M2.23.5 Programmation

Travaux pratiques d'algorithmique du deuxième semestre

Jean-François Remm
Jean-François Berdjugin

M2.23.5 Programmation: Travaux pratiques d'algorithmique du deuxième semestre

par Jean-François Remm et Jean-François Berdjugin

Table des matières

Liste des illustrations

Présentation et objectifs

Pour réaliser ces TP, vous disposez de 6 séances de trois heures. Nous utiliserons l'environnement de développement *eclipse*, si vous souhaitez l'installer, chez vous, vous pouvez le télécharger à l'URL suivante : `http://www.eclipse.org/` . Nous allons aborder la programmation aux travers de 2 séries de 3 TP :

- Introduction à la programmation
- Programmation d'un jeux de type "shoot them up"

Sources :

- `http://fr.wikipedia.org/wiki/Accueil` pour les définitions
- `http://www.planetalia.com/cursos/` pour la partie graphique
- `http://www.cokeandcode.com/info/tut2d.html` Space Invader
- `https://fivedots.coe.psu.ac.th/~ad/jg/` Killer Game Programming in Java

Chapitre 1. Introduction à la programmation

Communication entre objets

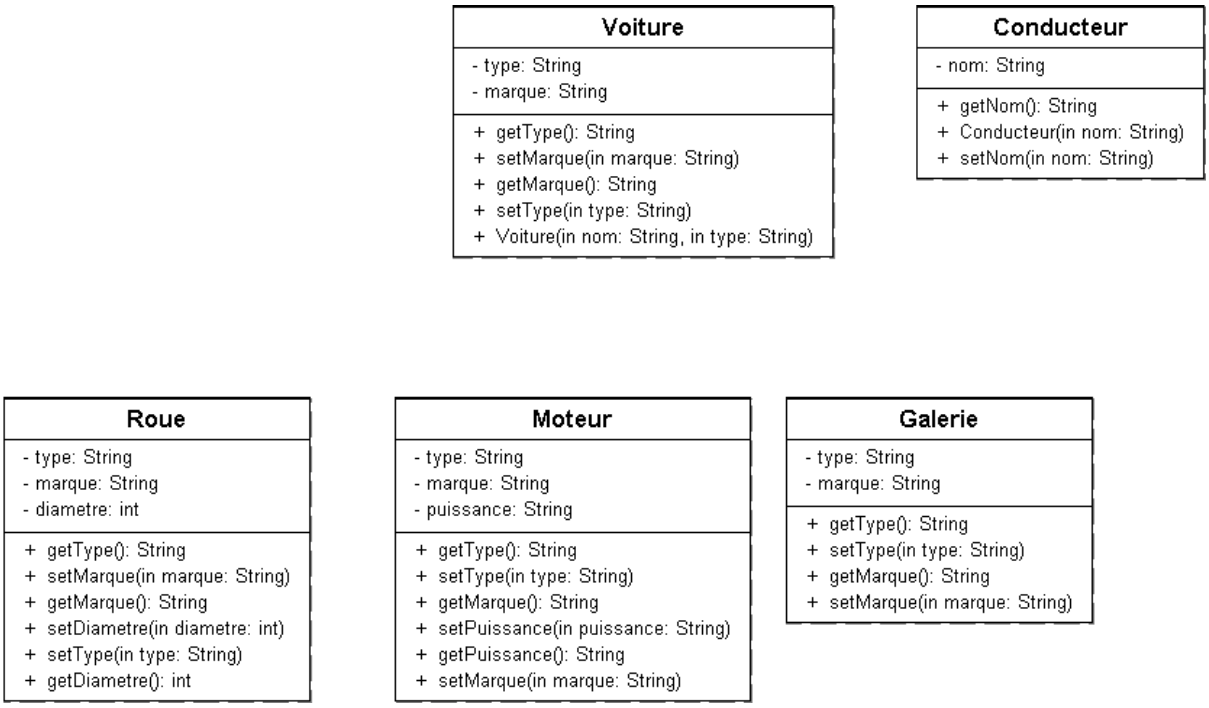
Nous allons illustrer la communication entre objets au sein d'un même processus. Les objets communiquent par messages, lorsqu'un objet utilise une méthode d'un autre objet, il envoie, à ce dernier, un message lui demandant de rendre un service. La notation UML(Unified Modeling Language) reconnaît au sein du diagramme de classe plusieurs associations qui traduisent la communication :

Association simple	les objets sont sémantiquement liés
L'agrégation partagée	les cycles de vie sont indépendants, les objets sont créés et détruit séparément.
L'agrégation de composition	les cycles de vie sont liés.

Les classes d'origine

Coder les classes *Voiture*, *Conducteur* et *Moteur* du diagramme de classe UML suivant.
Vous pouvez utiliser le copié collé de votre IDE (copier Voiture en Moteur).

Figure 1.1. Classe isolées



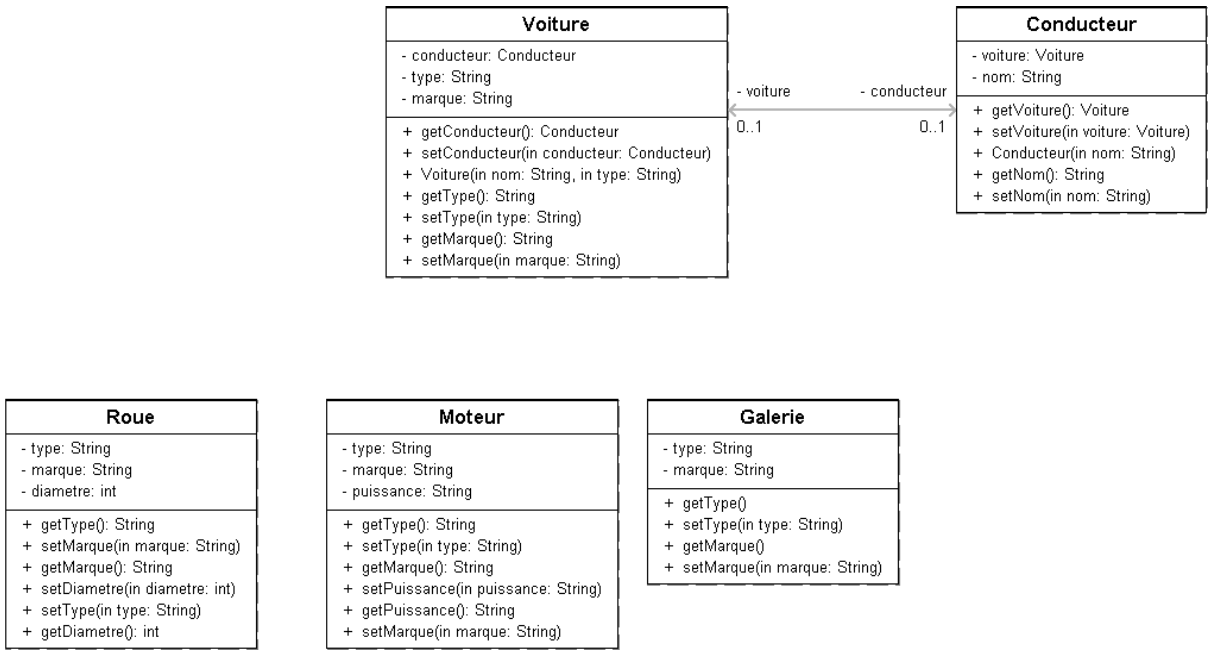
A ce stade nos classes ne peuvent traduire des associations, comment traduire qu'un véhicule appartient à une personne ?

Association simple

Comment pouvons nous traiter le cas général ou le *conducteur* a un *véhicule* et le *véhicule* est possédé par un *conducteur* : en rajoutant des attributs : *conducteur* dans la classe *Vehicule* et *vehicule* dans *Conducteur*.

Coder les méthodes et les attributs du diagramme suivant, puis tester en créant un utilisateur "Durand" qui possède une "R5 Renault".

Figure 1.2. Association simple

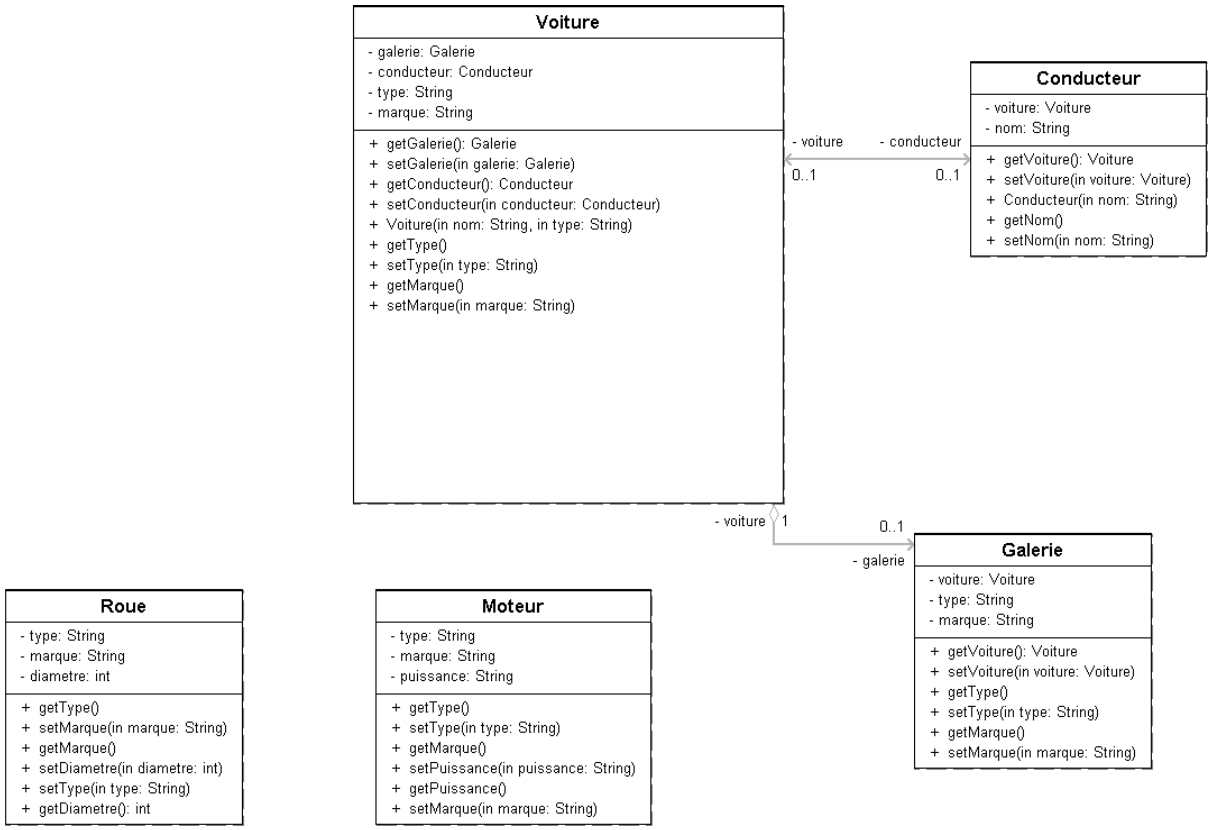


Agrégation partagée

Nous n'allons pas mettre en oeuvre l'agrégation partagée car les cardinalités de notre association étant 1-1, la manoeuvre serait équivalente. Nous voulons une voiture à options, qui possède ou qui ne possède pas une galerie.

L'association partagée correspond à une association "groupe-élément" ou la disparition du groupe n'entraîne pas la disparition de l'élément.

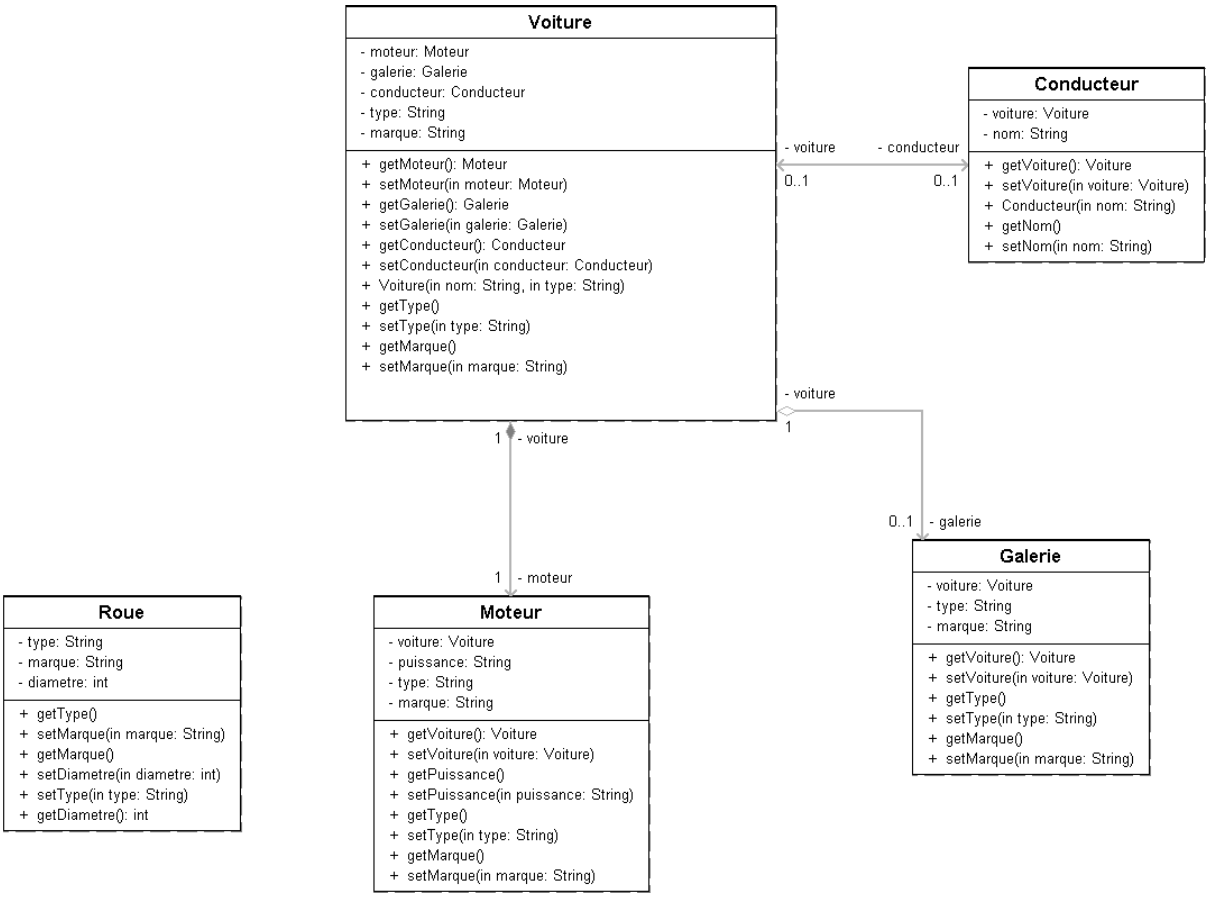
Figure 1.3. Agrégation partagée



Agrégation de composition

Nous allons maintenant traduire le fait que le moteur fait partie intégrante de la voiture, si la voiture est détruite alors le moteur l'est aussi. Implémenter le diagramme suivant.

Figure 1.4. Agrégation de composition

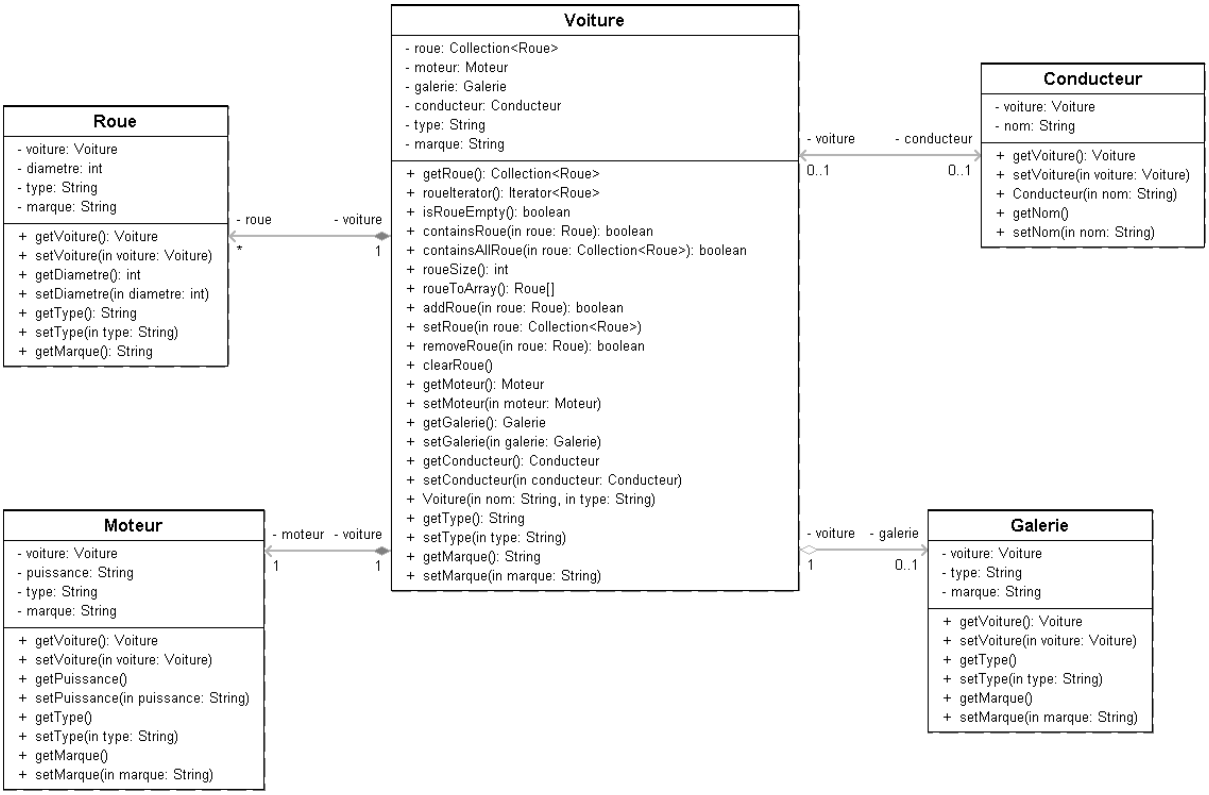


Cette solution n'est pas satisfaisante, il nous faut créer le moteur dans la voiture (*new Moteur()*). Deux possibilités existent, soit dans un constructeur soit dans une méthode. Nous allons modifier *setMoteur(Moteur m)* pour que l'attribut moteur soit instancié et soit une recopie de *m*. Le moteur de voiture étant une copie d'un autre moteur, les deux moteurs évoluent indépendamment.

Créer un moteur, puis une voiture, puis associer le moteur créé à la voiture enfin modifier le moteur, celui de la voiture est-il modifié.

Si vous voulons utiliser des cardinalités différentes de 0 ou 1 nous devons utiliser des collections, comme l'illustre la figure suivante :

Figure 1.5. Cardinalités multiples



Nous venons de voir comment faire communiquer des classes, nous allons maintenant apprendre une nouvelle façon de factoriser le code en réutilisant des classes.

Héritage, classe abstraite et interface

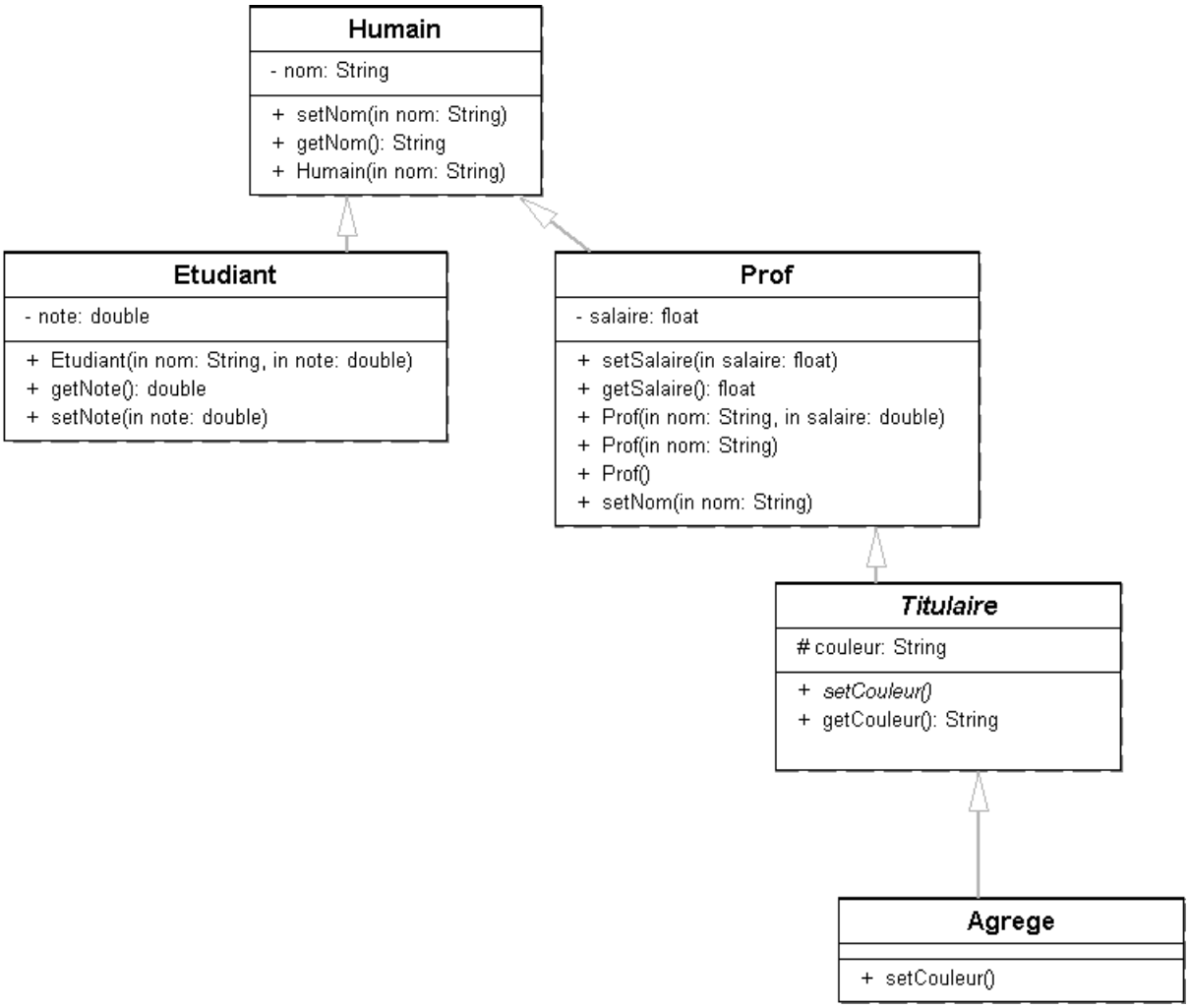
Nous allons étudier comment réutiliser des classes (le code source n'est pas nécessaire).

Héritage

L'héritage est un principe de la programmation orientée objet, permettant entre autre la réutilisabilité et l'adaptabilité des objets. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est basé sur des classes dont les "*filles*" héritent des caractéristiques de leur(s) "*mère*(s)". Les classes possèdent des attributs et/ou des méthodes qui leurs sont propres, et qui pourront être transmis aux classes filles découlant des classes mères. Chacune des classes filles peut donc posséder les mêmes caractéristiques que ses classes mères et bénéficier de caractéristiques supplémentaires à celles de ces classes mères. Chaque classe fille peut, si le programmeur n'a pas défini de limitation, devenir à son tour classe mère.

Nous allons travailler avec le diagramme suivant :

Figure 1.6. Arbre d'héritage



Commencer par coder la classe *Humain*.

Créer la classe Etudiant qui hérite de Humain (*public class Etudiant extends Humain*). La classe Etudiant disposera de toutes les méthodes de la classe *Humain*. *Humain* ne possédant pas de constructeur par défaut vous devrait faire appel explicitement au constructeur *public Humain(String nom)* avec *super(param)*. *super* possède la même syntaxe que *this* mais référence la classe mère.

De même coder la classe *Prof*, cette classe doit *redéfinir* la méthode *setNom(String nom)*. Le constructeur *Prof()* créé un *prof* avec un *salaire* de 0 et sans *nom*(*null* ou *""*), le constructeur *Prof(String nom)* créé un *prof* de nom *nom* avec un *salaire* de 0. *La surcharge n'est pas la redéfinition, la surcharge consiste à avoir plusieurs méthodes de même nom et la redéfinition consiste à réécrire le code d'une méthode héritée.*

Tester votre code.

Classe abstraite et interface

En programmation orientée objet (POO), une classe *abstraite* est une classe dont l'implémentation n'est pas complète et qui n'est pas instanciable. Elle sert de base à d'autres classes *dérivées* (*héritées*).

Créer la classe abstraite *Titulaire* (*public abstract class Titulaire extends Prof*) qui possède la méthode abstraite *setCouleur* (*public abstract void setCouleur();*), l'attribut *couleur* doit être déclaré *protected* (# en UML) pour pouvoir être modifié dans les classes héritées.

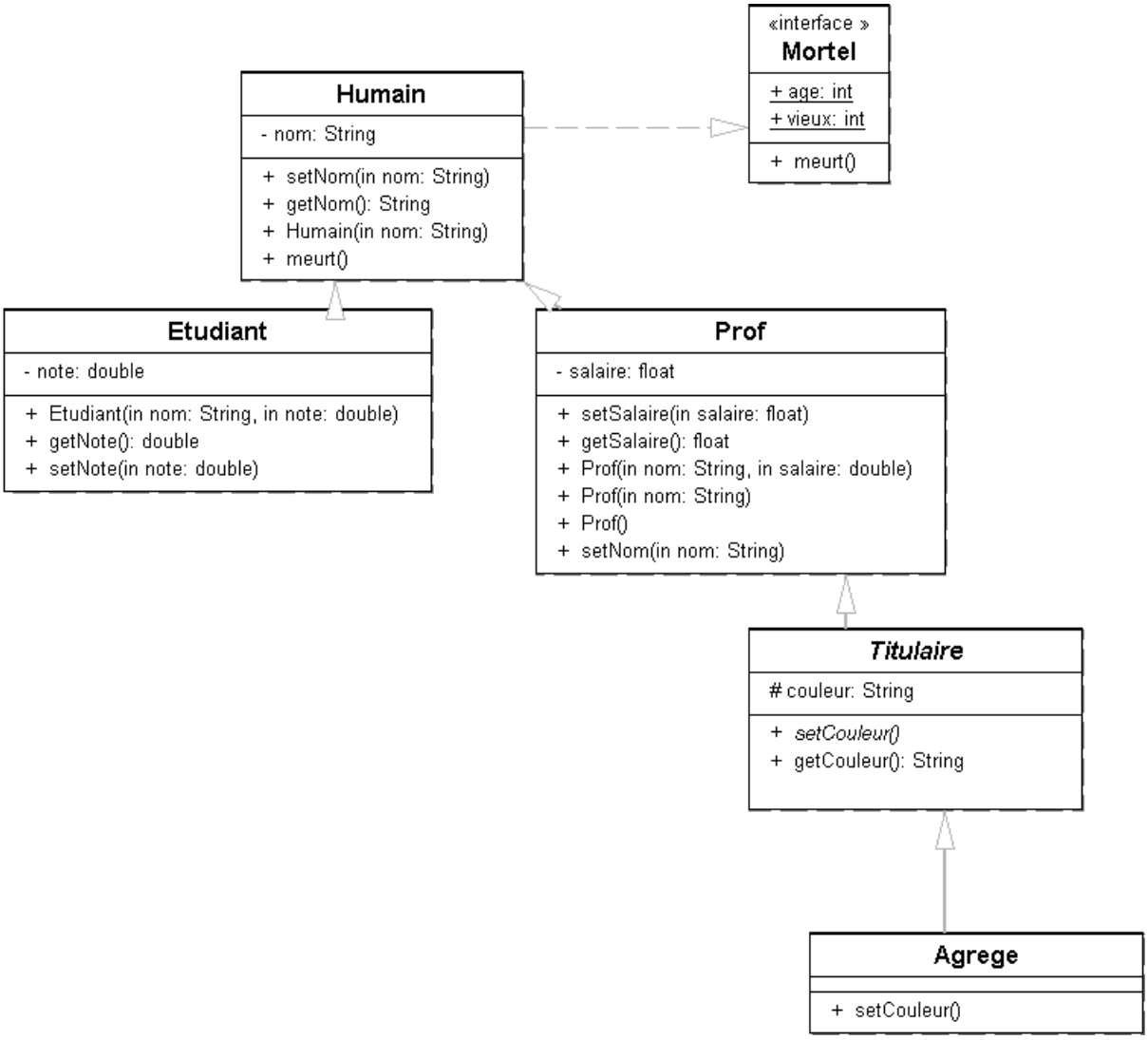
Créer la classe *Agrege*, les agrégés sont roses. Tester votre code.

Une *interface* est une classe abstraite sans implémentation. Toutes les méthodes sont abstraites. Seule l'interface de la classe apparaît. Nous allons créer une interface *Mortel* dont le code est le suivant :

```
public interface Mortel {
 public final static int vieux = 40;
 public int age=18;
 public void meurt();

}
```

Comment pouvons nous rendre les humains mortels en implémentant l'interface (implements Mortel). Réaliser cette opération et faite mourir un agrégé.

Figure 1.7. Interface



Polymorphisme

Le polymorphisme veut dire que le même service, aussi appelé opération ou méthode, peut avoir un comportement différent suivant la Classe dans laquelle il est utilisé. Nous l'avons vu avec la surcharge mais il peut être introduit en utilisant l'héritage. Utiliser les lignes suivantes :

```
Object a = new Agrege(); //surclassement
Agrege o = new Object();
```

Laquelle fonctionne ?

Essayons maintenant

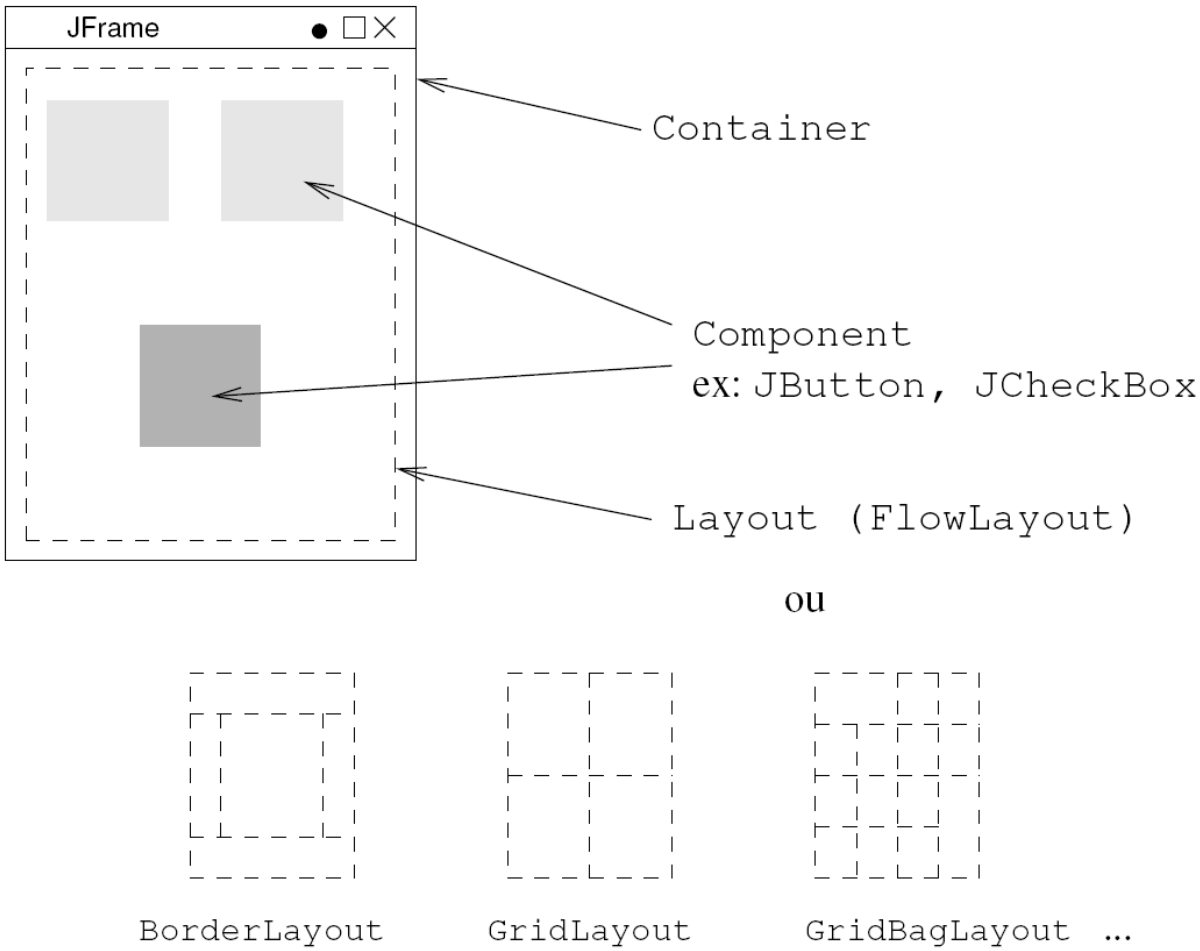
```
Object a = new Agrege(); //surclassement
Humain h = (Humain) a; //transtipage
h.setNom("toto");
```

Comme nous le voyons, la méthode à exécuter est déterminé à l'exécution et non pas à la compilation, par contre le surclassement est réalisé à la compilation.

Swing

Le but de ce TP est de réaliser vos premières applications utilisant l'API *javax.swing*.

Figure 1.8. Structure d'une JFrame



Introduction

La conception d'une interface graphique à l'aide de l'API *swing* comprend plusieurs étapes dont voici un résumé (adapté du tutorial *Java*) :

- Optionnel : choix des décorations (bordure, titre, aspect des boutons de plein écran, fermeture, ...) des fenêtres :
  - celui du look and feel courant :

    ```
    JFrame.setDefaultLookAndFeelDecorated(true);
    ```

  - celui du système de fenêtre courant : cas par défaut
- choix du Container de base : soit une *JFrame* (fenêtre standard), soit un *JPanel* (panneau dans une fenêtre déjà existante, comme c'est le cas pour une *JApplet*)

```
JFrame f = new JFrame("Titre") ;
```

- Optionnel : choix de l'action à faire lors de la fermeture de la fenêtre

  ```
  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) ;
  ```

  A faire impérativement sous Eclipse pour libérer la mémoire.

- association d'un *LayoutManager* (gestionnaire de répartition au container). Depuis la version 1.5 du JDK, on n'est plus obligé de passer par le *contentPane*. On peut directement faire ceci :

  ```
  f.setLayout(new FlowLayout()) ;
  ```

- ajout des Components (*JButton*, *JCheckBox*, ...). Remarque identique qu'à l'étape précédente : jusqu'à la version de JDK précédent la 1.5, on aurait du faire f.getContentPane().add(...). Là on peut écrire :

  ```
  f.add(new JButton("Bouton")) ;
  ```

- dimensionnement du Container :

  - choisie

    ```
    f.setSize(400,400) ;
    ```

  - donnée par la taille préférée du contenu :

    ```
    f.pack() ;
    ```

- le rendre visible (c'est mieux)

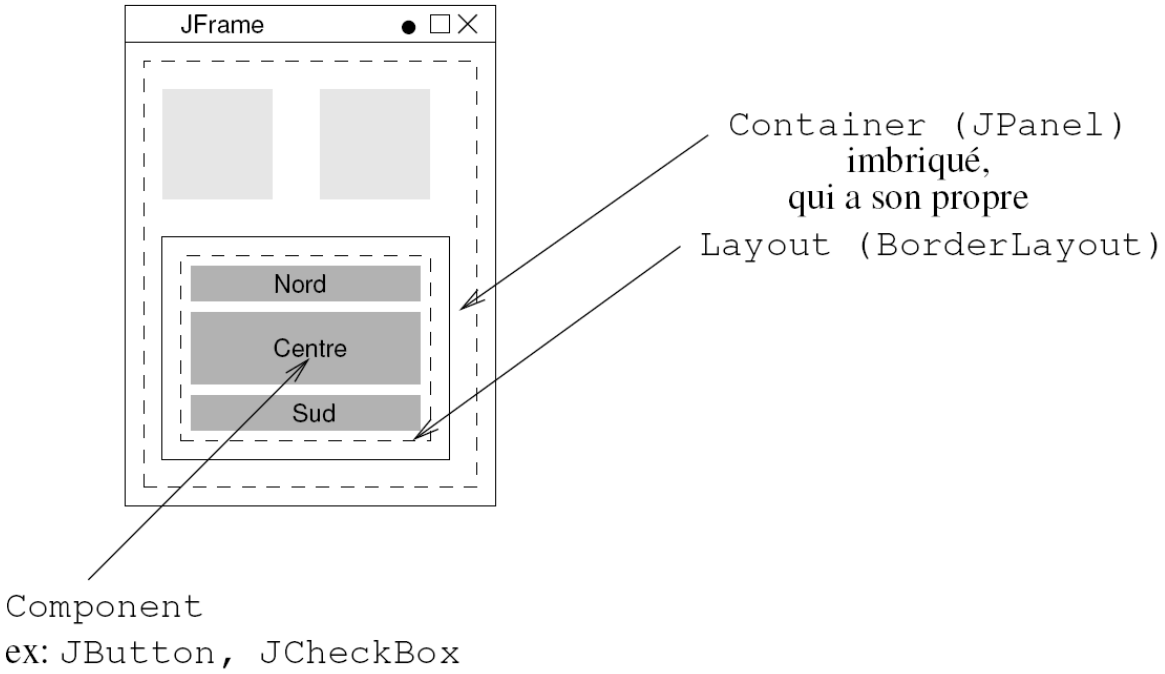
  ```
  f.setVisible(true) ;
  ```

La conception d'une interface graphique peut se faire à plusieurs niveaux selon le vieil adage diviser pour mieux régner en imbriquant plusieurs Container (*JPanel*), en associant à chacun, un *LayoutManager* différent. La classe qui définit notre IHM (Interface Homme Machine), peut hériter de Container) (*JFrame* par exemple) et mettre les autres Components en variables d'instances :

Figure 1.9. IHM 1


```
import javax.swing.*;
import java.awt.*;
public class IHM extends JFrame
{
JButton monBouton;
// contructeur de IHM
public IHM(String titre)
{
super(titre);
setLayout(...);
monBouton = new JButton("Click here !!!");
add(monBouton...);
setSize(...);
setVisible(...);
}
```

Figure 1.10. Méthodologie




```
public static void main(String[] args)
{
IHM monIHM = new IHM("Ma frame à moi");
}
```

ou utiliser tous les Components (Container compris) en variables d'instances :

```
import javax.swing.*;
import java.awt.*;
public class IHM2
{
JFrame f;
JButton monBouton;
// contructeur de IHM2
public IHM2(String titre)
{
f = new JFrame(titre);
f.setLayout(...);
monBouton = new JButton("Click here !!!");
f.add(monBouton...);
f.setSize(...);
f.setVisible(...);
}
public static void main(String[] args)
{
IHM2 monIHM = new IHM2("Ma frame à moi");
}
```

Réalisations

Fenêtre simple

Écrire un programme qui fasse apparaître une fenêtre avec comme titre "Essai de Frame" (dimension de la JFrame 200x200)

Figure 1.11. Essai de Frame

Méthode :

- Consulter l'API (paquetages java.awt et javax.swing)
- Créer la JFrame
- Dimensionner la JFrame (à l'aide des méthodes setSize(int,int) ou pack())
- La faire afficher

Disposition de composants graphiques dans une JFrame

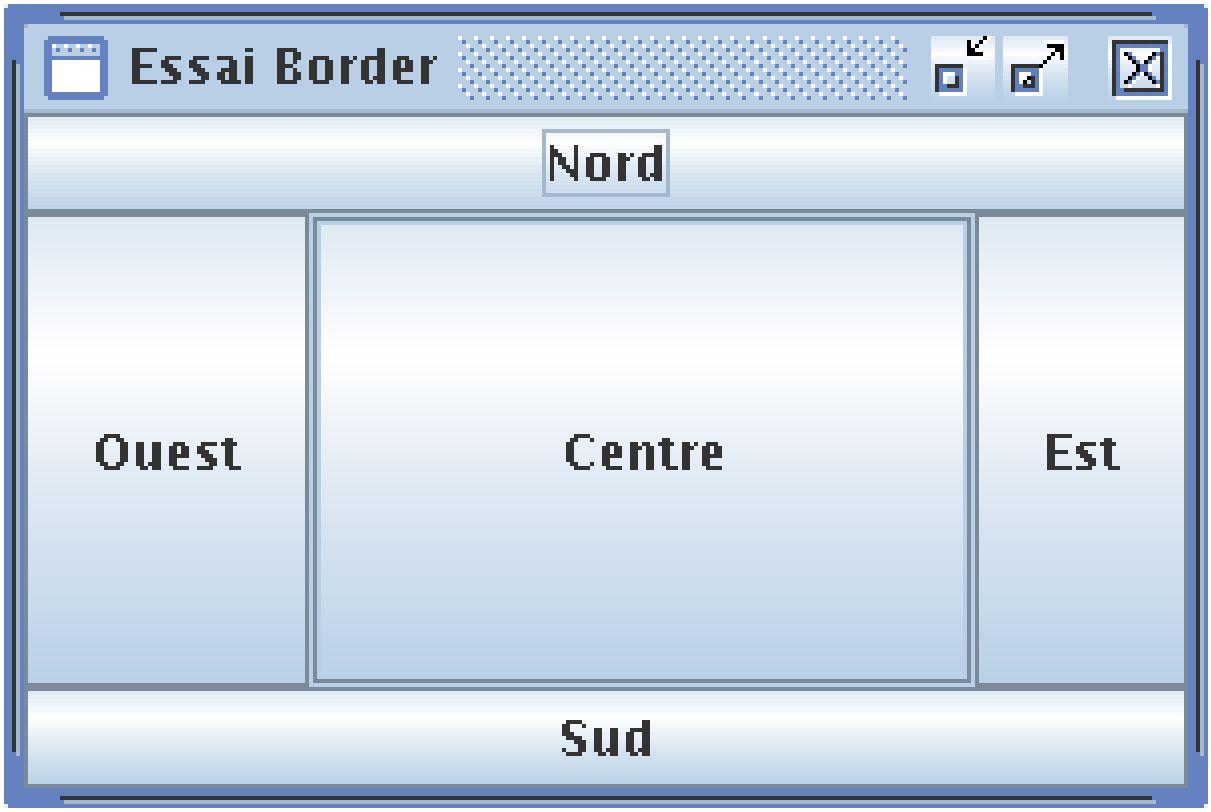
Dans la partie qui suit nous allons enrichir notre JFrame en ajoutant différents composants, la méthodologie suivante sera utilisée :

- créer la JFrame
- créer un Layout (BorderLayout, GridLayout, ...)
- associer ce gestionnaire de placement à la Frame en utilisant la méthode setLayout de la classe Container (une JFrame EST un Container)
- utiliser la méthode add appropriée pour rajouter les diérents éléments (JButton, JComboBox ...) à la JFrame
- dimensionner la JFrame (à l'aide des méthodes setSize(int,int) ou pack())
- la faire acher (à l'aide de setVisible(boolean)).

BorderLayout avec des composants JButton

Nous allons reproduire la JFrame suivante :

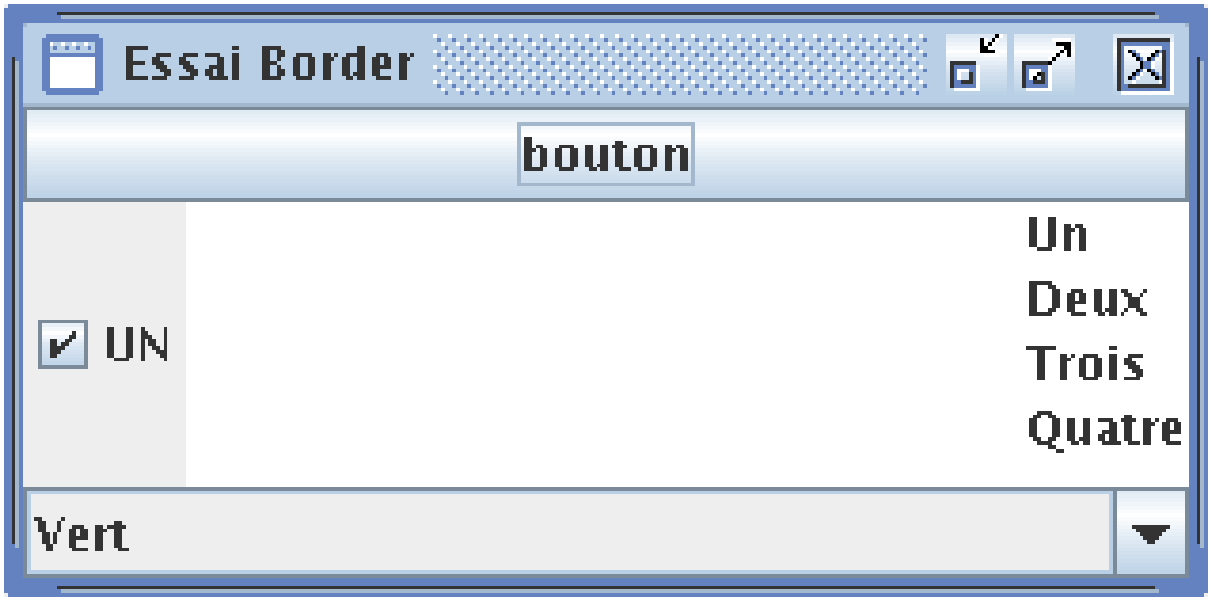
Figure 1.12. BorderLayout et JButton



BorderLayout avec d'autres composants

Nous allons reproduire la JFrame suivante :

Figure 1.13. BorderLayout et d'autres Components



Il faut intégrer à la place des boutons :

- un JComboBox avec trois item de choix : Vert, Rouge et Bleu.
- une JList avec quatre items de choix : Un, Deux, Trois , Quatre.

- une JCheckBox déjà coché
- un JTextArea de 5 lignes et 20 colonnes au centre

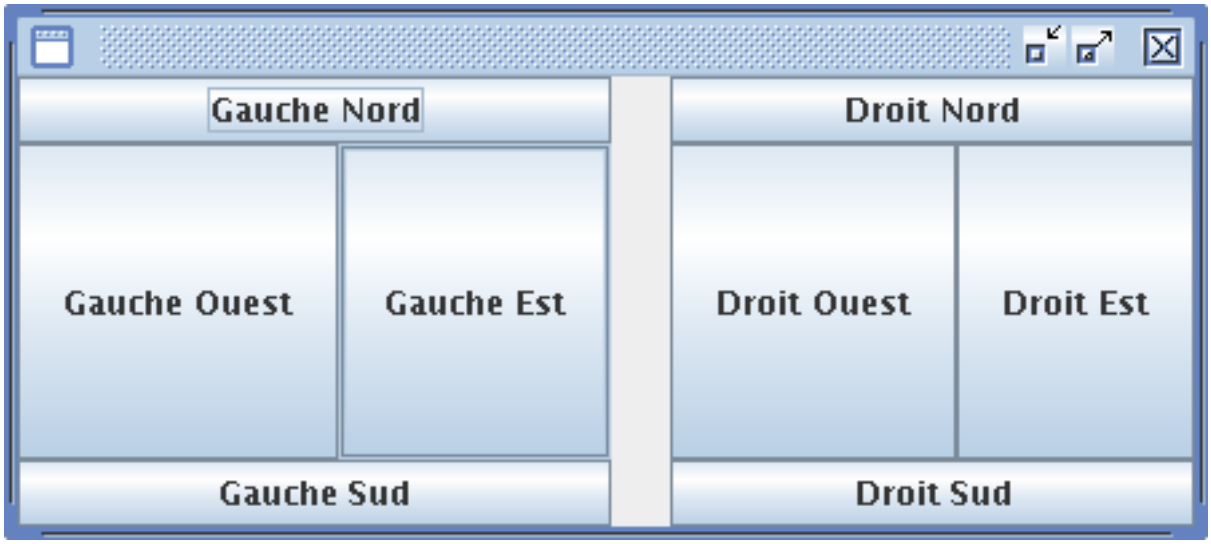
GridLayout

Reproduire la figure :

Figure 1.14. GridLayout

BorderLayout avec deux Panels

Figure 1.15. BorderLayout avec DEUX JPanels



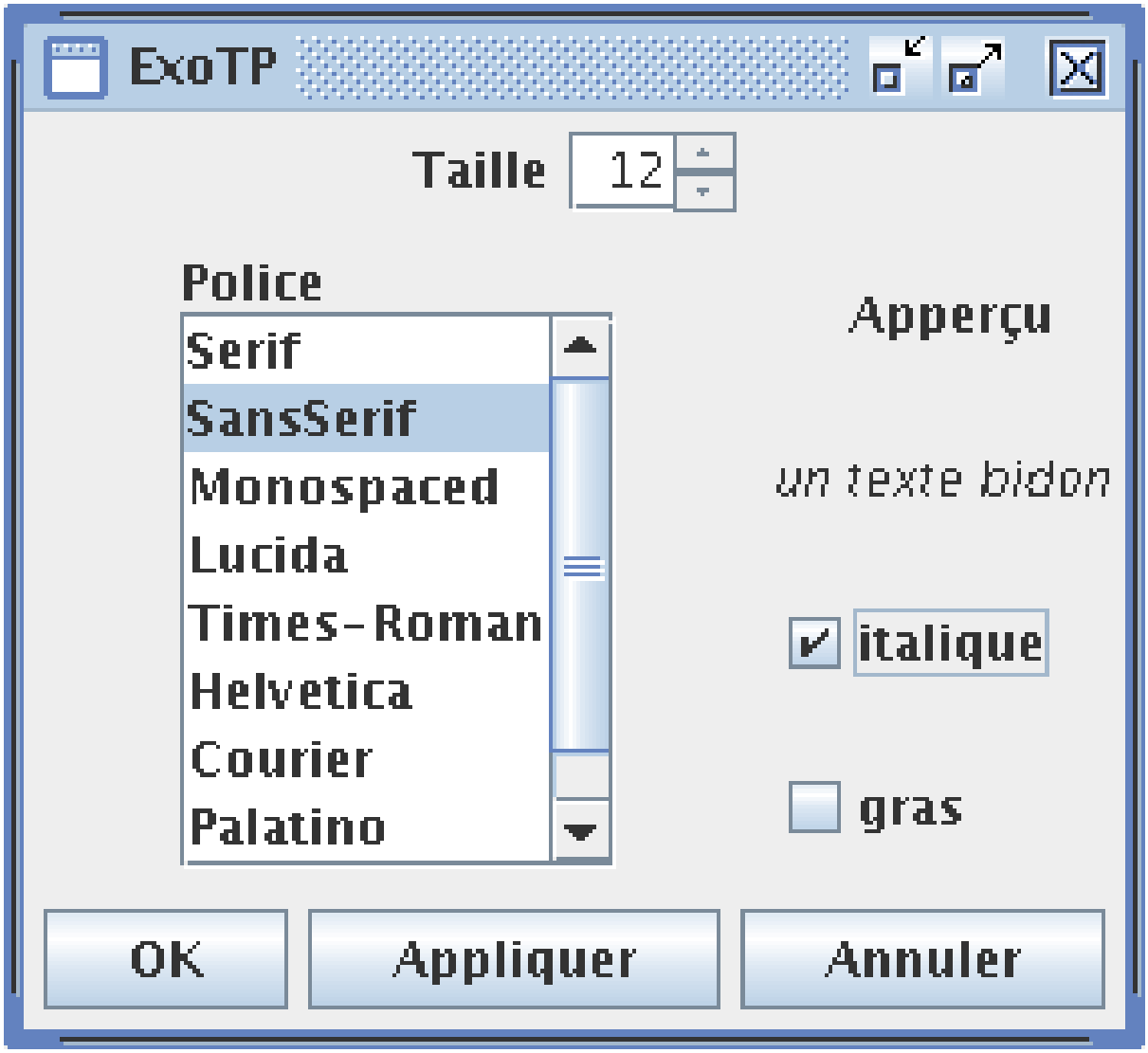
Reproduire la figure (dimension de la JFrame 650x200) :

- associer BorderLayout à la JFrame
- mettre un JPanel à l'est et un JPanel à l'ouest
- travailler chaque JPanel (ajouter les composant et le layout manager)
- faire afficher la JFrame

Exercice récapitulatif

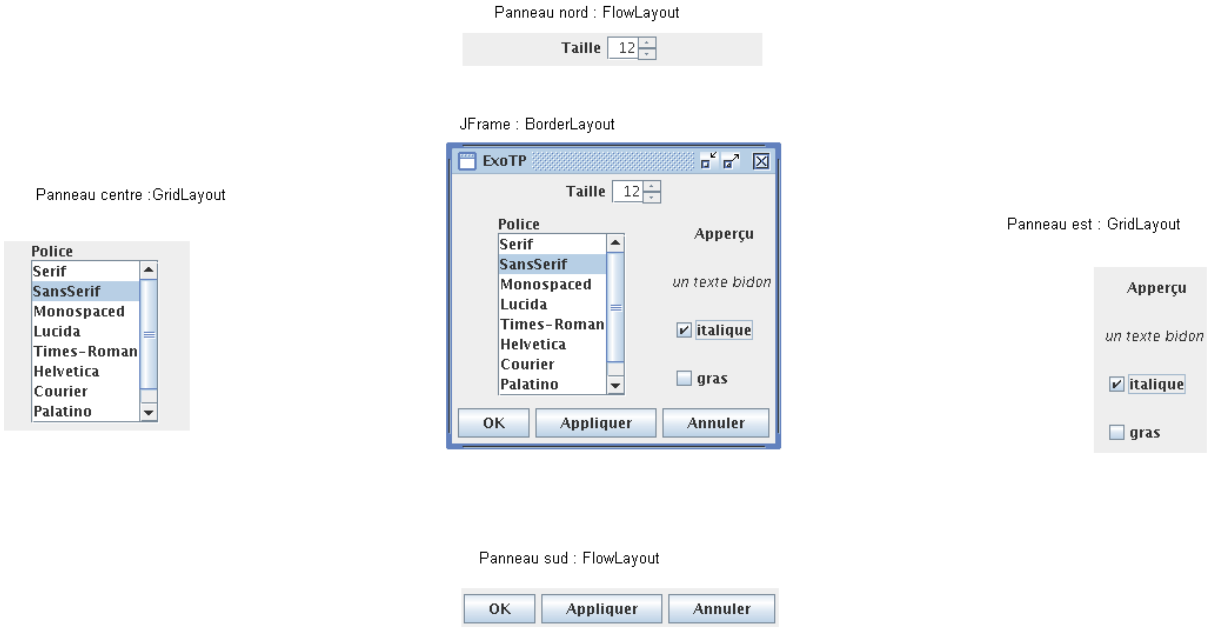
Reproduire à l'identique la JFrame donnée :

Figure 1.16. JFrame à reproduire



La décomposition adoptée pour réaliser la JFrame avec les layouts associés vous est donnée :

Figure 1.17. Découpage de la JFrame en Panels



Les ascenseurs sont obtenus avec des *JScrollPane*, les fontes avec *Font* et le compteur avec *JSPinner*.

La boîte de dialogue est obtenue en utilisant (*JOptionPane.showMessageDialog(...)*).

Evénements

Le but de ce TP est de faire de la gestion des événements avec *swing*.

Complément : les classes internes

Dans les versions antérieures à la version 1.1, toutes les classes étaient distinctes. Il est maintenant possible de dénir un classe interne à une autre. Sans aller dans le détail, les classes internes servent à décrire des relations fortes entre deux classes. En particulier, la création d'une instance de la classe englobante donne lieu à création d'une instance de la classe interne (si elle est non statique). Celle-ci a alors accès aux variables d'instance de la classe englobante. Cette particularité permet une gestion des événements simpliée. Syntaxe :

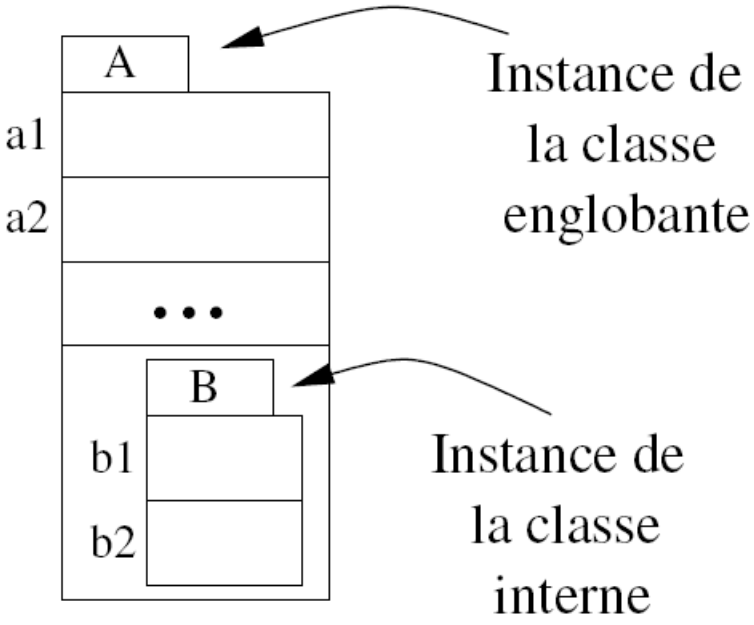
```
public class Englobante
{
class Interne
{
} // fin class Interne
}// fin classe Englobante
```

Idée d'utilisation :

```
public class Englobante // interface graphique
{ComposantAction c;
ComposantVue v; ...
c.addTrucListener(new Interne()); // dans une méthode
```

```
...
class Interne implements TrucListener // traitement
{
// on peut appliquer directement des méthodes sur v
}
}// fin classe Englobante
```

Figure 1.18. Dépendance entre instances



Architecture des classes

Le but est de gérer une action de l'utilisateur sur l'ihm (exemple : clic sur un bouton).

Base

La méthodologie est la suivante :

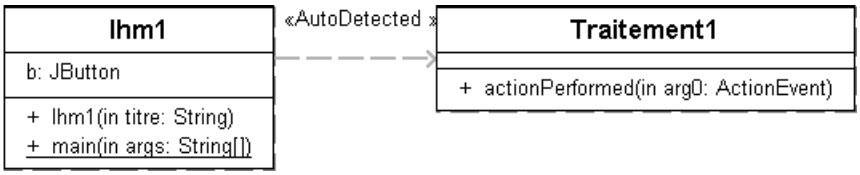
- Créer une classe *IHMn* qui crée l'ihm (par exemple, une *JFrame* et le positionnement de composants dans celle-ci)

- Créer une classe *Traitementn* qui permet de gérer un événement (par exemple : Clic sur un bouton) : cette classe implémente une interface *XXXListener* (*ActionListener* pour un bouton) en implantant la (ou les) méthode(s) de l'interface, (*actionPerformed(..)* dans le cas d'*ActionListener*) (consulter l'API).

- L'association entre le responsable d'un événement (par exemple, le bouton) et l'objet de la classe de traitement se fait à l'aide de la méthode *addXXXListener*.

Application de cette méthode dans un exemple : un clic sur le bouton génère un message "Ça marche" sur la sortie standard.

Figure 1.19. Base IHM/Traitement




```
import javax.swing.JButton;
import javax.swing.JFrame;

public class Ihm1 extends JFrame {

 /**
  * @param args
  */
 JButton b;
 public Ihm1(String titre)
 {
  super(titre);
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.b=new JButton(" Go !! ");
  this.add(b);
  Traitement1 tt = new Traitement1();
  this.b.addActionListener(tt);
  this.pack();
 }

 public static void main(String[] args) {
  // TODO Auto-generated method stub
  Ihm1 ihm1 = new Ihm1("click");
  ihm1.setVisible(true);
  }

}
```

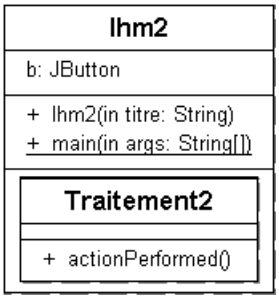
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Traitement1 implements ActionListener {


 public void actionPerformed(ActionEvent arg0) {
  System.out.println("Ca marche");

 }

}
```

Inconvénient de cette méthode ; comme les deux classes (*Ihm1* et *Traitement1*) sont séparées, il est très dicile d'agir (modifier un label, rendre des composants invisibles, ...) sur l'IHM depuis la classe de traitement.

Classe internes

Utilisation des classes *internes* pour pouvoir facilement modier l'IHM.

Figure 1.20. Classe interne




```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

public class Ihm2 extends JFrame {

 /**
  * @param args
  */
 JButton b;
 public Ihm2(String titre)
 {
  super(titre);
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.b=new JButton(" Go !! ");
  this.add(b);
  Traitement2 tt = new Traitement2();
  this.b.addActionListener(tt);
  this.pack();
 }

 public static void main(String[] args) {
  // TODO Auto-generated method stub
  Ihm2 ihm2 = new Ihm2("click");
  ihm2.setVisible(true);
 }

 // une classe interne de traitement
 public class Traitement2 implements ActionListener
 {
  public void actionPerformed(ActionEvent e) {
   b.setText("Ca marche");
  }
 }

}
```

Le "clic" sur le bouton change son contenu.

Délégation

Cette solution consiste à faire gérer les événements générée par l'utilisation de l'IHM par une classe déléguée. Cette solution peut simplier le développement d'interface plus complexe, en séparant plus clairement IHM et traitement applicatif des actions.

Figure 1.21. Délégation


```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

public class Ihm3 extends JFrame {

 /**
  * @param args
  */
 JButton b;
 public Ihm3(String titre)
 {
  super(titre);
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.b=new JButton(" Go !! ");
  this.add(b);
  Traitement3 tt = new Traitement3(this);
  this.b.addActionListener(tt);
  this.pack();
 }

 public static void main(String[] args) {
  // TODO Auto-generated method stub
  Ihm3 ihm3 = new Ihm3("click");
  ihm3.setVisible(true);
 }

}

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Traitement3 implements ActionListener {

 Ihm3 ihm;
 public Traitement3(Ihm3 ihm)
 {
 // on recupere un reference sur l'ihm
 this.ihm = ihm;
 }
 public void actionPerformed(ActionEvent e)
 {
 ihm.b.setText("Ca marche");
 }

}
```

Avantage : nous avons séparé le *traitement* de la *présentation* de l'IHM. Inconvénient : il faut passer une référence de l'IHM à la classe de traitement et laisser accès au variables d'instances depuis une autre classe (ou dénir des méthodes d'accès comme *public Button getButton()* dans l'IHM). Conclusion : la deuxième solution est à retenir dans ce tp, la dernière si vous voulez aller plus loin en java.

Tri des diérents Events

Si une application comporte des Components générant plusieurs Event de même nature (cas de plusieurs boutons, générant chacun un ActionEvent), on utilise la méthode getSource() pour connaître le responsable de l'événement.

Réalisations

Un bouton simple

Produire une *JFrame* avec un *JButton* qui permette de quitter l'application. Indications :

- il faut importer *javax.swing.** ET *java.awt.event.**
- il écrire une classe qui implémente *ActionListener* (et donc implante la méthode

  ```
  public void actionPerformed(ActionEvent e)
  ```

)
- la méthode *actionPerformed* ne fait que *System.exit(0)* ; (voir les APIs)
- Enfin, il faut penser à utiliser la méthode *addActionListener* pour associer un *Listener* au bouton : propagation de l'événement de la source vers la destination.

Deux boutons

Produire une *JFrame* avec deux *JButton* qui permettent, l'un de quitter, l'autre d'afficher un message sur la sortie standard. Ici la difficulté est de faire le tri entre les deux boutons. Indications :

- La mise en oeuvre est la même que pour seul bouton
- Dans la méthode *actionPerformed*, on peut savoir qui est le responsable de l'événement avec la méthode *getSource()* appliquée à un objet événement. Ensuite l'utilisation de tests d'égalités permet de définir l'identication du responsable.
- Les actions à réaliser sont très simples.

Actions plus complexes

Produire une JFrame avec un JButton qui permette de modier le texte d'un JLabel. Indications :

- le problème est du même niveau que le précédent. Il faut pouvoir récupérer dans la méthode de traitement de l'événement, en plus du(es) bouton(s), le Label.
- Ensuite, un simple appel à setText(String) permet de modier le label.

Quitter

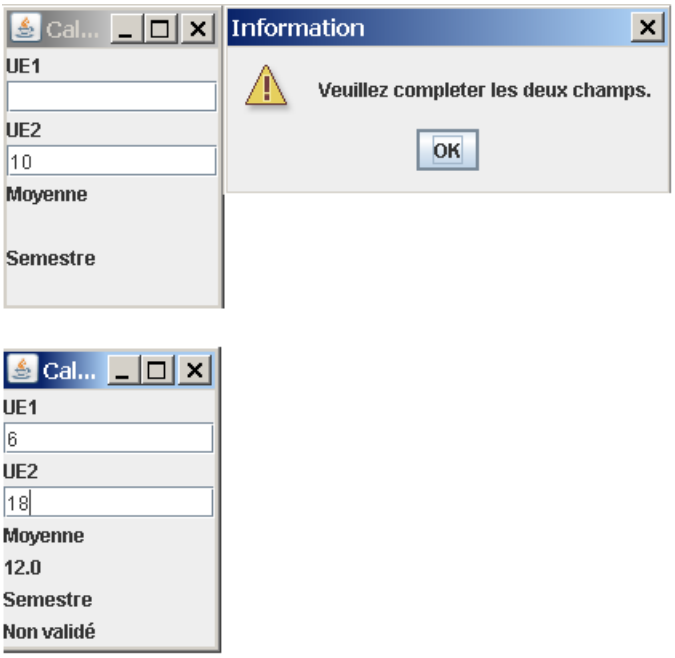
Associer l'action quitter au bouton prévu à cet eet dans la barre de titre de la fenêtre. Pour ça deux solutions :

- à l'ancienne : regarder WindowListener
- en utilisant la méthode setDefaultCloseOperation() de la classe JFrame (voir les constantes de cette classe pour le paramètre)

Exercice récapitulatif

Réaliser le calculateur de semestre suivant :

Figure 1.22. Semestre



Pour avoir le semestre, il faut que chaque UE soit supérieure ou égale à 8 et que la moyenne générale soit de 10. Si une des deux UE(JTextField) alors une fenêtre de dialogue doit être affichée (JOptionPane.showMessageDialog). Enfin, utiliser

```
JFrame frame;
```

comme attribut peut vous faciliter la vie dans la gestion de l'événement.

Chapitre 2. Programmation d'un "shoot them up"

Créer un nouveau projet java nommé *jeux*, dans le répertoire `bin` du projet créer un répertoire `ressource`.

Gérer des ressources

Notre jeux va contenir deux type de ressources des images GIF(Graphics Interchange Format) et des son WAV(WAVEform audio format). Les jeux se doivent d'être rapide, nous allons utiliser de la mémoire pour gagner du CPU(Central Processing Unit) en fabriquant un gestionnaire de ressource. Avant de gérer nos ressources nous allons apprendre à les utiliser.

Affichage d'une image

En java un composant (java.awt.Component ou javax.swing.JComponent) est un objet qui a une représentation graphique et qui peut interagir avec l'utilisateur. Chaque composant est doté d'une méthode *paint(Graphics g)* qui est utilisée pour lui donner son aspect visuel. Cette méthode est appelée automatiquement par le système graphique. Il est possible, pour n'importe quel *Component* (en le sous classant) de redéfinir cette méthode. Néanmoins un des Component se prête bien à cet usage : *JPanel*. Sous classer *JPanel* et redéfinir la méthode *paint* permet donc de donner l'aspect voulu à la zone. La méthode *paint* prend en paramètre un objet *Graphics*. *Graphics* est une classe (*abstraite*) qui déni des méthodes pour dessiner des lignes, afficher des images, ... Il n'est pas possible de créer directement un objet *Graphics*(car la classe est abstraite). Depuis la version 1.2 de Java, il est possible de transtyper l'objet *Graphics* en un objet *Graphics2*, qui donne accès à une API plus riche.

```
public void paint(Graphics g)
{ Graphics2D g2 = (Graphics2D) g ;
g2. ... }
```

Nous allons illustrer notre propos en créant un cercle de couleur rouge de diamètre 200 pixels placé en 0,0 dans une fenêtre de 200 pixels par 200 pixels.

Créer un paquetage *tests* qui contiendra notre première classe *TestCercle*.

1. Créer la classe TestCercle en choisissant de créer un main et d'avoir pour super classe JFrame
2. En utilisant source -> override implement méthode, choisir paint de JComponent
3. Dans le main créer un objet de type TestCercle, le dimensionner (setSize(...)) et le rendre visible(setVisible(...))
4. Dans la méthode paint(...) donner une couleur à g2 (setColor) et dessiner un oval(drawOval(...))
5. Tester vous devez avoir un cercle rouge dessiné dans la JFrame.

Le fait d'utiliser une JFrame nous empêche par la suite de transformer notre code en une JApplet, nous allons donc créer un JPanel que nous inclurons dans la JFrame. Au passage, vous avez pu constater que le cercle n'est pas bien positionné car la taille de la JFrame n'est pas la taille du ContentPain (La zone d'affichage).

1. Créer une classe TestCercleJPanel qui hérite de JPanel, qui contient la méthode paint(...) et constructeur sans paramètres qui défini la taille(setSize) et la taille préférée (setSize).
2. Créer une classe TestCercleFrame qui contient un main, dans ce main un objet de type TestCercleJPanel est créé ainsi qu'un objet de type TestCercleFrame

   L'objet de type TestCercleFrame a pour Layout le layout par défaut, il contient l'objet TestCercleJPanel(add(...)) et est dimensionné en fonction de la taille préférée de ses composants(pack()). Bien entendu l'objet de type TestCercleFrame doit être visible.

Nous n'avons toujours pas afficher notre GIF, pour afficher notre GIF nous allons utiliser la méthode drawImage de Graphics2D et placer un monstre au centre de notre cercle. L'image sera chargée en utilisant ImageIO.read(url). Pour pouvoir déplacer notre projet nous utiliserons getResource(nom) qui nous permet d'obtenir une ressource à partir de son nom.

1. Dans paint déclarer une URL (java.net.URL) qui aura pour valeur this.getClass().getClassLoader().getResource("ressources/monstre.gif").

   getClass permet d'obtenir la classe courante, getClassLoader d'obtenir le classLoader et enfin getRessource permet d'obtenir la ressource, cette solution offre pour avantage de permettre d'utiliser des ressources présentes dans un jar.

2. Déclarer une BufferedImage créer avec ImageIO.read(...) . Il vous faudra gérer les exceptions de type IOException.

3. Afficher l'image avec la méthode drawImage() de graphics2D.

4. Tester

Pour finir ajouter un texte avec

Pour en savoir plus vous pouvez consulter le tutorial de Sun : `http://java.sun.com/products/jfc/tsc/articles/painting/index.html`

Jouer un son

Java offre deux approche pour jouer des sons :

javax.sound Une API qui permet de faire du traitement du signal

AudioClip une classe de java.applet qui nous permet de jouer un son.

Nous allons utiliser la seconde solution car correspondant à nos attentes.

1. Toujours dans paint déclarer un objet AudioClip initialisé en utilisant JApplet.newAudioClip(...)

2. Le jouer en utilisant play() ou loop()

3. Tester

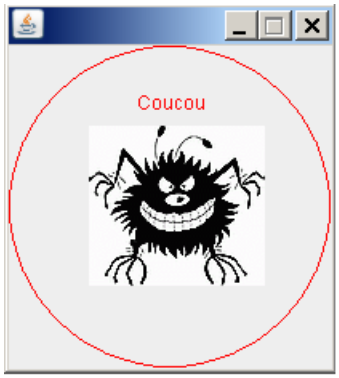
Nous pouvons améliorer la lecture de notre son en le lançant dans un nouveau thread. Le thread est créé avec la classe interne suivante :

```
class PlayAudioClip extends Thread
 {
  AudioClip audioClip;

  public PlayAudioClip(String s) {
   URL url = this.getClass().getClassLoader().getResource("ressources/"+s);
   audioClip = JApplet.newAudioClip(url);
  }

  public void run() {
   audioClip.play();

  }

 }
```

Modifier votre code pour utiliser la classe interne.

Au final vous devez avoir ceci :

Figure 2.1. TestJFrame



Construction du gestionnaire des ressources

Un jeux fait continuellement appel à des ressources, aussi vaut-il mieux les stocker en mémoire que de les lire à chaque utilisation. Le stockage ne peut-être fait à l'initialisation, de quelles ressources à t-on besoin, quel délai va induire le chargement. Une solution est de charger les ressources à la demande dans une structure de donnée ; nous utiliserons une HashTable.

Le gestionnaire de ressource que nous allons créer sera abstrait et devra être sous-classé pour gérer des sons et des images.

RessourceCache

Commençons par créer un paquetage mon_jeux. Dans ce paquetage nous allons créer notre gestionnaire de ressources(RessourceCache) qui aura deux classes filles SpriteCache et SoundCache.

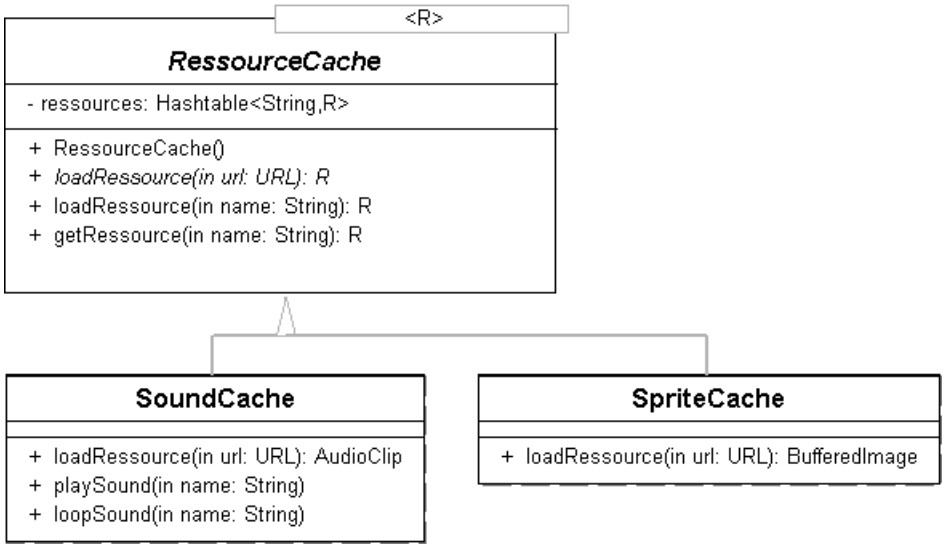
Le code RessourceCache est le suivant :

```
package mon_jeux;

import java.net.URL;
import java.util.Hashtable;

public abstract class RessourceCache<R> {
 private Hashtable<String,R> ressources;

 public RessourceCache() {
  ressources = new Hashtable<String,R>();
```

```
 }

 public abstract R loadRessource(URL url);

 public R loadRessource(String name)
 {
  URL url = null;
  url = this.getClass().getClassLoader().getResource(name);
  return this.loadRessource(url);
 }

 public R getRessource(String name)
 {
  R r = ressources.get(name);
  if (r == null)
  {
   r = loadRessource("ressources/"+name);
   ressources.put(name, r);
  }
  return r;

 }
}
```

Notre classe est abstraite car la méthode public abstract R loadRessource(URL url) est abstraite, elle n'a pas de code associée, un objet de type RessourceCache ne peut être instancié. Il nous faut donc créer deux classes filles qui redéfiniront la méthode abstraite. RessourceCache est générique, elle prend en paramètre le type de la ressource.

Figure 2.2. Gestionnaire de ressources



SpriteCache

Implémenter la classe SpriteCache qui hérite de RessourceCache<BufferedImage>. Vous n'avez que la méthode BufferedImage loadRessource(URL url) à redéfinir.*Il est encore possible de gagner en rapidité en utilisant des images compatibles.*

SoundCache

Implémenter la classe SoundCache qui hérite RessourceCache<AudioClip>. Vous devez :

- redéfinir la méthode public AudioClip loadRessource(URL url)
- implémenter la méthode public void playSound(final String name) qui joue un son unique
- implémenter la méthode public void loopSound(final String name) qui joue un son en boucle

Si vous ne souhaitez pas utiliser de classe interne vous pouvez créer votre thread comme suit : Thread t = new Thread(new Runnable() { public void run() { truc à faire } });

Définir un niveau (Level)

Nous allons définir une classe Level qui représente un niveau, c'est classe sera instanciée par notre jeux Game et sera utilisée dans le constructeur des entitées de notre jeux (Entity) :

Définir les acteurs (Entity)

Les monstres

Le joueur

Les tirs

Définir le jeux

A vous de continuer