Java, les objets : tout de suite !
Conception de classes (1)

Cnam Paris
jean-michel Douin, douin@cnam.fr
Version du 6 Mars 2003

Notes de cours associées au chapitre 7
tutorial BlueJ
http://www.bluej.org/doc/documentation.html

Ce support accompagne, référence le livre de David J. Barnes & Michael Kölling
Objects First with Java A Practical Introduction using BlueJ
Pearson Education, 2003 ISBN 0-13-044929-6.

Sommaire

- **Conception de classes**

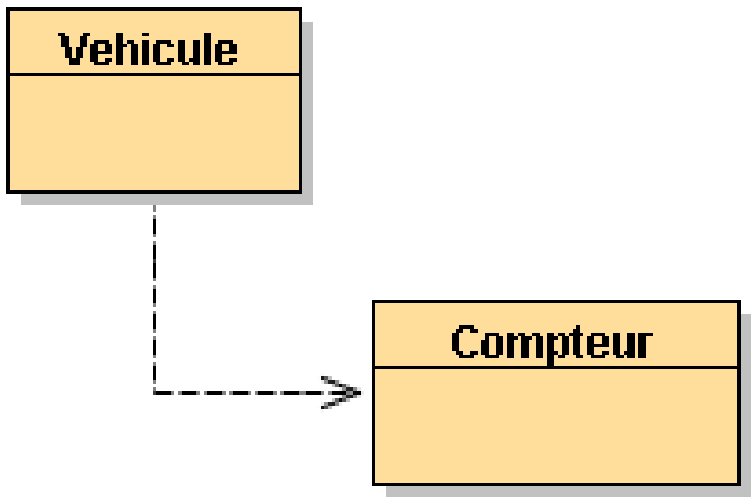
  Comment assurer la maintenance, réutilisation, robustesse ?

- **Couplage et cohésion**

  couplage faible et cohésion forte
  méthodologie basée sur les Patterns
  Pattern ou Modèles de Conception Réutilisables

Couplage

- **Liens unissants les classes**



Couplage fort

- **Les classes sont imbriquées**

  Ne peuvent être séparées simplement

  Bannir les variables d'instances « public » et « protected »

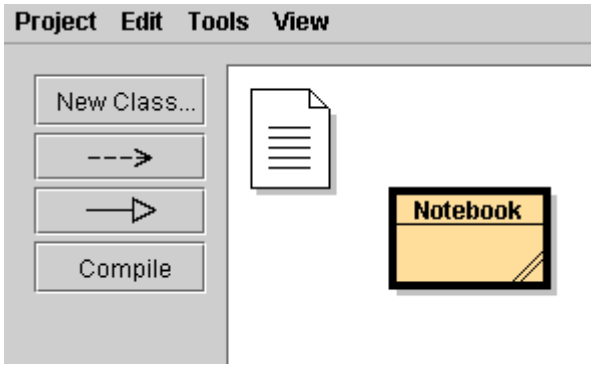
- **Tests interdépendants**

Couplage faible si :

- La classe peut être testée séparément
- La substitution d ’une classe par une autre est possible et aisée
- L ’ajout d ’une nouvelle fonctionnalité est isolée et ne concerne qu ’une classe

- Une idée du couplage faible Code 4.1 page 79

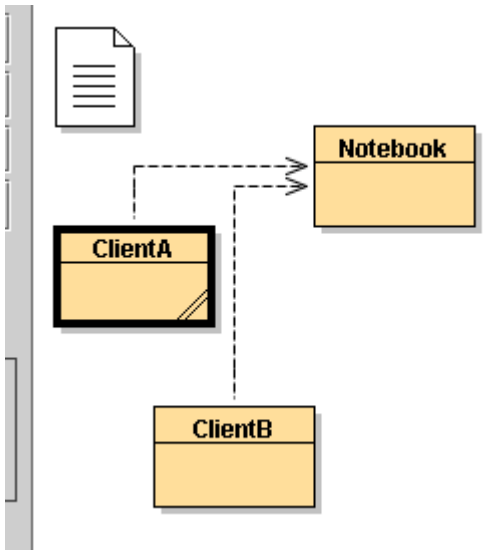
Exemple Code 4.1 page 79




```
import java.util.ArrayList;

/**

 */
public class Notebook{
    // Storage for an arbitrary number of notes.
    private ArrayList notes;

    public Notebook(){
        notes = new ArrayList();
    }
```

Deux clients de la classe Notebook

- **Visibilité des clients :**



Constructor Summary

Notebook() Perform any initialization that is required for the notebook.

Method Summary

void	**listNotes**() List all notes in the notebook.
int	**numberOfNotes**()
void	**removeNote**(int noteNumber) Remove a note from the notebook if it exists.
void	**storeNote**(java.lang.String note) Store a new note into the notebook.

Les Modèles fondamentaux

- **Pattern Délégation**
  Couplage faible
- **Pattern Immutable**
  Robustesse

abordés dans des chapitres ultérieurs

- **Pattern Interface**
- **Pattern Proxy**

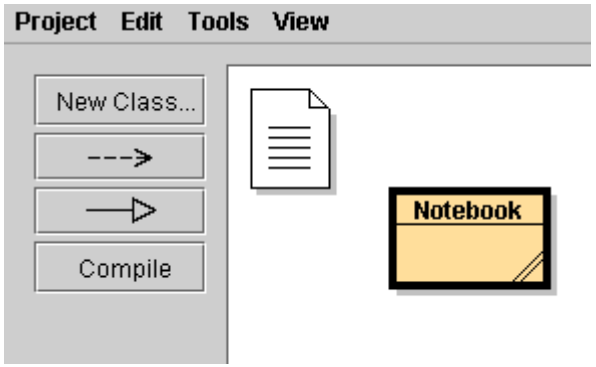
Notebook : une modification

- **java.util.Vector remplace ArrayList**

```
import java.util.Vector;

/**

 */
public class Notebook{
    // Storage for an arbitrary number of notes.
    private Vector notes;

    public Notebook(){
        notes = new Vector();
    }
```

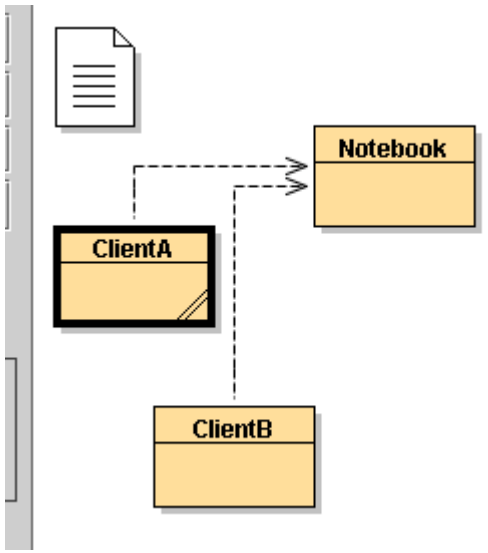
Code 4.1 page 79 avec java.util.Vector

```
public void storeNote(String note){
  notes.addElement(note);
}
public int numberOfNotes() {
  return notes.size();
}

public void removeNote(int noteNumber){
  if(noteNumber < 0) {
  } else if(noteNumber < numberOfNotes()) {
            notes.removeElementAt(index));
  } else {}
}

public void listNotes() {
        int index = 0;
        while(index < notes.size()) {
            System.out.println(notes.elementAt(index));
            index++;
        }}}
```



Deux clients de la classe Notebook

- **Modification sans conséquence**
- **Visibilité des clients : la même**



Constructor Summary

Notebook() Perform any initialization that is required for the notebook.

Method Summary

void	**listNotes**() List all notes in the notebook.
int	**numberOfNotes**()
void	**removeNote**(int noteNumber) Remove a note from the notebook if it exists.
void	**storeNote**(java.lang.String note) Store a new note into the notebook.

C ’est le Pattern Délégation

```
public class Notebook{
    private Vector notes;  // ou ArrayList notes; page 79

    public Notebook(){
        notes = new Vector();
    }
     public void storeNote(String note){
        notes.addElement(note); // ou notes.add(note); page 79
     }
```

- Acquisition d ’une instance de la classe et
- Délégations des opérations à cette instance **(ici notes)**

- **Avantages**

  Couplage faible

  Réutilisation, maintenance, évolution accrues

Schéma du Pattern Délégation

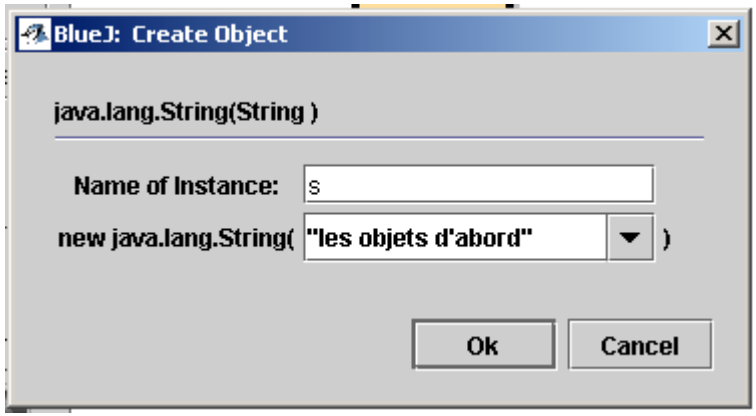
```
public class ClasseA{
  //données d 'instance
  private ClasseB classeB;
  //constructeur
  public classeA(){ classeB = new ClasseB(); }
  //méthodes
  public void m(){classB.m1();} // délégation à m1
  ...
}

public class ClasseB{
  public void m1(){...}
}
```

Les clients utilisent la classe A

Pattern Immutable

- **Robustesse et maintenance**
- **Une fois créé, l ’état de l ’objet ne peut changer**
  - **Pas de méthodes permettant de modifier l ’état interne,**
  - **Uniquement des accesseurs ou des méthodes de lecture,**
  - **Toute méthode qui engendre un nouvel état doit mémoriser celui_ci dans une nouvelle instance de la même classe, plutôt que de modifier l ’état de l ’objet**
- **Exemple : java.lang.String**
  - **Avec BlueJ menu Tools -> Use LibraryClass -> java.lang.String**

Immutable comme java.lang.String



- **L ’état de s est figé**

  si « les objets d ’abord » représente l ’état de l ’objet, celui-ci ne peut être modifié

- **// même si l ’écriture ci-dessous tente d ’indiquer le contraire ...**

```
String s = new("les objets d'abord");
s.toUpperCase();
System.out.println(s);
```

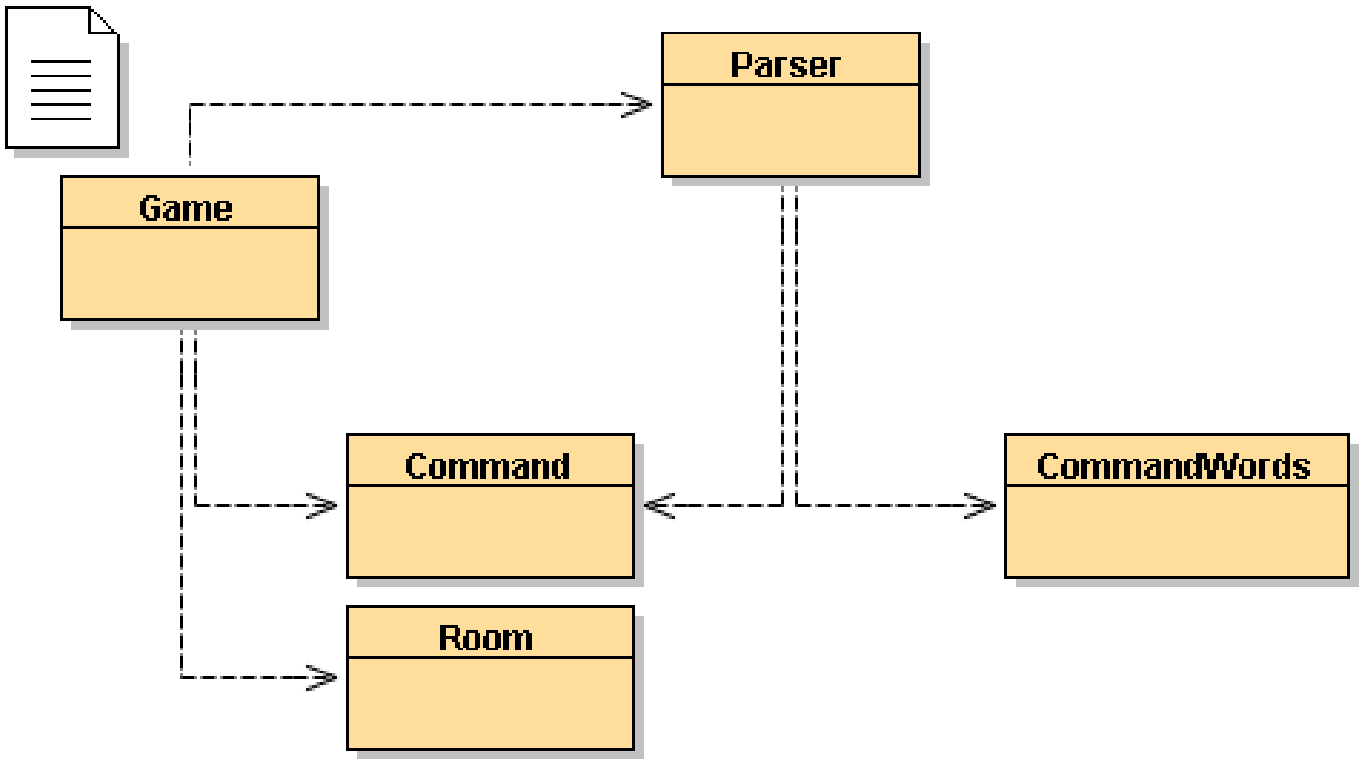
Pattern Immutable un exemple

```
public class Point{
  private int x;
  private int y;

  public Point(int x, int y){
    this.x = x;
    this.y = y;
  }

  public int getX(){   return x; }

  public int getY(){   return y; }

  public Point déplacer( int enX, int enY){
    return new Point( x + enX, y + enY);
  }
}
```

Schéma du Pattern Immutable

```
public class ClasseA{
  //données d 'instance
  private ClasseB état;


  //constructeur
  public ClasseA(){ état = new ClasseB(); }


  //méthodes
  public ClasseB getEtat(){return état;}


  public ClasseA m( ){
    ...
    return new ClasseA(état.m());
  ...
}
```

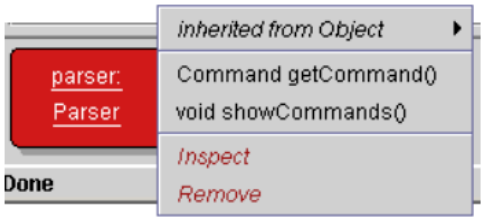
Zuul : le découpage logique

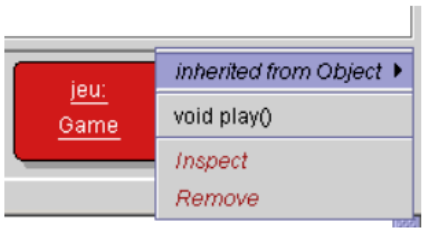


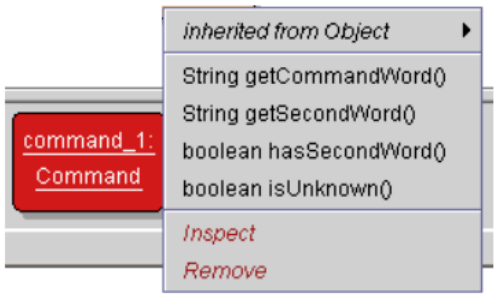
Les classes de zuul

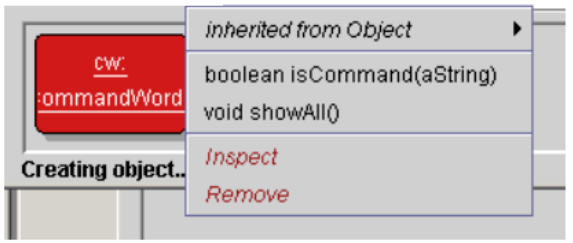
- La classe "CommandWords" recense les mots-clés autorisés et détermine si une commande de l'utilisateur est valide.
- La classe "Command" engendre une commande de l'utilisateur.
- La classe "Room" permet de créer les pièces du jeu, une pièce représente une scène.
- La classe "Parser" permet de lire le texte entré par l'utilisateur et le transforme en commande.
- La classe "Game" est la classe principale du jeu (méthode play)

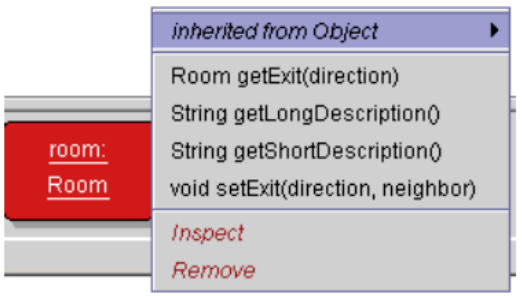
Les méthodes de chaque classe











Comment découper ? Quelle recette ?

- discussion

Les patterns : une réponse

- La classe : granularité trop fine
- Assemblage de classe = un nom

Donc

- Meilleure Abstraction dans le développement
- Distribution de la responsabilité des classes
- Maintenance plus facile
- Évolution plus aisée
- Réutilisation accrue.

- Mais comment choisir ?

Résumé, synthèse

- **Nécessaire abstraction**
- **Interface et proxy abordés un peu plus tard**
- **D 'autres Patterns**

  un catalogue initial existe (livre culte)

  Design Pattern par Gamma, Help, Johnson et Vlissides Éditeur Addison-Wesley ISBN 0-201-63361-2

  Les 3 tomes de M.Grand, Patterns in Java

  http://www.patterndepot.com/put/8/JavaPatterns.htm