'un changement de valeur demande au jongleur de jongler...

Le squelette d'un source de Bean, généré par netBeans... explication...

```
public class SimpleBean extends JLabel implements Serializable {
    public SimpleBean() {
        setText( "Hello world!" );
        propertySupport = new PropertyChangeSupport(this);
    }

    public static final String PROP_SAMPLE_PROPERTY = "sampleProperty";

    private String sampleProperty;
    private PropertyChangeSupport propertySupport;

    // recherche de cette méthode par un outil, par introspection
    public String getSampleProperty() { return sampleProperty;}

    // recherche de cette méthode par un outil, par introspection
    public void setSampleProperty(String value) {
        String oldValue = sampleProperty;
        sampleProperty = value;
        propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY,  

                                           oldValue, sampleProperty);
    }

    // recherche de cette méthode par un outil, par introspection
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }
}
```

- **Classes**
 - **Method**
 - **Constructor**
 - **Field**

- **java.lang: Class and Object**
- **java.lang.reflect: Constructor, Field, Method, Array**
 - **Constructeurs, Attributs, et Methodes**
 - **Array contient un ensemble de méthodes statiques pour la création**

Les méthodes d'une classe

Class
<div><div>+toString(): String</div><div><u>+forName(className: String): Class</u></div><div>+newInstance(): Object</div><div>+isInstance(obj: Object): boolean</div><div>+isInterface(): boolean</div><div>+isArray(): boolean</div><div>+isPrimitive(): boolean</div><div>+getName(): String</div><div>+getClassLoader(): ClassLoader</div><div>+getSuperclass(): Class</div><div>+getPackage(): Package</div><div>+getInterfaces(): Class</div><div>+getComponentType(): Class</div><div>+getModifiers(): int</div><div>+getSigners(): Object</div><div>+getDeclaringClass(): Class</div><div>+getClasses(): Class</div><div>+getFields(): Field</div><div>+getMethods(): Method</div><div>+getConstructors(): Constructor</div><div>+getField(name: String): Field</div><div>+getMethod(name: String, parameterTypes: Class): Method</div><div>+getConstructor(parameterTypes: Class): Constructor</div><div>+getDeclaredClasses(): Class</div><div>+getDeclaredFields(): Field</div><div>+getDeclaredMethods(): Method</div></div>

Method[] getMethods()

Toutes les méthodes
publiques locales et héritées.

Method[] getDeclaredMethods()

Toutes les méthodes locales.

java.lang.reflect.Method

Method
+getDeclaringClass(): Class +getName(): String +getModifiers(): int +getReturnType(): Class +getParameterTypes(): Class +getExceptionTypes(): Class +equals(obj: Object): boolean +hashCode(): int +toString(): String +invoke(obj: Object, args: Object): Object

Appeler une méthode

Method

- +getDeclaringClass(): Class
- +getName(): String
- +getModifiers(): int
- +getReturnType(): Class
- +getParameterTypes(): Class
- +getExceptionTypes(): Class
- +equals(obj: Object): boolean
- +hashCode(): int
- +toString(): String
- +invoke(obj: Object, args: Object): Object

Object invoke(Object obj, Object... args)
appelle cette méthode sur obj avec les paramètres args

Récupérer les paramètres

Method

+getDeclaringClass(): Class
+getName(): String
+getModifiers(): int
+getReturnType(): Class
+getParameterTypes(): Class
+getExceptionTypes(): Class
+equals(obj: Object): boolean
+hashCode(): int
+toString(): String
+invoke(obj: Object, args: Object): Object

Class[] getParameterTypes()

retourne un tableau
d'objets de classe Class
représentant le type des
paramètres formels de cette
méthode

Example

```
static void showMethods(Object o) {  
    Class<?> c = o.getClass();  
    for (Method m : c.getMethods()){  
        String methodString = m.getName();  
        System.out.println("Name: " + methodString);  
        String returnString = m.getReturnType().getName();  
        System.out.println(" Return Type: " + returnString);  
        System.out.print(" Parameter Types:");  
        for (Class<?> p : m . getParameterTypes() ) {  
            String parameterString = p.getName();  
            System.out.print(" " + parameterString);  
        } System.out.println();  
    }  
}
```

Incrementer une propriété d'un JavaBean

```
public int incrementProperty(String name, Object obj) {  
    String prop = Character.toUpperCase(name.charAt(0)) + name.substring(1);  
    String mname = "get" + prop; // un « getter »  
    Class[] types = new Class[] {};  
    Method method = obj.getClass().getMethod(mname, types);  
    Object result = method.invoke(obj, new Object[0]);  
    int value = ((Integer)result).intValue() + 1;  
    mname = "set" + prop; // un « setter »  
    types = new Class[] { int.class };  
    method = obj.getClass().getMethod(mname, types);  
    method.invoke(obj, new Object[] { new Integer(value) });  
    return value;  
}
```

Méthode main par introspection ...

```
public class Exemple1{  
    public static void main(String[] args){  
  
        Class<?> classe = Class.forName(args[0]);
```

// recherche de la méthode main

```
Method m = classe.getMethod("main",new Class[]{String[].class});
```

// recopie des paramètres [args[1]..args[args.length]]

```
String[] paramètres = new String[args.length-1];  
System.arraycopy(args,1,paramètres,0,args.length-1);
```

// exécution de la méthode main

```
m.invoke(null, new Object[]{paramètres});
```

```
}
```

```
usage java Exemple1 UneClasse param1 param2 param3
```

UneClasse n'est connue qu'à l'exécution

Obtenir les constructeurs

Class
<ul style="list-style-type: none">+toString(): String<u>+forName(className: String): Class</u>+newInstance(): Object+isInstance(obj: Object): boolean+isInterface(): boolean+isArray(): boolean+isPrimitive(): boolean+getName(): String+getClassLoader(): ClassLoader+getSuperclass(): Class+getPackage(): Package+getInterfaces(): Class+getComponentType(): Class+getModifiers(): int+getSigners(): Object+getDeclaringClass(): Class+getClasses(): Class+getFields(): Field+getMethods(): Method+getConstructors(): Constructor+getField(name: String): Field+getMethod(name: String, parameterTypes: Class): Method+getConstructor(parameterTypes: Class): Constructor+getDeclaredClasses(): Class+getDeclaredFields(): Field+getDeclaredMethods(): Method

Retourne le
ou les constructeurs l'objet .class
correspondant

Créer un nouvel objet

Constructor

- +getDeclaringClass(): Class
- +getName(): String
- +getModifiers(): int
- +getParameterTypes(): Class
- +getExceptionTypes(): Class
- +equals(obj: Object): boolean
- +hashCode(): int
- +toString(): String
- +newInstance(initargs: Object): Object

Object **newInstance**(Object... initargs)
crée un objet et l'initialise avec les arguments

Listing 1.

```
public class TwoString {  
    private String m_s1, m_s2;  
    public TwoString(String s1, String s2) {  
        m_s1 = s1; m_s2 = s2;  
    }  
}
```

Le code du Listing 2 récupère le constructeur et l'utilise pour créer une instance de la classe TwoString utilisant les Strings "a" et "b":

Listing 2. Appel du constructeur par réflexion

```
Class<?>[] types = new Class<?>[] { String.class, String.class };  
Constructor cons = TwoString.class.getConstructor(types);  
Object[] args = new Object[] { "a", "b" };  
TwoString ts = cons.newInstance(args);  
//TwoString ts = cons.newInstance("a", "b" );
```

Example

```
/** * Affiche les paramètres des différents constructeurs publiques d'une classe */
static void showConstructors(Object o) {
    Class c = o.getClass();
    for (Constructor cons : c.getConstructors()) {
        System.out.print("( ");
        for (Class p : cons.getParameterTypes()) {
            String parameterString = p.getName();
            System.out.print(parameterString + " ");
        } System.out.println(")");
    }
}
```


Obtenir les attributs d'une classe

Class
<div><div>+toString(): String</div><div><u>+forName(className: String): Class</u></div><div>+newInstance(): Object</div><div>+isInstance(obj: Object): boolean</div><div>+isInterface(): boolean</div><div>+isArray(): boolean</div><div>+isPrimitive(): boolean</div><div>+getName(): String</div><div>+getClassLoader(): ClassLoader</div><div>+getSuperclass(): Class</div><div>+getPackage(): Package</div><div>+getInterfaces(): Class</div><div>+getComponentType(): Class</div><div>+getModifiers(): int</div><div>+getSigners(): Object</div><div>+getDeclaringClass(): Class</div><div>+getClasses(): Class</div><div>+getFields(): Field</div><div>+getMethods(): Method</div><div>+getConstructors(): Constructor</div><div>+getField(name: String): Field</div><div>+getMethod(name: String, parameterTypes: Class): Method</div><div>+getConstructor(parameterTypes: Class): Constructor</div><div>+getDeclaredClasses(): Class</div><div>+getDeclaredFields(): Field</div><div>+getDeclaredMethods(): Method</div></div>

Field[] getFields()
tous les champs publics accessibles

Field[] getDeclaredFields()
tous les champs de cette classe

Field

```
+getDeclaringClass(): Class  
+getName(): String  
+getModifiers(): int  
+getType(): Class  
+equals(obj: Object): boolean  
+hashCode(): int  
+toString(): String  
+get(obj: Object): Object  
+getBoolean(obj: Object): boolean  
+getByte(obj: Object): byte  
+getChar(obj: Object): char  
+getShort(obj: Object): short  
+getInt(obj: Object): int  
+getLong(obj: Object): long  
+getFloat(obj: Object): float  
+getDouble(obj: Object): double  
+set(obj: Object, value: Object)  
+setBoolean(obj: Object, z: boolean)  
+setByte(obj: Object, b: byte)  
+setChar(obj: Object, c: char)  
+setShort(obj: Object, s: short)  
+setInt(obj: Object, i: int)  
+setLong(obj: Object, l: long)  
+setFloat(obj: Object, f: float)  
+setDouble(obj: Object, d: double)
```

Récupérer la valeur d'un champ de obj

Affecter la valeur d'un champ de obj

Incrémenter un attribut

```
public int incrementField(String name, Object obj) throws... {  
    Field field = obj.getClass().getDeclaredField(name);  
    int value = field.getInt(obj) + 1;  
    field.setInt(obj, value);  
    return value;  
}
```

Si ce champ est privé c.f. `field.setAccessible(true)` !

Les tableaux

Class

```
+toString(): String
+forName(className: String): Class
+newInstance(): Object
+isInstance(obj: Object): boolean
+isArray(): boolean
+isPrimitive(): boolean
+getName(): String
+getClassLoader(): ClassLoader
+getSuperclass(): Class
+getPackage(): Package
+getInterfaces(): Class
+getComponentType(): Class
+getModifiers(): int
+getSigners(): Object
+getDeclaringClass(): Class
+getClasses(): Class
+getFields(): Field
+getMethods(): Method
+getConstructors(): Constructor
+getField(name: String): Field
+getMethod(name: String, parameterTypes: Class): Method
+getConstructor(parameterTypes: Class): Constructor
+getDeclaredClasses(): Class
+getDeclaredFields(): Field
+getDeclaredMethods(): Method
```

boolean isArray()



Array

Array

+newInstance(componentType: Class, length: int): Object
+newInstance(componentType: Class, dimensions: int): Object
+getLength(array: Object): int
+get(array: Object, index: int): Object
+getBoolean(array: Object, index: int): boolean
+getByte(array: Object, index: int): byte
+getChar(array: Object, index: int): char
+getShort(array: Object, index: int): short
+getInt(array: Object, index: int): int
+getLong(array: Object, index: int): long
+getFloat(array: Object, index: int): float
+getDouble(array: Object, index: int): double
+set(array: Object, index: int, value: Object)
+setBoolean(array: Object, index: int, z: boolean)
+setByte(array: Object, index: int, b: byte)
+setChar(array: Object, index: int, c: char)
+setShort(array: Object, index: int, s: short)
+setInt(array: Object, index: int, i: int)
+setLong(array: Object, index: int, l: long)
+setFloat(array: Object, index: int, f: float)
+setDouble(array: Object, index: int, d: double)

Object get(Object array, int index)

retourne la valeur située à
index dans array

Agrandir un tableau

```
public Object growArray(Object array, int size) {  
    Class type = array.getClass().getComponentType();  
    Object grown = Array.newInstance(type, size);  
    System.arraycopy(array, 0, grown, 0,  
                     Math.min(Array.getLength(array), size));  
    return grown;  
}
```

Patron Visiteur : le retour

- **Visite en fonction du type de nœud**
 - **Adapté aux structures fermées :**
 - Structures évoluant peu ou pas
 - Par exemple un Arbre de Syntaxe abstraite de Java
 - <https://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html>

Comment prendre en compte toute modification ultérieure ?

- **Idée**
 - Recherche de la bonne méthode par introspection ?
 - <http://www.javaworld.com/article/2077602/learn-java/java-tip-98--reflect-on-the-visitor-design-pattern.html>

Le patron Visiteur : Avant

- **Au sein de chaque Noeud**

```
public <T> T accepter(Visiteur<T> v){  
    return v.visite(this);  
}
```

Une classe Visiteur et toutes les visites des nœuds

```
public class VisiteurParDefaut<T> extends VisiteurExpression<T>{  
  
    public T visite(Constante c){return null;}  
    public T visite(Variable v){return null;}  
    public T visite(FonctionJava f){return null;}  
  
    public T visite(Division d){return null;}  
    public T visite(Addition a){return null;}  
    public T visite(Multiplication m){return null;}  
    public T visite(Soustraction s){return null;}  
}
```

- Si un nouveau type de nœud est requis alors
 - **Modification de tous les visiteurs**
 - Par l'ajout de la méthode visite de ce nouveau type de Noeud

Le patron Visiteur : Après

- **Une seule méthode accepter :**
 - **Au sein de la Racine du composite et uniquement**

```
2 |
3 | public abstract class Expression{
4 |
5 |     public <T> T accepter(VisiteurExpression<T> v) {
6 |         return v.visite(this);
7 |     }
8 |
9 | }
```

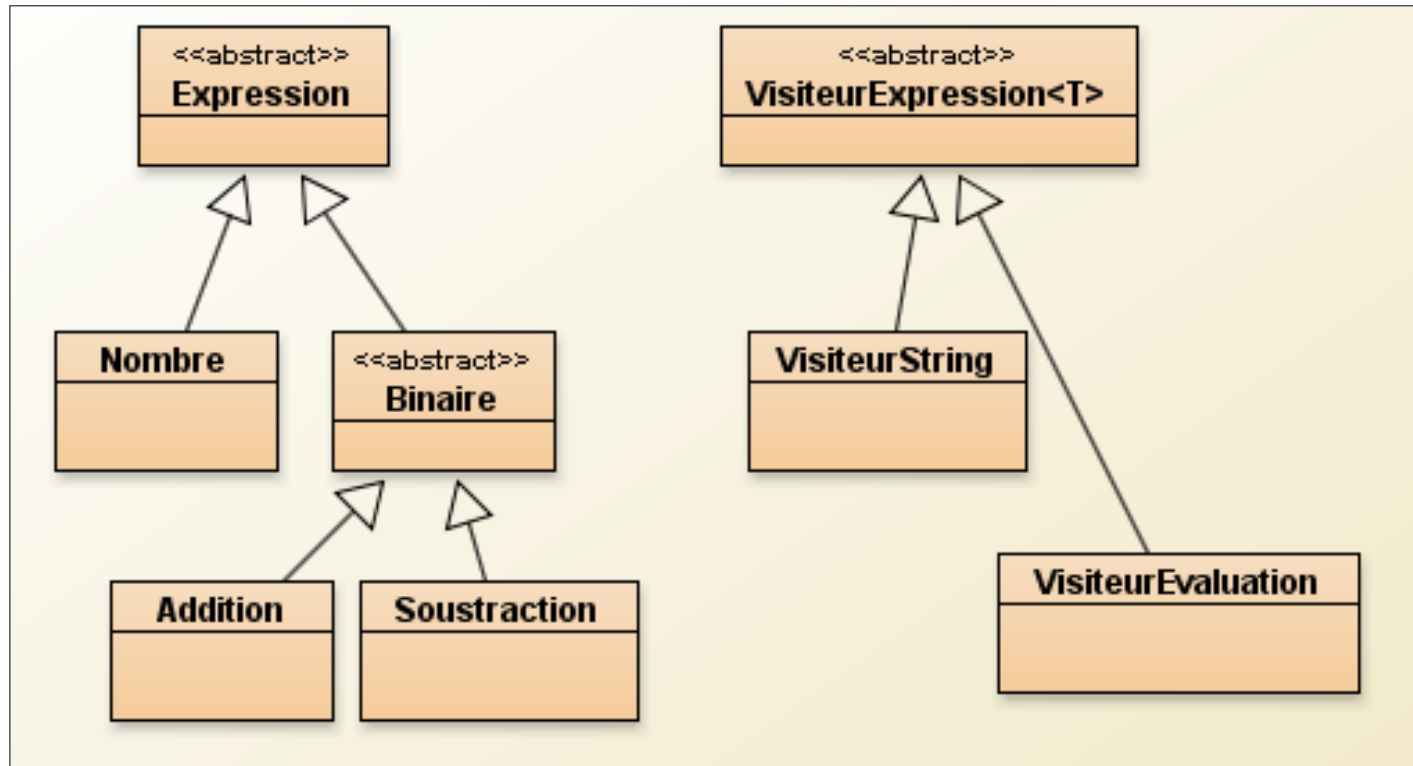
- **L'ajout de Nœud supplémentaire devient élémentaire...**
- **Une classe Visiteur munie d'une unique méthode**
 - Recherche par introspection de la méthode de visite spécifique du Nœud
 - au sein de la classe recherche de la méthode
visite(NouveauNoeud n)

La classe Visiteur + une petite visite

```
4
5 public abstract class VisiteurExpression<T>{
6
7     public T visite(Expression expr) {
8
9         Class<?> cl = this.getClass();
10        while(cl != Object.class){
11            try{
12                try{
13                    Method m = cl.getDeclaredMethod("visite",expr.getClass());
14                    return (T)m.invoke(this, expr);
15                }catch(Exception e){
16                    cl = cl.getSuperclass();
17                }
18            }catch(Exception e){
19            }
20        }
21        return null;
22    }
```

- Recherche et exécution de la méthode
 - visite(*Noeud visité*)

Les protagonistes



- La classe **VisiteurString** contient toutes les méthodes `visite(...)` de toutes les feuilles concrètes

Rendre visite

```
public void testAddition() {  
    Expression e = new Addition(new Nombre(3), new Nombre(2));  
    VisiteurExpression<String> ve = new VisiteurString();  
    String str = e.accepter(ve);  
    assertEquals("(3 + 2)", str);  
}
```

```
public void testSoustraction() {  
    Expression e = new Soustraction(new Nombre(3), new Nombre(2));  
    VisiteurExpression<String> ve = new VisiteurString();  
    String str = e.accepter(ve);  
    assertEquals("(3 - 2)", str);  
}
```

- **Même code que d'habitude ...**

Un nouveau Nœud ?

```
private static class Division extends Binaire{  
    public Division(Expression op1, Expression op2){  
        super(op1, op2);  
    }  
}
```

```
private static class VisiteurString2 extends VisiteurString{  
    public String visite(Division s){  
        return "(" + s.getOp1().accepter(this) + " / " + s.getOp2().accepter(this) + ")";  
    }  
}
```

```
public void testDivision(){  
    Expression e = new Division(new Nombre(3), new Nombre(2));  
    VisiteurExpression<String> ve = new VisiteurString2();  
    String str = e.accepter(ve);  
    assertEquals("(3 / 2)", str);  
}
```

- **Classe Division**, un simple appel de super
- **La visite** est une sous-classe du visiteur existant

Performance ... *100

```
public void testPerformance(){
    Expression expr = new Addition(// 3 + 2 - 5
                                   new Nombre(3),
                                   new Soustraction(
                                       new Nombre(2),
                                       new Nombre(5)));
    VisiteurExpression<Integer> vc = new VisiteurEvaluation();
    long top = System.currentTimeMillis();
    for(long i = 0; i<100000000L; i++){
        Integer res = expr.accepter(vc);
    }
    System.out.println("durée : " + (System.currentTimeMillis()-top) + " ms");
}
```

- **Avant : 672 ms**
 - De simples appels de méthodes
- **Après : 63484 ms**
 - Par introspection

Patrons et configuration

- **Comment configurer la stratégie et le mandataire depuis un fichier de configuration**
 - **Ou comment définir une classe depuis un fichier texte**

Deux exemples

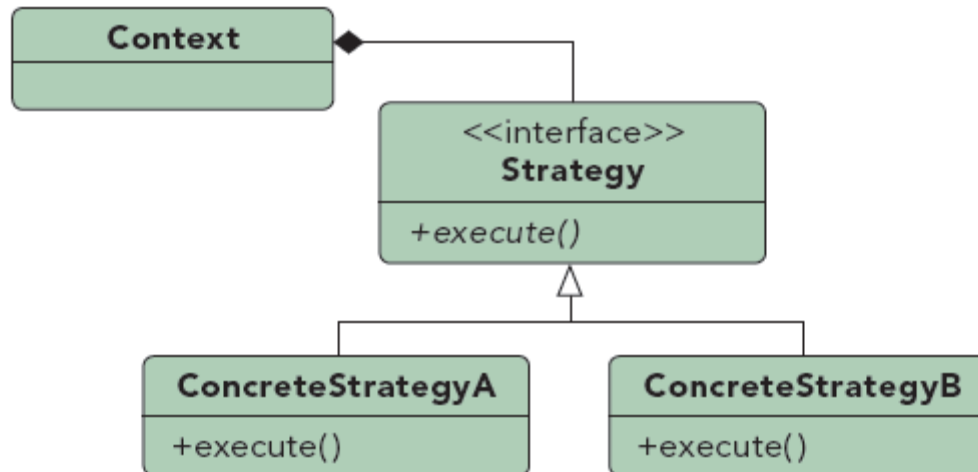
- **Patron Stratégie**
- **Patron Procuration**

Démonstration

- Le patron stratégie

STRATEGY

Object Behavioral

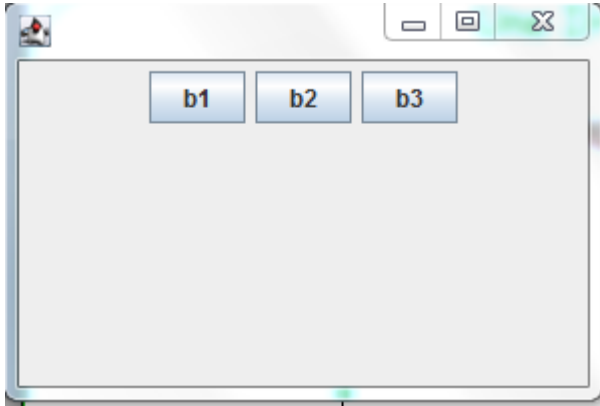


- La stratégie est définie dans un fichier de configuration

Une IHM, un JPanel

Stratégie de placement des objets graphiques

- `void setLayout(LayoutManager layout){ }`



`setLayout(new FlowLayout())`



`setLayout(new GridLayout())`

JPanel et le patron Stratégie

Placement des objets graphiques selon une stratégie
FlowLayout, GridLayout, GridBagLayout, CardLayout ...


```
public IHM(){  
    JPanel panel = new JPanel();  
    JButton b1 = new JButton("b1");  
    JButton b2 = new JButton("b2");  
    JButton b3 = new JButton("b3");  
  
    panel.setLayout(getLayoutManager());  
  
    panel.add(b1);panel.add(b2);panel.add(b3);  
    add(panel);  
    setBounds(100,100, 300, 200);  
    setVisible(true);setAlwaysOnTop(true);  
}  
  
private LayoutManager getLayoutManager(){  
    return new FlowLayout();  
}
```

getLayoutManager

```
private LayoutManager getLayoutManager(){
    LayoutManager layout = null;
    try{

        Properties props= new Properties();
        props.load(new FileInputStream(new File("README.TXT")));
        String className = props.getProperty("layout");
        Class<?> clazz = Class.forName(className);
        layout = (LayoutManager)clazz.newInstance();

    }catch(Exception e){
        layout = new FlowLayout();
        e.printStackTrace();
    }
    return layout;
}
```

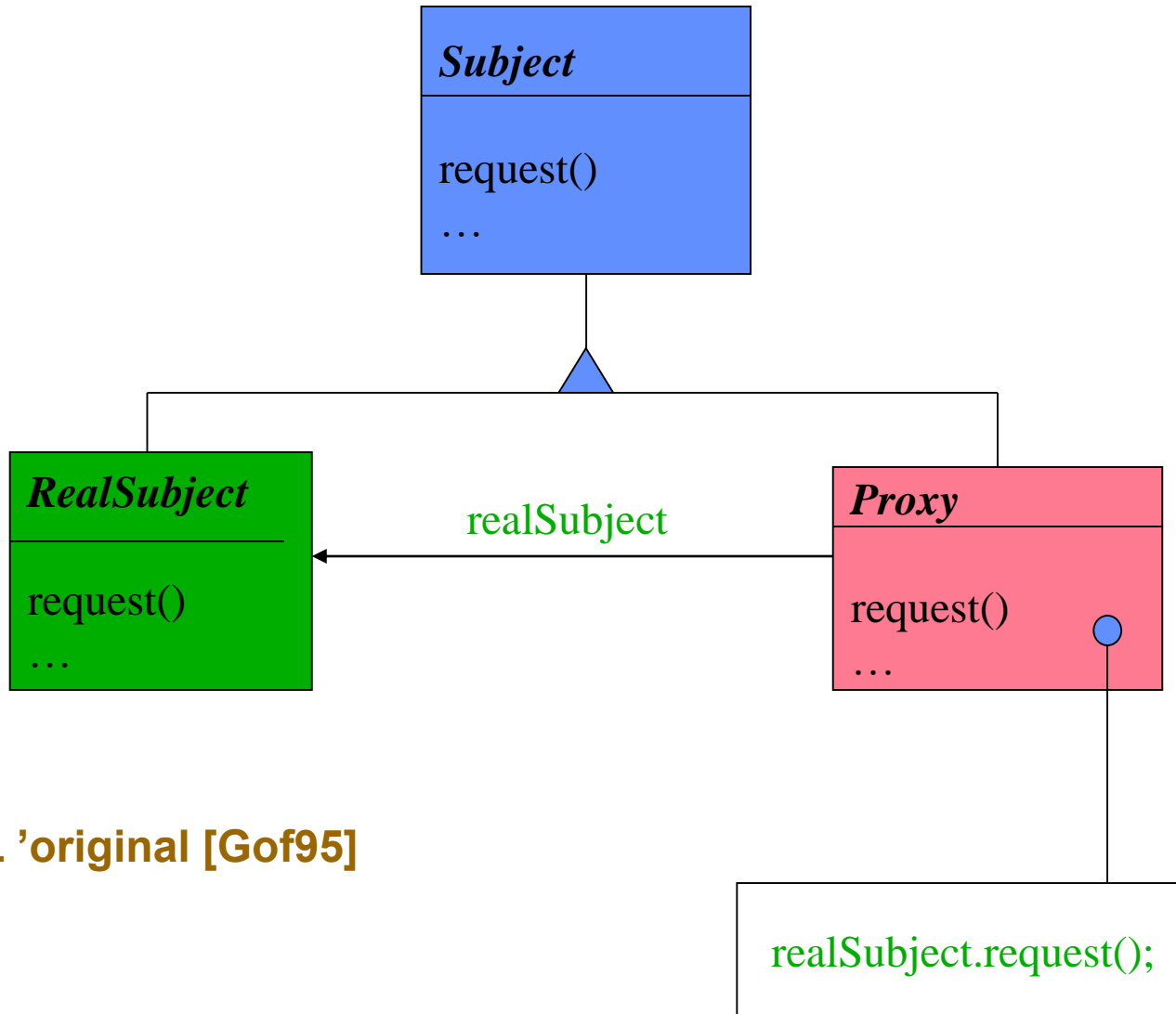


```
#
#layout=java.awt.FlowLayout

layout=java.awt.GridLayout

#layout=java.awt.GridBagLayout
#layout=java.awt.GridBagLayout
```

Proxy l'original : diagramme UML



– L 'original [Gof95]

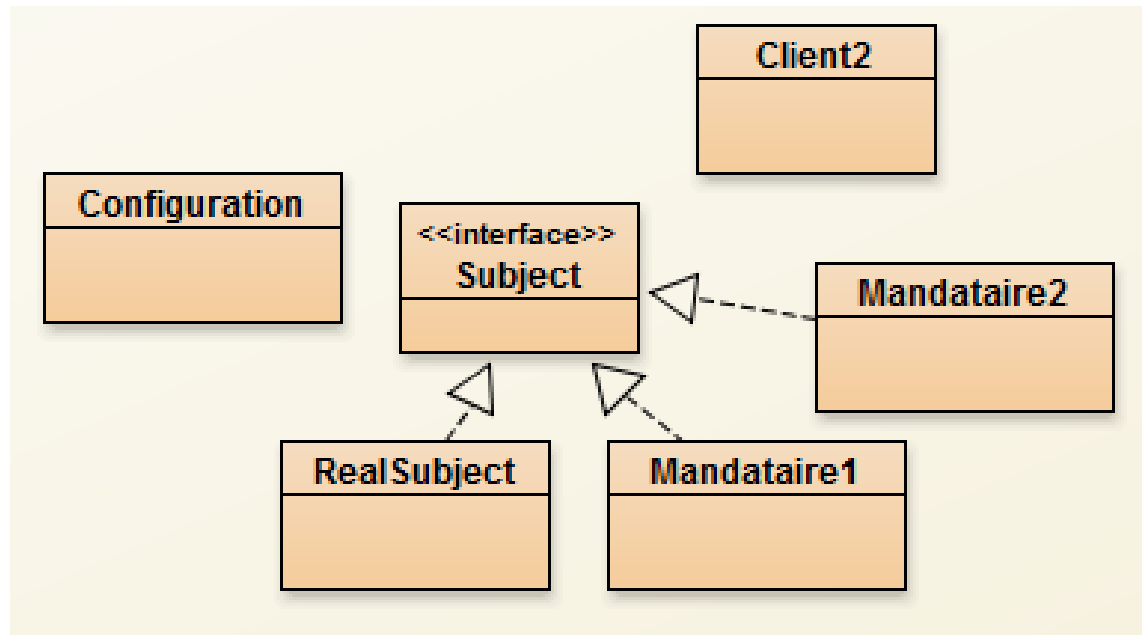
Patron Proxy : à la demande

- Nous souhaitons remplacer cette instruction Java
 - `Subject s = new Proxy();`
- Par celle-ci, en paramètre un nom de fichier texte
 - `Subject s = new ProxyInstance(nomDuFichier);`

```
// le fichier contient le nom du Mandataire  
mandataire=Proxy  
#mandataire=UneAutreClasseProxy
```

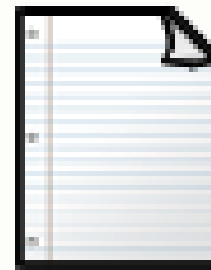
- Bien plus souple ?
 - Il suffit de modifier le fichier pour obtenir un nouveau mandataire
- Comment ?

Le projet Bluej



- Choix du mandataire: fichier texte

```
mandataire=Mandataire1
#ou mandataire=Mandataire2
# ... etc.
```



Configuration + le Client

class Configuration

```
public static Subject newProxyInstance(String nomDuFichier) throws Exception{
    Properties props = new Properties();
    props.load(new FileInputStream(new File(nomDuFichier)));
    String mandataireClassName = props.getProperty("mandataire");
    Class<?> classeMandataire = Class.forName(mandataireClassName);
    return (Subject)classeMandataire.newInstance();
}
```

```
public class Client2{
```

```
    public static void main(String[] args) throws Exception{
```

```
        String nomDuFichier = args[0]; // pour les tests ...
```

```
        Subject s = Configuration.newProxyInstance(nomDuFichier);
```

```
        String res = s.requete("http://jfod.cnam.fr");
```

```
        System.out.println(" res : " + res);
```

```
        res = s.requete("http://www.google.fr");
```

```
        System.out.println(" res : " + res);
```

```
    }
```

```
}
```

- **Démonstration...**

Suite sommaire

- **Parcours des instances de la classes Class**
 - **Démonstration éventuelle**
- **Conclusion**
- **Annexe Le patron Procuration revisité**

Parcours de l'arbre d'héritage

```
// parcours de l'arbre vers la racine java.lang.Object
Class<?> classe = Class.forName(args[0]);
while(classe != null){
    System.out.println(" classe : " + classe.getName());
    classe = classe.getSuperclass();
}

// classe .class
Class<String> s = String.class;
Class<Integer> i = Integer.class;
Class<?> c = int.class;

// Signature de méthodes, String s, int i
Class<?>[] sig1 = new Class<?>[]{String.class, int.class};

// Signature de méthodes, Object o
Class<?>[] sig2 = new Class<?>[]{Object.class};

// Quelles sont les méthodes publiques et héritées ?
classe = Class.forName(args[0]);
for( Method m : classe.getMethods()){
    System.out.println("getMethods : " + m);
}
```

Parcours toujours

// Quelles sont les méthodes accessibles en excluant les héritées?

```
classe = Class.forName(args[0]);
for( Method m : classe.getDeclaredMethods()) {
    System.out.println("getDeclaredMethods : " + m);
}
```

// Quelles sont les Méthodes publiques avec cette signature

// Class<?>[] sig2 = new Class<?>[]{Object.class};

// présentes dans l'arbre dont la feuille est args[0] ?

```
classe = Class.forName(args[0]);
while(classe != null){
    for( Method m : classe.getMethods()){
        try{
            classe.getMethod(m.getName(), sig2);
            System.out.println("classe : " + classe.getName() + " méthode " + m);
        }catch(Exception e){
        }
    }
    classe = classe.getSuperclass();
}
```

Parcours stop

```
// Quelles sont les classes dans lesquelles
// certaines méthodes de args[0] ont été redéfinies ?
// attention aux fonctions et à la covariance possible ...
Map<Method,List<Class<?>>> map = new HashMap<Method,List<Class<?>>>();
    classe = Class.forName(args[0]); // la feuille
    // pour toutes ses méthodes
    for( Method m : classe.getDeclaredMethods()){
        Class<?> cl= classe.getSuperclass();
        while(cl!=null){
            // pour toutes les méthodes de cette classe
            for( Method m1 : cl.getDeclaredMethods()){
                if(m.getName().equals(m1.getName())){
                    try{ // et la même signature
                        cl.getDeclaredMethod(m.getName(),m.getParameterTypes());
                        List<Class<?>> liste = map.get(m);
                        if( liste==null) liste = new ArrayList<Class<?>>();
                        liste.add(cl);
                        map.put(m,liste);
                    }catch(Exception e){
                    }
                }
            }
            cl = cl.getSuperclass();
        }
    }
    for(Method me : map.keySet()){
        System.out.println(" redéfinition de " + me + " dans les classes " + map.get(me));
    }
}
```

Exécution à la volée

```
public static void main(String[] args) throws Exception{

// exécution de la méthode main,
    Class<?> cl = Class.forName(args[0]);
    Class<?>[] signature = new Class<?>[]{String[].class};
    Method main = cl.getDeclaredMethod("main",signature);
    String[] args2 = new String[args.length-1];
    System.arraycopy(args,1,args2,0,args2.length);
    main.invoke(null,new Object[]{args2});

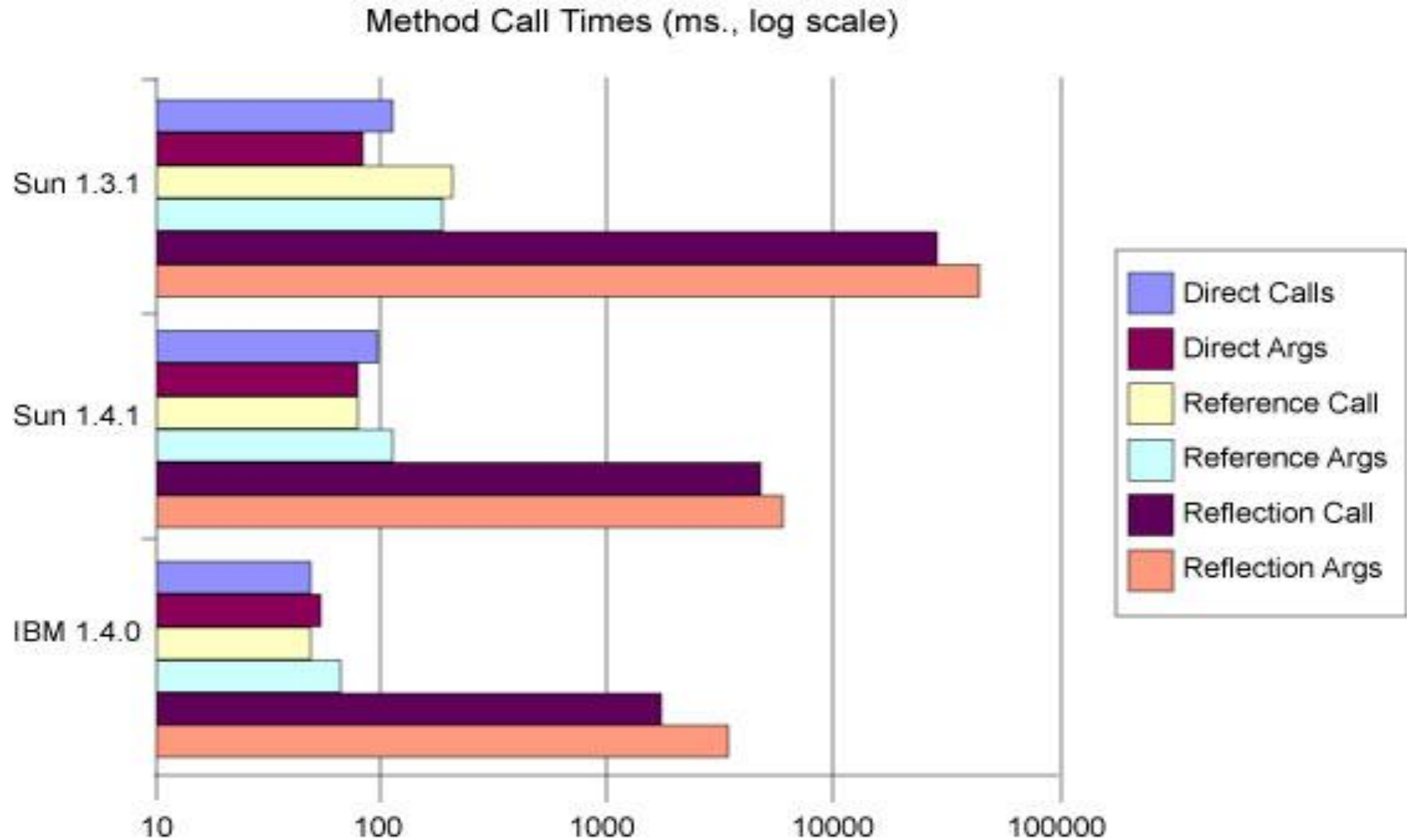
// exécution d'un constructeur
    Class<?> classe = Class.forName(args[1]);
    Class<?>[] typeDesArguments = new Class<?>[]{};
// à la recherche du constructeur sans paramètre
    Constructor<?> cons = classe.getConstructor(typeDesArguments);
// le constructeur
    Object[] paramètres = new Object[]{};    // sans paramètre
    Object c = cons.newInstance(paramètres); // exécution

// exécution d'une méthode
    classe = Class.forName(args[1]);
    Method m = classe.getMethod("m1",new Class<?>[]{int.class,Float.class});
    m.invoke(c,new Object[]{3,new Float(3.12)}); // exécution

}
```

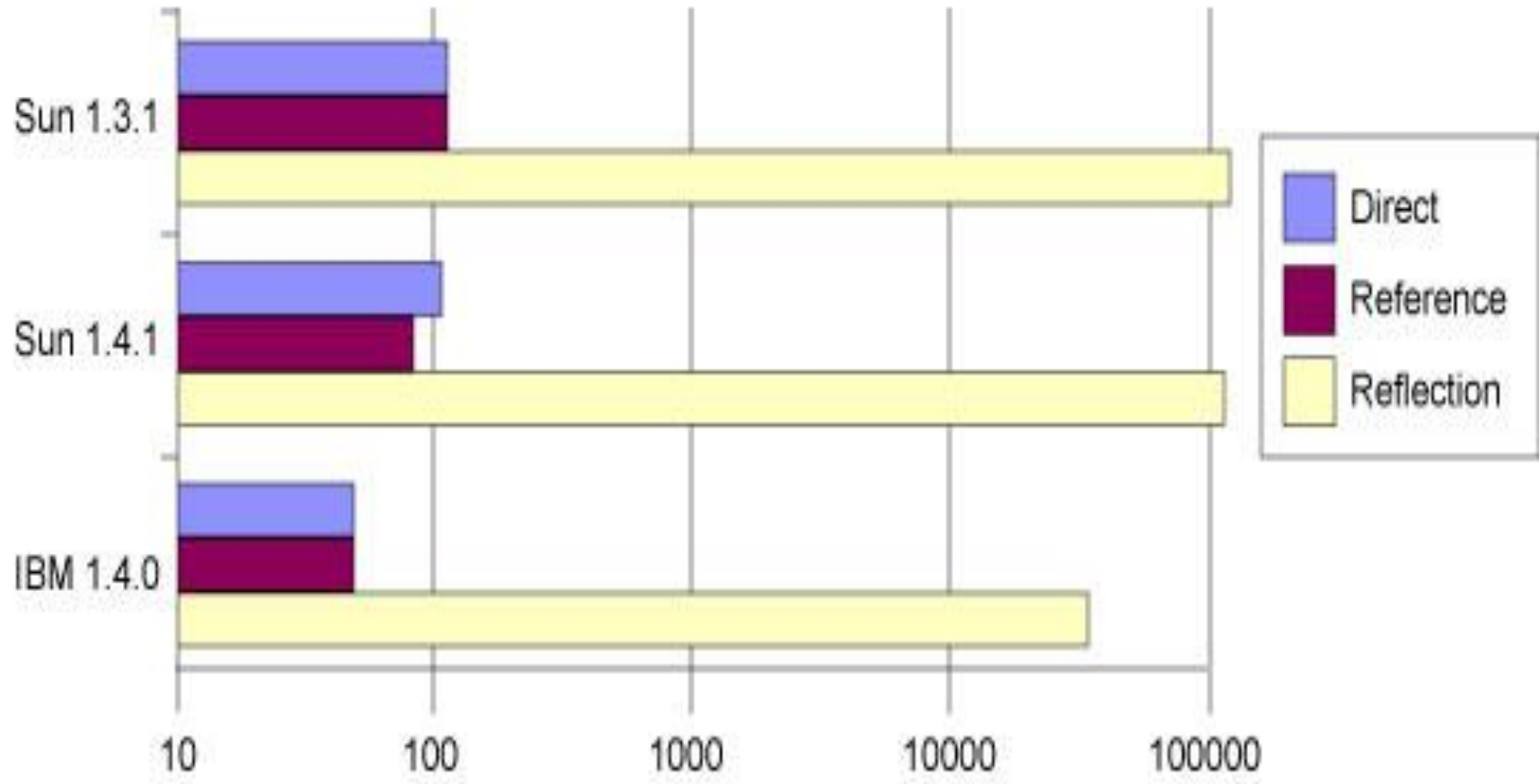
Performance

Beaucoup d'avantages mais pas efficace car interprété (au second niveau !)



Performance

Field Access Times (ms., log scale)



Conclusion

- **Industrie du composant logiciel**
 - Succès de javaEE
 - Enterprise JavaBeans, des composants tout prêts côté serveur
 - Conteneur d'EJB
 - Framework « Injection de dépendance »
 - Spring, etc...
- **Conteneur de Beans créé pour NFP121**
 - femtoContainer bientôt utilisé

Annexe Le patron Procuration, le retour

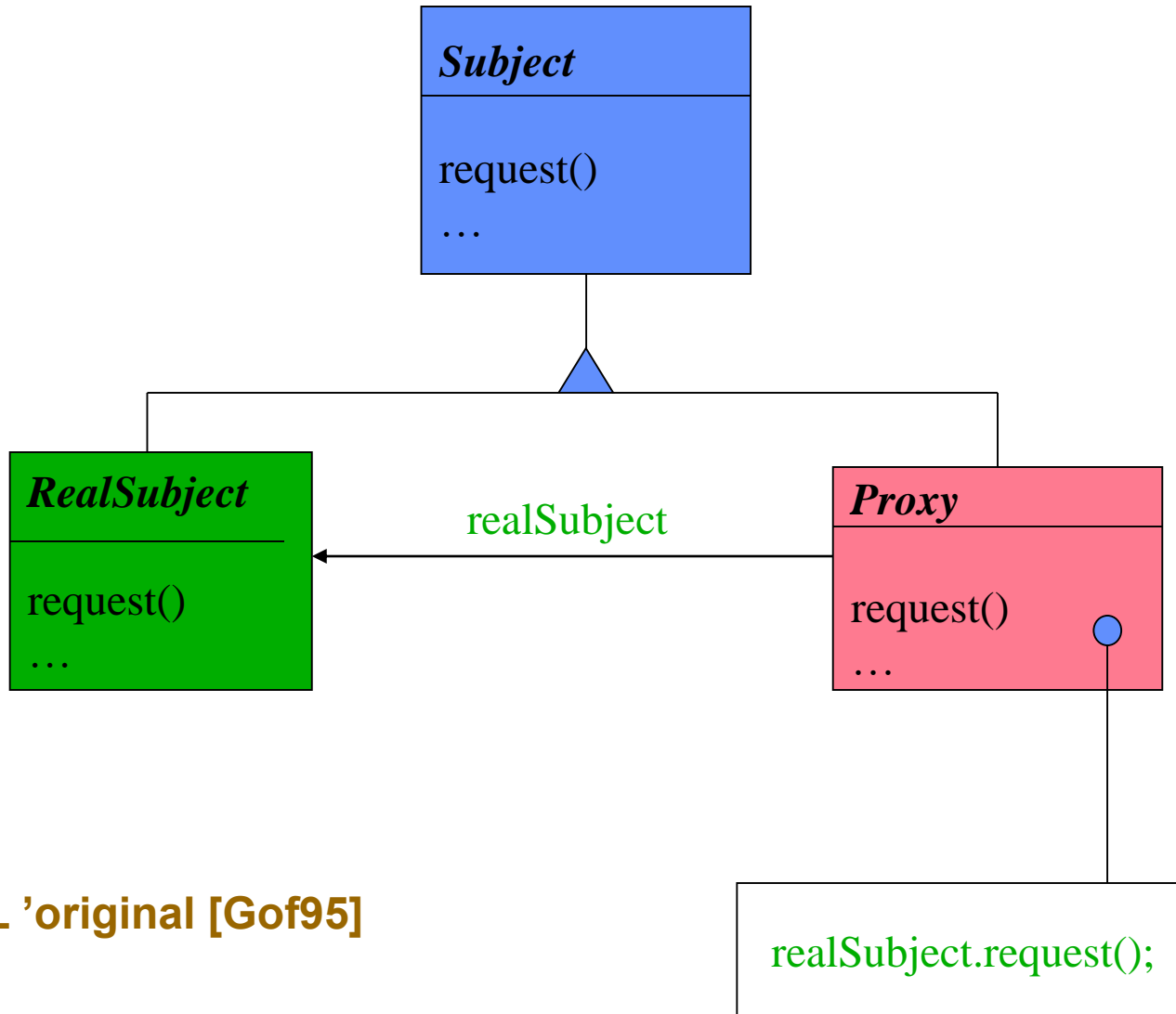
- **extrait de l'API du J2SE Dynamic Proxy**

- <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/InvocationHandler.html>
- <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html>

- **Un exemple**

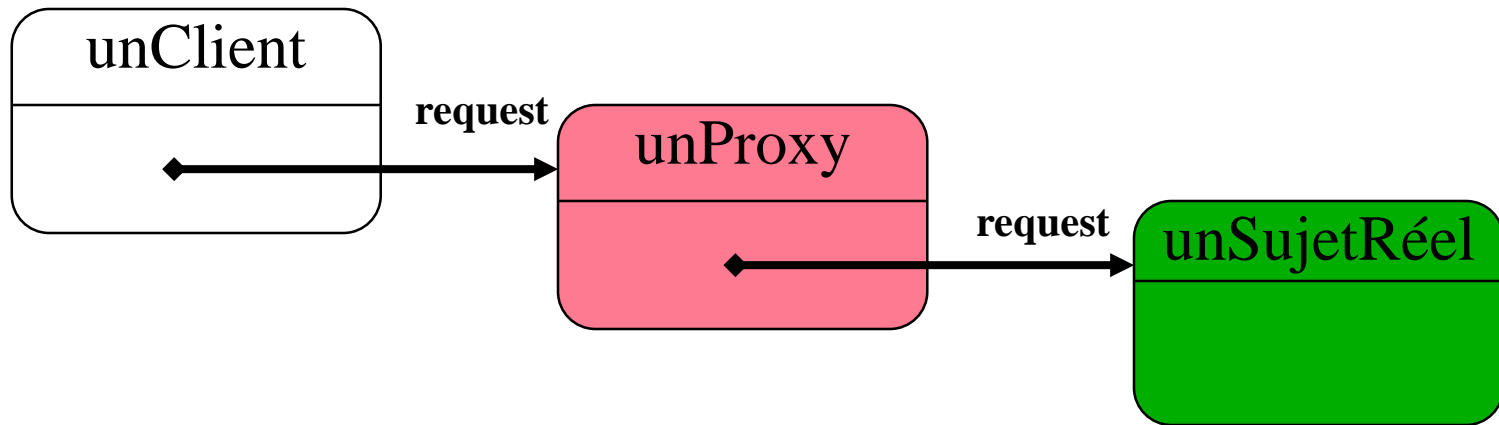
- `public static<T> List<T>unmodifiableList(List<? extends T> list)`

Proxy l'original : diagramme UML



– L 'original [Gof95]

Un exemple possible à l'exécution

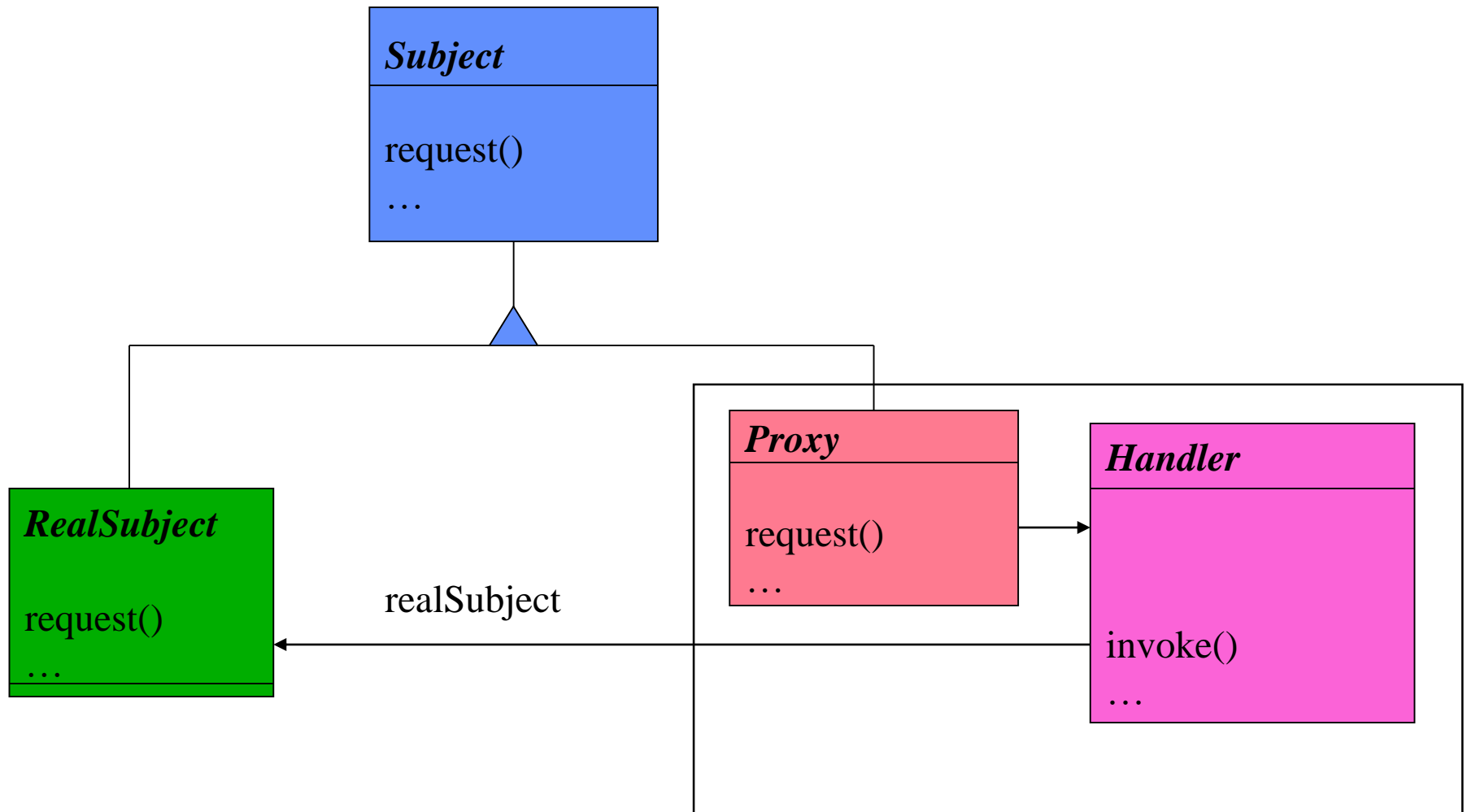


- Une séquence et des instances possibles
 - **unProxy.request()** → **unSujetRéel.request()**
 - **unProxy** est créé dynamiquement

DynamicProxy, c.f. le patron Procuration

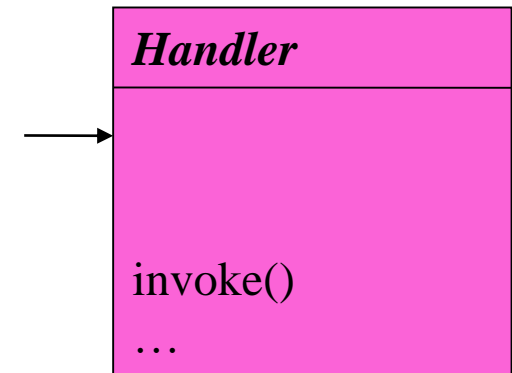
- **extrait de l'API du J2SE Dynamic Proxy**
 - <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/InvocationHandler.html>
 - <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html>
- **Génération de byte code à la volée**
 - Rare exemple en java
- **Paquetages concernés**
- **java.lang.reflect et java.lang.reflect.Proxy**

Le diagramme UML revisité



- Vite un exemple ...

Handler implements InvocationHandler



interface java.lang.reflect.InvocationHandler;

- ne contient qu'une seule méthode

Object invoke(Object proxy, Method m, Object[] args);

- proxy : le proxy généré
- m : la méthode choisie
- args : les arguments de cette méthode

Handler implements InvocationHandler

```
public class Handler implements InvocationHandler{

    private Subject service;

    public Handler(){
        this.service = new RealSubject();    // par exemple...
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Exception{

        return method.invoke(service, args); // par introspection
    }
}
```

Il nous manque le mandataire qui se charge de l'appel de invoke
ce mandataire est créé par une méthode ad'hoc toute prête
`newProxyInstance` issue de la classe `java.lang.reflect.Proxy`

Création dynamique du Proxy/Mandataire

```
public static Object newProxyInstance(ClassLoader loader,  
                                         Class[] interfaces,  
                                         InvocationHandler h) throws  
....
```

crée dynamiquement un mandataire

spécifié par le chargeur de classe *loader*

lequel implémente les interfaces *interfaces*,

*la méthode *h.invoke* sera appelée par l'instance du Proxy retournée*

retourne une instance du *proxy*

*Méthode de classe de la classe *java.lang.reflect.Proxy*;*

Exemple ...

// obtention du chargeur de classes

```
ClassLoader cl = Service.class.getClassLoader();
```

// l'interface implémentée par le futur mandataire

```
Class[] interfaces = new Class[]{Subject.class};
```

// le mandataire Handler

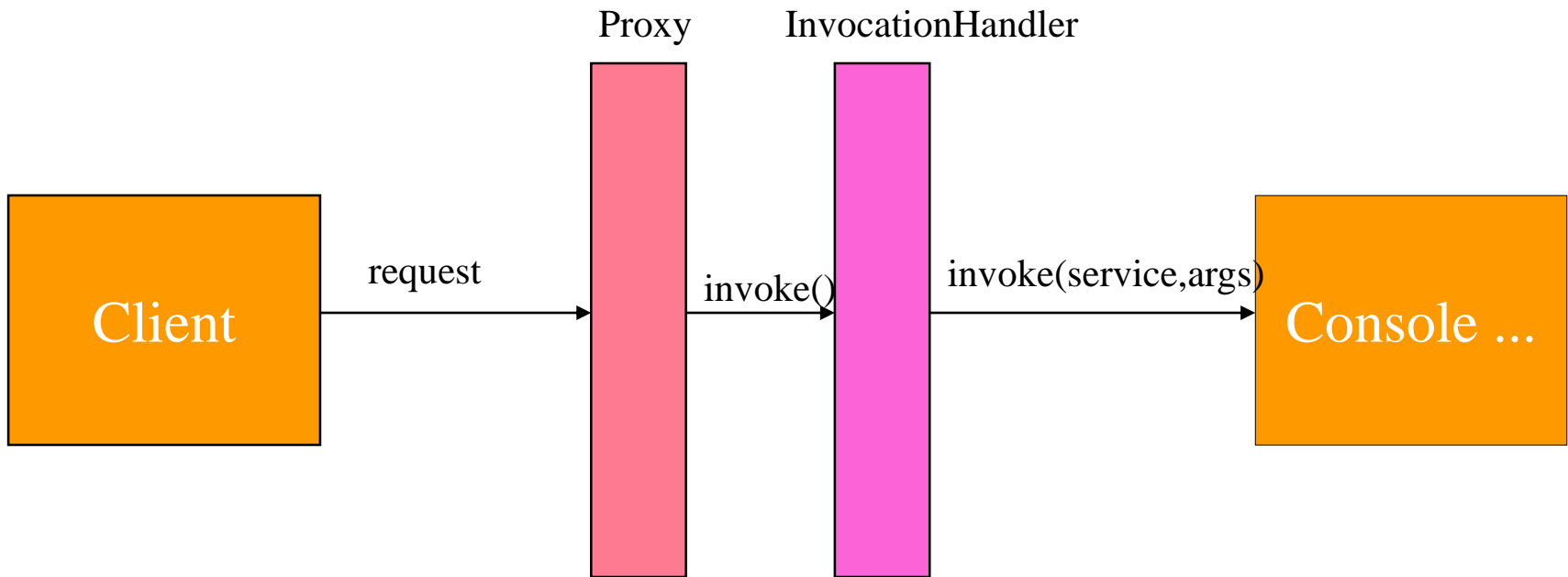
```
InvocationHandler h = new Handler();
```

// Creation du mandataire

```
Subject proxy = (Subject )
```

```
Proxy.newProxyInstance(cl, interfaces, h);
```


Un dessin



```
Subject proxy = (Subject) Proxy.newProxyInstance (cl, interfaces, h) ;  
  
proxy.request();
```

DynamicProxy deux exemples

- **1) Filtrage à la volée de certaines méthodes**
 - Cf. la méthode `unmodifiable` de la `java.util.Collections`
 - `public static<T> List<T>unmodifiableList(List<? extends T> list)`
 - Démonstration ?
- **2) Une pile instrumentée (le retour !)**
 - Vérification de contraintes à l'exécution si elles existent ...

1) Filtrage à la volée de certaines méthodes

- **Étant donné**

- L'interface et la classe `Donnée<T>` données ci dessous

```
public interface Donnée<T>{  
    public T lire();  
    public void écrire(T t);  
}
```

```
public class DonnéeImpl implements Donnée<Integer>{  
    private int x;  
    public Integer lire(){  
        return x;  
    }  
    public void écrire(Integer t){  
        x = t;  
    }  
}
```

- Nous souhaitons en cours d'exécution installer des mandataires chargés de filtrer les appels de méthodes
 - Par exemple, interdire l'appel d'écrire, puis lire, puis les deux .. puis

Filtrage à la volée, un usage

```
Donnée<Integer> d = new DonnéeImpl();  
d.écrire(3);
```

```
// ROM, read only memory  
d = Filtrage.nouveauFiltre(d,new String[]{"lire"});  
try{  
    d.écrire(3);  
}catch(Exception e){e.printStackTrace(); }
```

```
// WOM, write only memory ?  
d = Filtrage.nouveauFiltre(new DonnéeImpl(),new String[]{"écrire"});  
d.écrire(3);  
try{  
    d.lire();  
}catch(Exception e){e.printStackTrace();}
```

```
// ni lire ni écrire ?  
d = Filtrage.nouveauFiltre(d,new String[]{});  
try{  
    d.écrire(3);  
    d.lire();  
}catch(Exception e){e.printStackTrace();}
```

Classe Filtrage, méthode nouveauFiltre

```
import java.lang.reflect.Proxy;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Filtrage {

    public static <T> Donnée<T> nouveauFiltre(Donnée<T> donnée,
                                              String[] autorisées){
        return (Donnée<T>) Proxy.newProxyInstance(
            Donnée.class.getClassLoader(),
            new Class<?>[]{Donnée.class},
            new Filtre<T>(donnée, autorisées));
    }

    private static class Filtre<T> implements java.lang.reflect.InvocationHandler{
        // ..... Page suivante
    }
}
```

Classe Filtrage, classe interne et statique Filtre

```
private static class Filtre<T> implements java.lang.reflect.InvocationHandler{

    private Donnée<T> donnée;
    private String[] autorisées;

    private Filtre(Donnée<T> donnée,String[] autorisées){
        this.donnée = donnée;
        this.autorisées = autorisées;
    }

    public Object invoke(Object proxy, Method m, Object[] args)throws Throwable{

        try {
            for(String s : autorisées){
                if(s.equals(m.getName())){
                    return m.invoke(donnée, args);
                }
            }
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        } catch (Exception e) {
            throw new RuntimeException("unexpected invocation exception: " + e.getMessage());
        }
        throw new RuntimeException(m.getName() + " est une méthode inhibée ");
    }

}
```

Filtrage, le mandataire, proxy en une ligne

Essai syntaxique avec une classe anonyme

```
//      final String[] autorisées = new String[]{"lire"};
//      final Donnée<Integer> dl = new DonnéeImpl();
//      Donnée<Integer> proxy
//          = (Donnée<Integer>) Proxy.newProxyInstance(
//              Donnée.class.getClassLoader(),
//              new Class<?>[]{Donnée.class},
//              new java.lang.reflect.InvocationHandler(){
//                  public Object invoke(Object proxy, Method m, Object[] args) throws Throwable{
//                      try {
//                          for(String s : autorisées){
//                              if(s.equals(m.getName())){
//                                  return m.invoke(dl, args);
//                              }
//                          }
//                      } catch (InvocationTargetException e) { throw e.getTargetException();
//                      } catch (Exception e) {
//                          throw new RuntimeException("unexpected exception: " + e.getMessage());
//                      }
//                      throw new RuntimeException(m.getName() + " est une méthode inhibée ");
//                  }
//              });
//      proxy.écrire(4);
```

Démonstration ...

2) Une pile instrumentée

- **Hypothèses**

- La classe `PileInstrumentee` existe ,
- Cette classe implémente l'interface `PileI`,
- Cette classe possède ce constructeur
 - `PileInstrumentee(PileI p){ this.p = p; }`

// déjà vue ...

// c.f le patron proxy

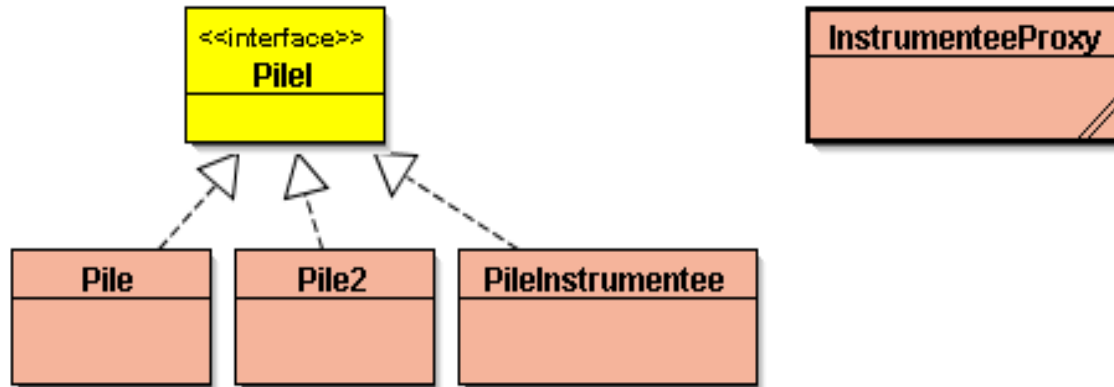
// c.f le patron adapter

- **Un « processus d'instrumentation à la volée »**

- Crée une instance instrumentée
- Intercepte tous les appels de méthodes
- Déclenche si elle existe la méthode instrumentée, sinon la méthode originale est appelée
- Capture l'assertion en échec, et affiche la ligne du source incriminé
- Propage toute autre exception

- **Comment ? → DynamicProxy + introspection**

Exemple d'initialisation



```
p = (PileI) InstrumenteeProxy.newInstance(new Pile(10));
exécuterUneSéquence(p);
```

```
public static void exécuterUneSéquence(PileI p) throws Exception{
    p.empiler("b");
    p.empiler("a");
    System.out.println(" la pile : " + p);
    ...
}
```

PileInstrumentée un extrait, explications ...

```
public class PileInstrumentee implements PileI{

    private PileI p;

    public PileInstrumentee(PileI p){
        this.p = p;
    }

    /** @pre taille > 0;
     *   @post p.capacite() == taille;
     */
    public PileInstrumentee(int taille){

        assert taille > 0 : "échec de la pré assertion taille > 0";

        this.p = new Pile(taille); // proxy

        assert p.capacite() == taille : "échec post p.capacite() == taille";

    }
}
```

PileInstrumentée un extrait, explications ...

```
/**
 * @post p.sommet().equals(o)
 * @exsures PilePleineException p.estPleine();
 */
public void empiler(Object o) throws PilePleineException{
    if(p.estPleine()){
        try{
            p.empiler(o);
            assert false : " exception attendue ???";
        }catch(Exception e){
            assert e instanceof PilePleineException;
            throw new PilePleineException();
        }
    }

    p.empiler(o);

    try{
        assert p.sommet().equals(o);
    }catch(PileVideException e){
        assert false;
    }}
}
```

Instrumentation

- Un « processus d'instrumentation à la volée »

- Crée une instance instrumentée

- ```
p = (PileI) InstrumenteeProxy.newInstance(new Pile(10));
```

- Intercepte tous les appels de méthodes

- ```
InstrumenteeProxy implements InvocationHandler
```

- Déclenche si elle existe la méthode instrumentée, sinon la méthode originale est appelée

- ```
la méthode invoke
```

- Capture l'assertion en échec, et affiche la ligne du source incriminé

- ```
catch (InvocationTargetException e) {  
    if (e.getTargetException() instanceof AssertionError) {
```

- Propage toute autre exception

- ```
throw e.getTargetException();
```

# InstrumenteeProxy : le constructeur

---

```
public class InstrumenteeProxy implements InvocationHandler{
 private Object cible;
 private Object cibleInstrumentee;
 private Class<?> classeInstrumentee;

 public InstrumenteeProxy(Object target) throws Exception{
 this.cible = target;

 // à la recherche de la classe instrumentée
 classeInstrumentee = Class.forName(target.getClass().getName()+"Instrumentee");

 // à la recherche du bon constructeur
 Constructor cons = null;
 for(Class<?> c : target.getClass().getInterfaces()){
 try{
 cons = classeInstrumentee.getConstructor(new Class<?>[]{c});
 }catch(Exception e){}
 }

 // création de la cible instrumentée
 cibleInstrumentee = cons.newInstance(target);
 }
}
```

# InstrumenteeProxy : la méthode invoke

```
public Object invoke(Object proxy, Method method, Object[] args) throws Exception{
 Method m = null;
 try{ // recherche de la méthode instrumentée, avec la même signature
 m = classeInstrumentee.getDeclaredMethod(method.getName(), method.getParameterTypes());
 }catch(NoSuchMethodException e1){
 try{ // la méthode instrumentée n'existe pas, appel de l'original
 return method.invoke(cible, args);
 }catch(InvocationTargetException e2){ // comme d'habitude ...
 throw e2.getTargetException();
 }
 }

 try{ // invoquer la méthode instrumentée
 return m.invoke(cibleInstrumentee, args);
 }catch(InvocationTargetException e){
 if(e.getTargetException() instanceof AssertionError){
 // c'est une assertion en échec
 if(e.getTargetException().getMessage() != null) // le message
 System.err.println(e.getTargetException().getMessage());
 System.err.println(e.getTargetException().getStackTrace()[0]);
 }
 throw e.getTargetException(); // propagation ...
 }
}
```

# InstrumenteeProxy, suite et fin

---

```
public static Object newInstance(Object obj) throws Exception{
 return
 Proxy.newProxyInstance(obj.getClass().getClassLoader(),
 obj.getClass().getInterfaces(),
 new InstrumenteeProxy(obj));
}
```

- Ouf !
- Simple côté client

```
p = (PileI)InstrumenteeProxy.newInstance(new Pile(10));
exécuterUneSéquence(p);
```



# Démonstration

---

- ?