Implémentation

De UML au code

Gerson Sunyé
http://sunye.free.fr

Université de Nantes

29 octobre 2013

Plan

Motivation

- Absence de sémantique dans UML.
- En d'autres termes, il n'y a pas de règle pour traduire un modèle de conception en code.
- Pour cela, il faut établir une *stratégie d'implémentation*.

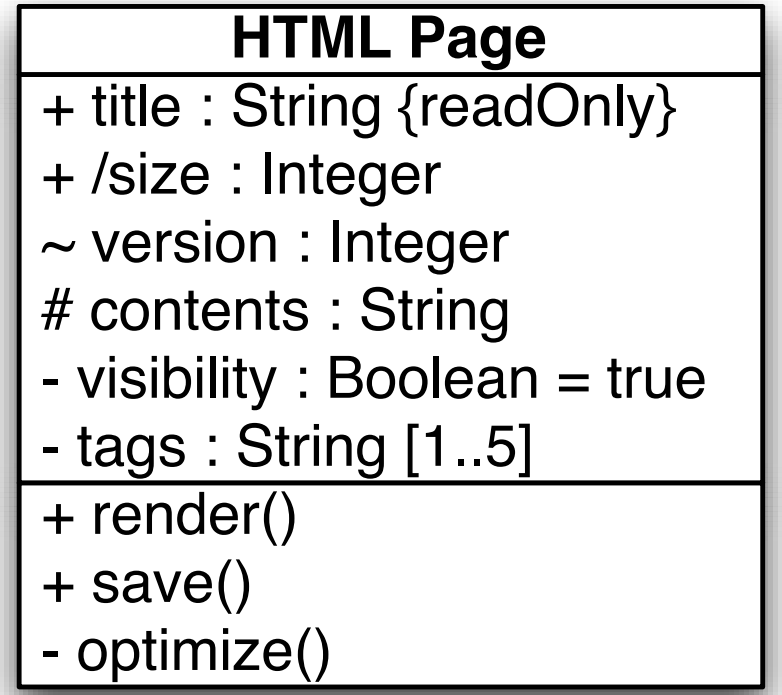
Principes

- Un modèle de conception UML n'est pas la copie conforme du code source.
- Il doit avoir le minimum possible d'informations [1] et ne doit pas être surchargé avec des informations redondantes.
- UML n'est pas un C++ visuel [2].

1. Sans perdre l'expressivité.
2. Ni Java, Eiffel, Smalltalk, Python, Sather, Ruby, Dylan, Objective-C, Clos, Perl, Beta, ObjPascal, C#, SmallScript, Scheme, Curl, Cecil, OCAML, Delphi, Oberon, Self, etc.

Exemple

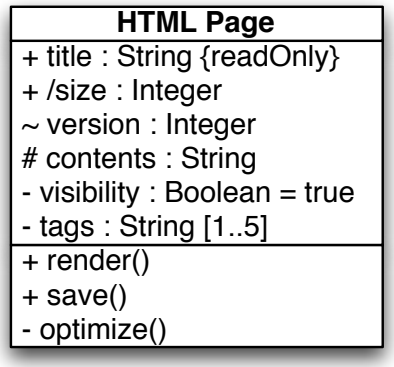
Implémentation de la classe HtmlPage



Exemple

Approche naïve: types primitifs

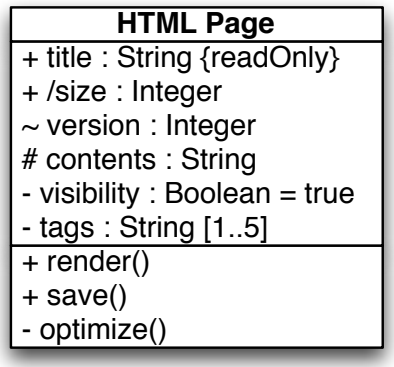
```
public class HTMLPage {
    public String title;
    int version;
    protected String contents;
    private boolean visibility =
        true;
    private String[5] tags;
}
```



Exemple

Approche naïve: classes

```
public class HTMLPage {
    public String title;
    Integer version;
    protected String contents;
    private Boolean visibility =
        Boolean.TRUE;
    private Collection<String> tags =
        new ArrayList<String>(5);
}
```



Exemple

Limites de l'approche naïve

- Traduction des types de base: plusieurs choix possibles [3].
- Implémentation des attribut dérivés.
- Implémentation des attributs en lecture seule.
- Accès/Manipulation des attributs multivalués.

3. Integer: AtomicInteger, Integer, int, short, byte, long, etc.

Plan

Types

Table de correspondance

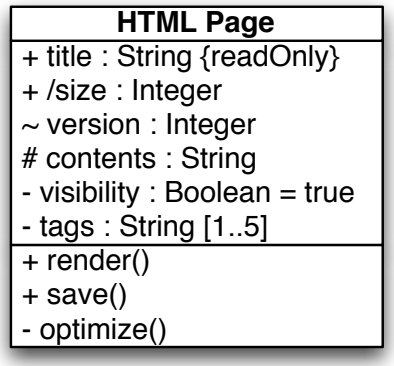
Integer	java.lang.Integer
String	java.lang.String
Boolean	java.lang.Boolean
UnlimitedNatural	fr.alma.UnlimitedNatural
Date	java.util.Date
Time	java.sql.Time

Définition d'une correspondance entre les types de conception et les types du langage de programmation.

Implémentation d'attributs

Approche mature

```
class HTMLPage {
    private String contents;
    protected String getContents() {
        return contents;
    }
    protected void setContents(String
        contents = s;
    }
    // (...)
}
```



Attributs privés, méthodes d'accès.

Approche mature

Conséquences

- Tous les attributs sont privés (encapsulation).
- La visibilité est assurée par les méthodes d'accès.
- Possibilité d'implémentation des attributs en lecture seule et dérivés.

Implémentation d'attributs

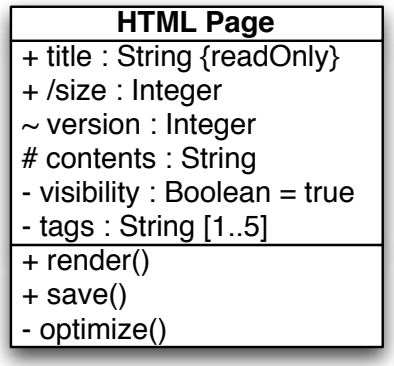
Approche *Flying Circus*

```
class HTMLPage(object):

    def getContents(self):
        return self.__contents

    def setContents(self, val):
        self.__contents = val

    contents = property(getContents, setContents)
```



Utilisation de propriétés: Pyhton, C#.

Implémentation d'attributs

Approche *Flying Circus*

```
class HTMLPage {

    private String m_contents;

    public String Contents
     {
          get
          {
               return m_contents;
          }
          set
          {
               m_contents = value;
          }
```

Approche réflexive

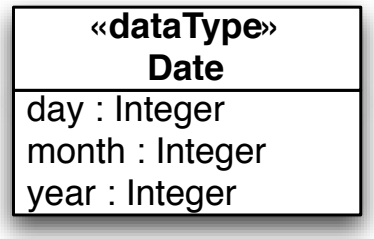
```
class HTMLPage {
    private ValueHolder<String> contents;
    private Map<String, ValueHolder> attrs;
    public String getContents() {
        return contents.value();
    }
    public void setContents(String s){
        contents.set(s);
    }
    // reflexive methods
    public void refSet(String key, Object value) {
        attrs.get(key).set(value);
    }
    public Object refGet(String key) {
        attrs.get(key).value();
```

Plan

Datatypes

Type, dont:

- les valeurs n'ont pas d'identité.
- les opérations sont de fonctions "pures".



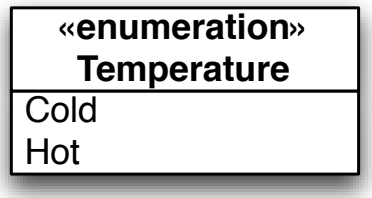
Implémentation de Datatypes

Approche martienne

```
expanded class DATE
    creation make, from_string
    feature -- Creation and initialization
    make(d : INTEGER, m : INTEGER, y : INTEGER) is
        do
            day := d ; month := m; year := y
        end -- make
    from_string(s : STRING) is
        do
            -- My students will complete this
        end -- from_string
    feature {ANY}
    add_days(d : INTEGER) : DATE is
        do
            create Result.make(day+d, month, year)
        end --add_days
    feature {NONE} -- Private
    day : INTEGER;
    month : INTEGER;
    year : INTEGER;
    invariant
        not day > 31
        not month > 12
end -- DATE
```

Enumerations

- Types dont le domaine des valeurs est défini par un ensemble de noms.
- Synonymes du type entier.



Enumerations

Approche simple

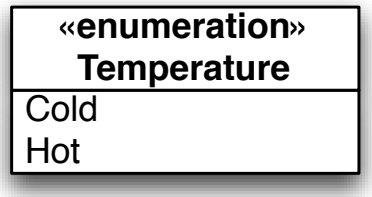
Utiliser le bon langage :

```
public enum Temperature {COLD, HOT};
```

Implémentation d'énumérations

Approche sans sûreté de typage

```
public class Temperature {
    public static final int COLD=0;
    public static final int HOT=1;
}
```



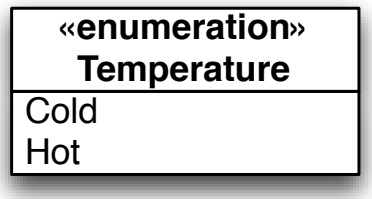
Java, Python, Eiffel, etc.

Implémentation d'énumérations

Approche avec sûreté de typage

```
public class Temperature implements
    Comparable<Temperature> {
    private final String name;
    public static final Temperature COLD=
        new Temperature("Cold");
    public static final Temperature HOT=
        new Temperature("Hot");
    private static int nextOrdinal=0;
    private final int ordinal=nextOrdinal++;

    private Temperature(String name) {
        this.name = name;}
    public int compareTo(Temperature other) {
        return ordinal - other.ordinal;}
}
```



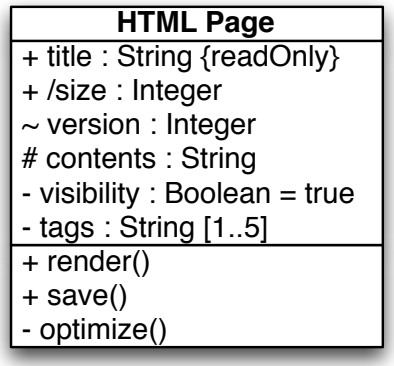
Java, Python, Eiffel, etc.

Implémentation d'énumérations

Approche martienne

```
expanded class TEMPERATURE
inherit ANY redefine out end;
feature {ANY} -- Queries
    out  : STRING
    cold : TEMPERATURE is once Result.set_image ("cold") end
    hot  : TEMPERATURE is once Result.set_image ("hot") end
    from_string (str : STRING) is
        -- this will be implemented soon by my students
    end -- from_string
feature {NONE} -- Private
   all_images : DICTIONARY[TEMPERATURE, STRING] is
      once
         create Result;
         Result.put(cold, cold.out);
         Result.put(hot, hot.out);
      end -- all_images
   set_image (new_image : STRING) is
      do
         out := new_image.twin;
      end -- set_image
end -- TEMPERATURE
```

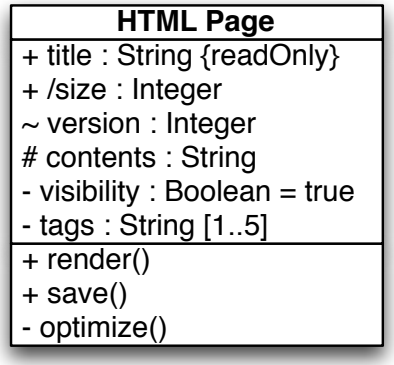
Approche simple




```
public class HTMLPage {
    // (...)
}
```

Approche mature

Interfaces

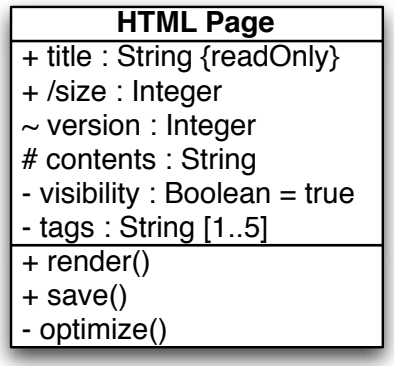



```
interface HTMLPage {
    // (...)
}

public class HTMLPageImpl implements HTMLPage {
    // (...)
}
```

Approche mature

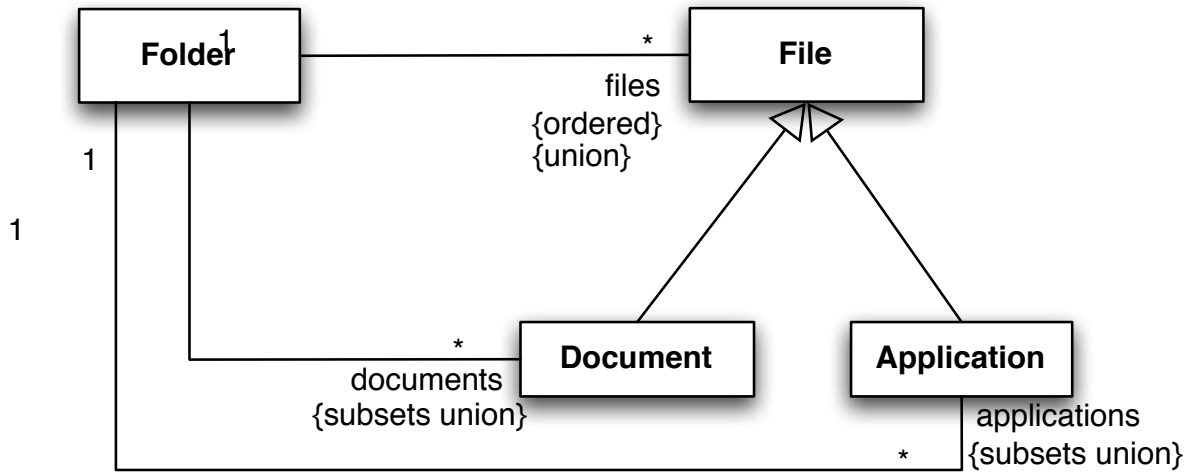
Méta-classes




```
interface HTMLPage {
    // (...)
}
private class HTMLPage {
    // (...)
}
interface HTMLPageClass {
    HTMLPage createHTMLPage();
}
public class HTMLPageClassImpl implements HTMLPageClass {
    private Collection<HTMLPage> instances;
    public HTMLPage createHTMLPage() {
        HTMLPage aux = new HTMLPageImpl();
        instances.add(aux);
        return aux;
    }
}
```

Plan

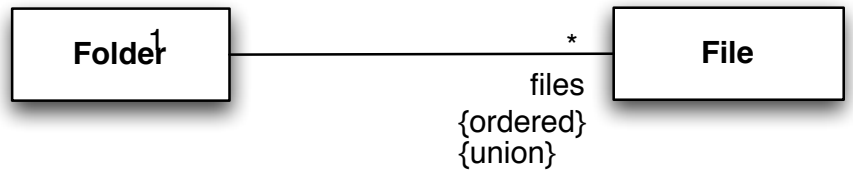
Rappel



Propriétés des rôles: set, bag, ordered, sequence, redefines, subsets, union.

Implémentation d'associations

Approche naïve

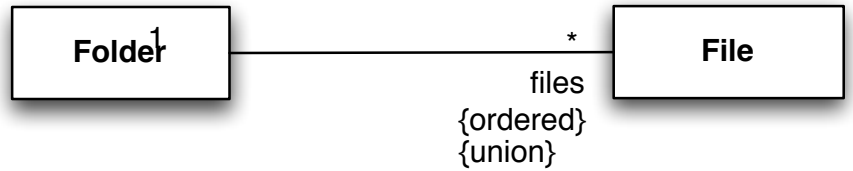



```
interface Folder {
    addFile(File);
    removeFile(File);
}
```

```
interface File {
    setFolder(Folder);
    getFolder(Folder);
}
```

Implémentation d'associations

Approche naïve




```
interface Folder {
    addFile(File);
    removeFile(File);
}
```

```
interface File {
    setFolder(Folder);
    getFolder(Folder);
}
```

Problèmes:

- Qui maintient la cohérence?
- Cohérence de l'interface?
- Que fait-on des rôles?

Implémentation d'associations

Approche kahwa

```
interface Folder {
    List<File> getFiles();
}
class FileList implements
    List<File> {
    boolean add(File) {...}
    remove(File) {...}
}
```

```
interface File {
    setFolder(Folder);
    Folder getFolder();
}
```

L'implémentation de l'interface List<File> doit s'occuper de la cohérence. Problème : et de l'autre côté?

Implémentation d'associations

Approche kahwa

```
class FileList implements List<File> {
    boolean add(File f) {
        // La liste doit connaitre Folder:
        f.setFolder(this.folder);
        files.add(f);
    }
}
class File {
    setFolder(Folder f) {
        folder.getFiles().remove(this);
        // La jolie boucle:
        folder.getFiles().add(this);
        this.folder = f;
    }
}
```

Bien évidemment, cette implémentation ne marche pas.

Implémentation d'associations

Approche kahwa

```
class FileList implements List<File> {
    boolean add(File f) {
        f.basicSetFolder(this.folder);
        files.add(f);
    }
}
class File {
    basicSetFolder(Folder f) {
        folder.getFiles().remove(this);
        this.folder = f;
    }
    setFolder(Folder f) {
        basicSetFolder(f);
        folder.basicAddFile(this);
    }
}
```

Implémentation d'associations

Approche par itérateurs [Harrison et al., 2000]

```
interface Folder {
    FileIterator getFiles();
}
interface FileIterator extends Iterator, File {
    remove();
    insert(File);
    next();
    /* File methods */
}
```

L'interface reste cohérente, même si la multiplicité maximal est de 1. L'implémentation de l'interface File permet de respecter la loi de Demeter [Lieberherr et al., 1988].

Implémentation d'associations

Approche OCL

```
interface Folder {
    Collection<File> getFiles();
}
```

```
public interface Collection<E> {
        Collection<E> select(BooleanExpression<E>);
        boolean forAll(BolleanExpression<E>);
        boolean includes(E);
        boolean includesAll(Collection<E>);
        // (...)
}
```

Multiplicités

- Vérification des multiplicités maximales et minimales.

Vérification des multiplicités

Approche naïve

```
class FileList implements List<File> {
    FileList(int upper, int lower) {...}
    add(File) {
        if (this.size() < upper) {...}
    };
    remove(File) {
        if (this.size() > lower {...}
    };
}
```

Problèmes:

- Comment fait-on lorsqu'on ne veut rien vérifier ?
- Et si l'on souhaite faire d'autres sortes de vérification ?
- La vérification de la borne inférieure n'est pas très inspirée...

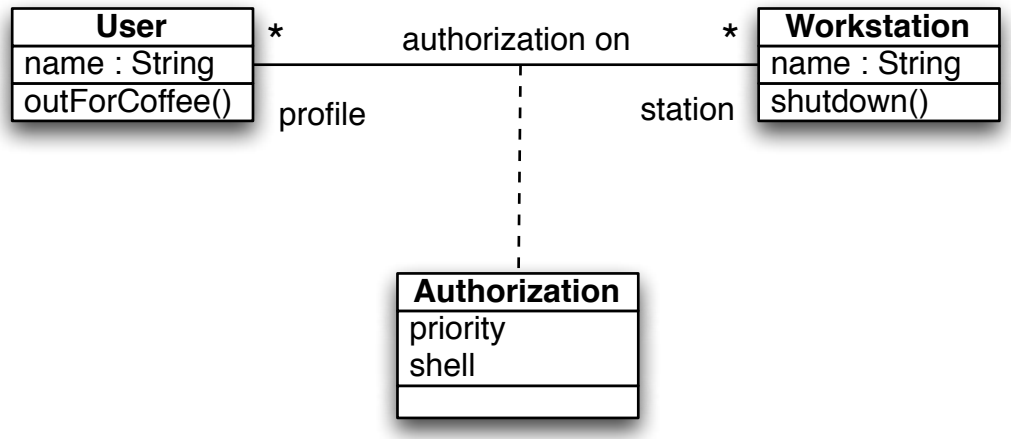
Vérification des multiplicités

Approche par décorateurs

```
public class UpperBoundDecorator<E> implements List<E> {
    protected int upper;
    protected List decoratedList;

    public UpperBoundDecorator(List<E> aList, int upper)
        this.decoratedList = aList;
        this.upper = upper;
    }
    public boolean add(E elem) {
        if (decoratedList.size() < upper)
            return decoratedList.add(elem);
        else
            throw new WrongSizeException();
    }
```

Classe-association

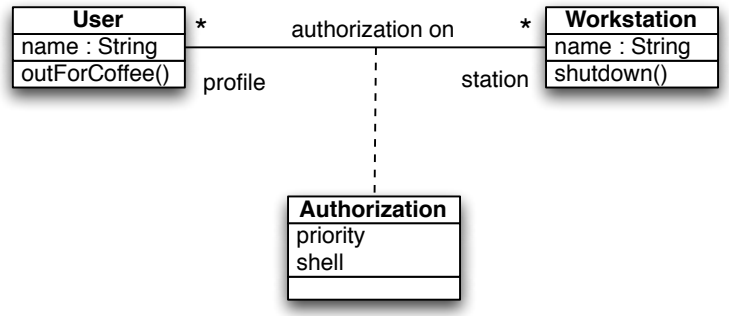


Implémentation des classes-associations

```
class AuthorizationFactory {
    private static AuthorizationFactory instance =
        new AuthorizationFactory();
    private AuthorizationFactory() {}
    public static AuthorizationFactory getInstance() {
        return instance;
    }
    public Authorization link(Workstation w, User u) {
        return new AuthorizationImpl(w,u);
    }
}
```

Implémentation des classes-associations

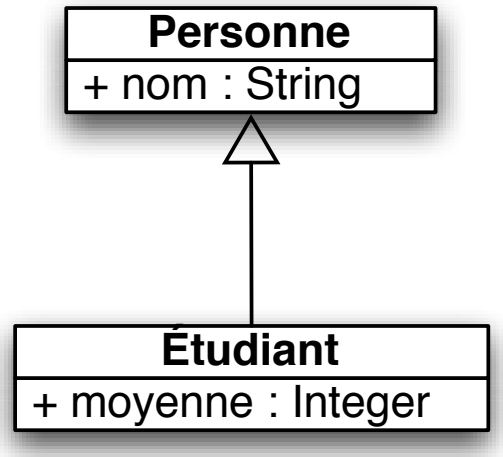
Approche mature




```
interface StationListIterator
    extends Workstation, Authorization {
    void next();
    void remove();
    void insert(Workstation);
}
```

Plan

Héritage Simple



Héritage Simple

Approche simple

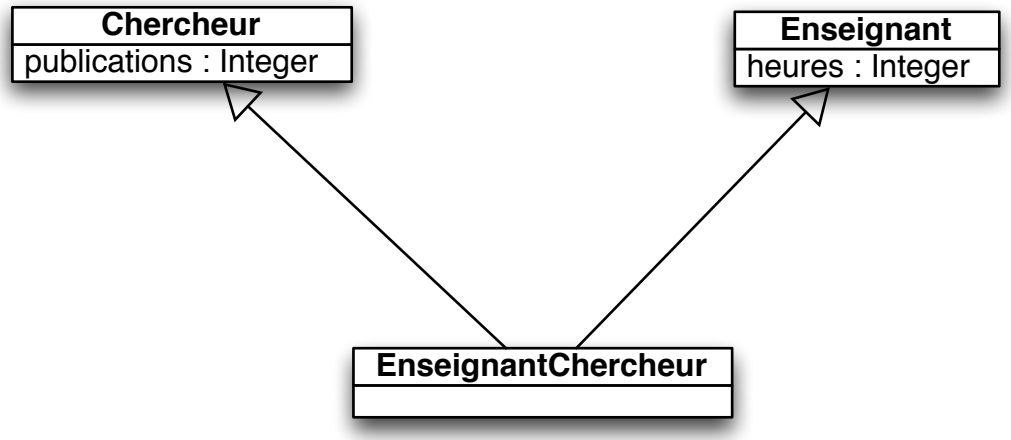
```
class Etudiant extends Personne {
    public int getMoyenne() {
        return  moyenne;
    }
    public void setMoyenne(int val){
        moyenne = val;
    }
    protected int moyenne;
}
```

Héritage Simple

Approche mature

```
interface Etudiant extends Personne {
    public int getMoyenne();
    public void setMoyenne(int);
}
class EtudiantImpl extends PersonneImpl
    implements Etudiant {
    // (...)
}
```

Héritage multiple



Héritage multiple

Approche simple

```
interface EnseignantChercheur
    extends Enseignant, Chercheur {
}
class EnseignantChercheurImpl
    implements EnseignantChercheur {
}
```

Héritage entre les interfaces. Pas d'héritage entre les classes.

Héritage multiple

Approche mature

```
class EnseignantChercheurImpl
      extends EnseignantImpl
      implements EnseignantChercheur {
      private final Chercheur chercheur;
}
```

Voir aussi :
http://csis.pace.edu/ bergin/patterns/multipleinheritance.html

Plan

Standards de codage

- Basé sur les conventions de codage de Sun.
- http://java.sun.com/docs/codeconv/

Nom de fichier

Nom de fichiers :

- README
- GNUmakefile
- Makefile
- etc.

Structure d'un fichier source

1. Commentaires (licence).
2. Importation de paquetages.
3. Déclarations de la classe/interface.
   a. Commentaires sur la classe.
   b. Déclaration de la classe/interface.
   c. Commentaires sur l'implémentation.
   d. Attributs de classe (publiques, protégés, paquetage, privés).
   e. Attributs d'instance.
   f. Constructeurs.
   g. Méthodes, groupées par fonctionnalité (et non pas par visibilité).

Identation

- Longueur des lignes 80
- Coupure des lignes :
  - Après une virgule.
  - Avant un opérateur.
  - Aligner avec le début de l'expression.

```
someMethod(longExpression1, longExpression2, longExpress
        longExpression4, longExpression5);

var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                              longExpression3));
```

Commentaires

- Utiliser les balises Javadoc.
- Voir : http://java.sun.com/javadoc/writingdoccomments
- @param, @return, @throws, @exception, @author, @version, @see, @since, @serial, @serialField, @serialData, @deprecated, {@link}, etc.

Déclarations

- Un attribut/variable par ligne.
- Si possible, initialiser la variable lors de sa déclaration.
- Au début d'un block.
- Pas d'espace entre le nom d'une méthode et la parenthèse ouvrante.
- Une ligne blanche entre chaque méthode.

Instructions

- Une instruction par ligne.
- Toujours utiliser des accolades à l'intérieur des boucles et des conditionnels.

Dénomination

Packages

- Préfix : lettres minuscules (ASCII), représentant soit le domaine (com, edu; goc, mil, net, org) ou deux lettres d'identification d'un pays (ISO 3166, 1981).
- Les autres composants du paquetage peuvent varier selon l'organisation interne (projet, machine, département).
- Exemples :
  - com.sun.eng
  - com.apple.quicktime.v2
  - edu.cmu.cs.bovik.chees
  - fr.univnantes.alma

Classes

- Les noms des classes sont des substantifs.
- Séparer les noms composés par des majuscules: e.g. ImageSprite.
- Éviter les acronymes, à l'exception de ceux qui sont largement répandus : e.g. URL, HTML.

Interfaces

- Mêmes règles que les classes.
- Exemples :
  - **interface** RasterDelegate.
  - **interface** Storing.

Méthodes

- Les méthodes sont des verbes.
- Les mots composés sont séparés par des majuscules, la première lettre en minuscule.
- Exemples :
  - run().
  - runFast().
  - getBackground().
- Les noms ne doivent pas commencer par des caractères tels que "_" ou "$", même si Java le permet.

Variables

- Des noms courts, mais significatifs.
- Des noms mnémoniques[4], i.e., qui indiquent rapidement leur rôle à l'observateur occasionnel.
- Éviter les noms d'une seule lettre (x, i, j), sauf pour les variables temporaires.

4. MNÉMONIQUE (mné-mo-ni-k') adj. Qui a rapport à la mémoire. Art mnémonique. Figures mnémoniques. S. f. La mnémonique, l'art de faciliter les opérations de la mémoire.

Constantes

- Les constantes de classe ou ANSI sont en majuscules.
- Les noms composés sont séparés par des soulignés ("_").

```
static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;
```

Programmation (1/3)

- Pas d'attributs publiques, sauf pour les classes-données (sans comportement).
- Ne pas utiliser une instance pour accéder à une propriété de classe:

```
classMethod();                  // OK
AClass.classMethod();           // OK
anObject.classMethod();         // AVOID!
```

Programmation (2/3)

- Ne pas coder directement les constantes numériques (sauf -1, 0 et 1).
- Ne pas faire des affectations en cascade :

```
fooBar.fChar = barFoo.Ichar = 'c';
```

- Ni imbriqués :

```
d = (a = b + c) + r;
```

Programmation (3/3)

- Utiliser les parenthèses pour simplifier les expressions à opérateurs multiples :

```
if (a == b && c == d)      // AVOID!
if ((a == b) && (c == d)) // RIGHT
```

Renvoi de valeurs

Remplacer	Par
`if (booleanExpression) { return true; } else { return false; }`	`return booleanExpression;`
`if (condition) { return x; } return y;`	`return (condition ? x : y);`

Commentaires spéciaux

XXX : Mauvais code, mais qui fonctionne.

FIXME : Mauvais code, qui ne fonctionne pas.

TODO : À faire.

Conventions additionnelles

Extraites du projet ArgoUML

- Toutes les variables d'instance sont privées.
- Utilisation de Javadoc pour chaque classe, attribut et méthode.
- Pas de commentaires à l'intérieur d'une méthode [5].

5. Si un morceau de code a besoin d'être commenté, il faut l'isoler et le placer dans une autre méthode.

Bibliographie


William Harrison, Charles Barton and Mukund Raghavachari.
Mapping UML designs to Java.
OOPSLA, 178-187, 2000.


Karl J. Lieberherr, Ian Holland and Arthur J. Riel
Object-oriented programming: An objective sense of style.
OOPSLA, 323-334, 1988.