M2.23.5 Programmation

Travaux dirigés et travaux pratiques d'algorithmique du deuxième semestre

Jean-François Remm
Jean-François Berdjugin
Vincent Lestideau

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

par Jean-François Remm, Jean-François Berdjugin, et Vincent Lestideau

Table des matières

Liste des illustrations

Présentation et objectifs

Pour réaliser ces TP et ces TD, vous disposez de 6 séances de trois heures de TP et de 3 séances d'une heure et demie de TD. 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/`. En TD, nous allons apprendre comme mettre en relation entre des classes ces relations seront nommées des *associations*. Vous avez déjà utilisé ce concept (l'association) dans la classe CompteBancaire en utilisant la classe Personne et de même pour la classe Personne vous avez utilisé la classe Date et la classe String. Nous avons mis en relation quatre classes. L'association n'est pas le seul moyen de factoriser du code, il existe aussi la notion d'*héritage* et des notions corrélées : les *classes abstraites*, les *interfaces* et le *polymorphisme*. En TP nous aurons deux grandes parties :

- Introduction à la programmation, une reprise des TD complétée par la découverte de *SWING* une bibliothèque graphique et de la notion de *programmation événementielle.*
- Programmation d'un jeux de type "shoot them up" à finir en devoir maison.

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. Travaux dirigés

Nous disposons de 3*1,5h de TD, pendant ces trois séances nous allons apprendre comment mettre en relation des classes et comment factoriser du code grâce à la notion d'héritage.

Avertissement

La notion UML (Unified Modeling Language) n'a rien avoir avec la méthode *Merise* vue en SI (Systèmes d'Information).

Nous allons illustrer la majorité de nos propos en utilisant un diagramme statique de la notation UML : le *diagramme de classes*. Ce TD n'est pas un TD d'UML mais l'utilisation d'une notation standard et son aspect graphique va nous permettre de mieux communiquer et de nous abstraire de la syntaxe *Java*.

1. Classes/Association Les bases

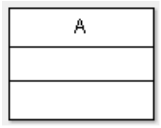
Nous allons étudier comment associer simplement des classes, nous utiliserons la notation UML avec à chaque fois le code Java correspondant.

1.1. Cours et exemples

1.1.1. La classe

Commençons par le plus simple : la classe vide.

Figure 1.1. Classe vide



Sa traduction java est :

```
public class A {}
```

Une classe vide ne sert à rien. Une classe pour être utile doit être peuplée d'attributs (variables d'instance ou variables de classe) et de méthodes (constructeurs, méthodes d'instance, méthodes de classe). Le méthodes et les variables ont une accessibilité :

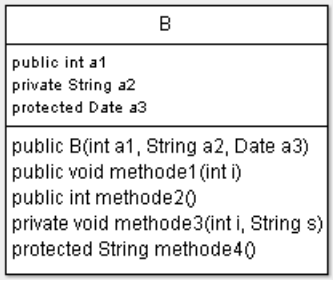
public
: accessible depuis n'importe où,

private
: accessible seulement dans la classe,

protected
: accessible depuis les classes filles et/ou dans le même package.

C'est vision est simplifiée par rapport à java qui est complexifié par la notion de paquetage (package).

Figure 1.2. Classe avec trois compartiments

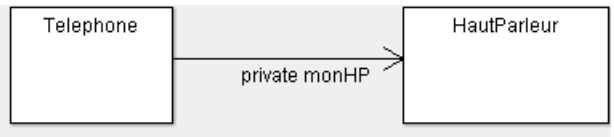



```
package td.exemples;

import java.util.Date;

public class B {
 public int a1;
 private String a2;
 protected Date a3;

 public B(int a1, String a2, Date a3) {

 }

 public void methode1(int i)
 {

 }

 public int methode2()
 {
  return 0;
 }

 private void methode3(int i, String s)
 {

 }

 protected String methode4()
 {
  return "";
 }
}
```

1.1.2. L'association

L'association permet d'exprimer une relation entre deux classes. Elle exprime une connexion sémantique entre deux classes (relation logique). Elle est représentée par un trait plein qui peut-être complété par des flèches. Les flèches définissent la visibilité.

Figure 1.3. Association simple




```
public class Telephone {
 private HautParleur monHP;
}
```

Sur le diagramme précédent la classe Telephone "voit" la classe HautParleur. Nous avons associé à un téléphone, un haut parleur. Mais comment faire pour associer à un téléphone 15 ou n touches ? En utilisant des multiplicités.

1.1.3. Les multiplicités

Les multiplicités sont comparables aux cardinalités du système Merise (mais sont placées de l'autre coté), elles servent à indiquer le nombre minimum et maximum d'instances de chaque classe dans la relation liant 2 ou plusieurs classes :

Digit
: Le nombre exacte qui peut être implémenté sous forme de tableau.

* ou 0..*
: de zéro a plusieurs qui peut être implémenté sous la forme d'une collection souvent un vecteur.

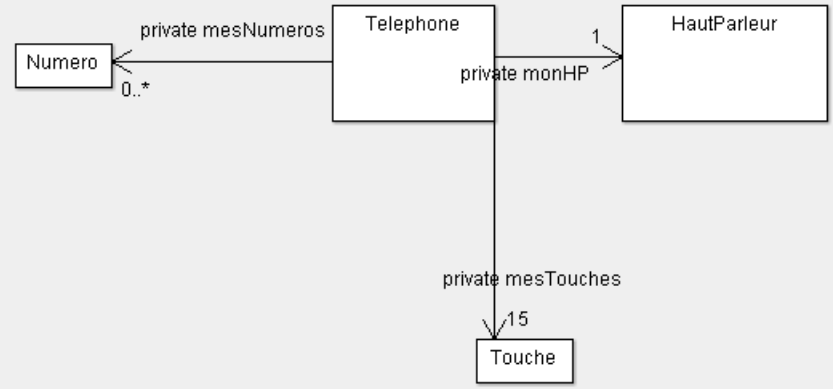
0..1
: zéro ou un, le zéro est implémenté en utilisant une référence nulle.

1..*
: un à plusieurs.

n..m
: de n à m.

[1]

Figure 1.4. Association avec multiplicité

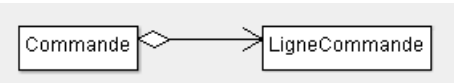



```
public class Telephone {
 private HautParleur monHP;
 private Touche[] mesTouches;
 private Vector<Numero> mesNumeros;
}
```

1.1.4. L'agrégation

Il existe des cas particuliers d'association : les agrégations et les compositions. L'agréation traduit la notion de tout-partie ou est une partie de. Elle est soumise à des restrictions une instance de classe ne peut se composer elle même et il ne peut-y avoir de cycle dans les dépendances.

Figure 1.5. Agrégation



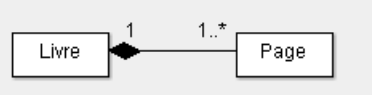
La traduction java reste la même que pour l'association.

[1] Il aussi possible d'exprimer des plages plus complettes comme 1, 2..10, 100..*.

1.1.5. La composition

La composition est une forme de particulière de l'agréation ou le tout est responsable du cycle de vie de la partie. La partie doit-être créée (*new*) et détruite dans le tout. De plus une partie ne peut faire partie de plusieurs tous.

Figure 1.6. Composition




```
package td.exemples;

import java.util.Vector;

public class Livre {
 private Vector<Page> pages;
 public Livre()
 {
  pages = new Vector<Page>();
  Page page = new Page();
  pages.add(page);
 }
}
```

Dans l'exemple précédent, la page est crée dans le Livre, ainsi sa vie est liée à celle du livre.

1.2. Exercices

Nous allons dans un premier temps réfléchir sur un diagramme de classe puis dans un deuxième temps introduire l'exemple qui sera vu en TP.

1.2.1. Diagramme de classe d'une maison

Une maison est composée de pièces, elles mêmes composées de murs. Les pièces contiennent ou ne contiennent pas des objets. La disparition de la maison entraîne celle des pièces.

1.2.2. La voiture

Une voiture est caractérisée par sa marque et son modèle, elle est possédée par une personne qui a un nom. La voiture possède un moteur qui ne peut-être réutilisé après la vie de la voiture. Le moteur est caractérisé par sa puissance et sa marque. Proposer un diagramme de classe et le code java associé.

2. Classes/Association Le détail

UML dispose d'un mécanisme d'extension : les stéréotypes. Ils se présentent sous forme de guillemets ou d'accolade, ils permettent d'étendre la sémantique des éléments de modélisation. Nous verrons des stéréotypes d'association qui augmente le sens de l'association pour faciliter le codage .

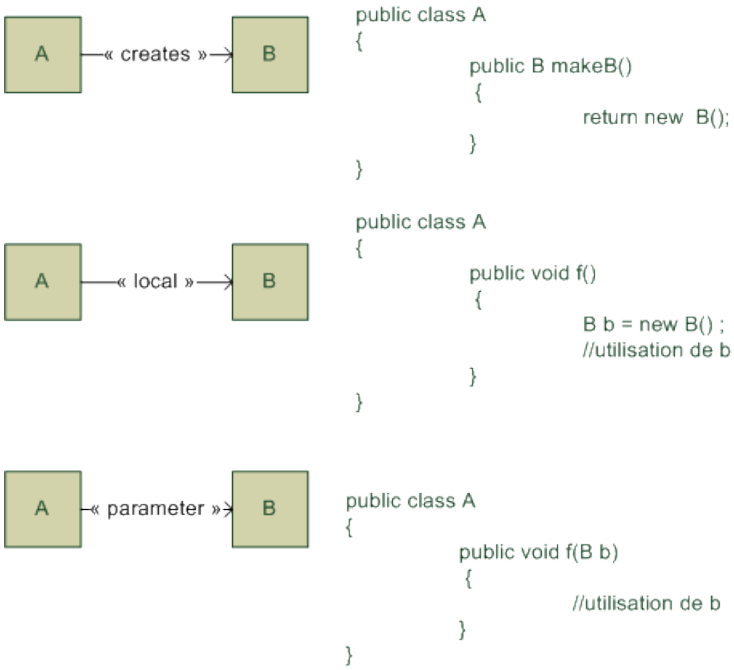
Nous verrons aussi la possibilité d'imbriquer une classe dans une classe interne. L'imbrication nous rendra de grands services lors de la réalisation d'IHM (Interface Homme-machine).

2.1. Cours et exemples

2.1.1. Les stéréotypes d'association

Nous allons au travers de quatre stéréotypes d'association étudier du code Java.

Figure 1.7. Stéréotypes d'association



2.1.1.1. create

Le stéréotype "create" indique que la source va créer la cible est la rendre disponible pour le reste de l'application.

2.1.1.2. local

Le stéréotype "local" indique que la source va créer une instance de la cible, conservée dans une variable locale et non un attribut.

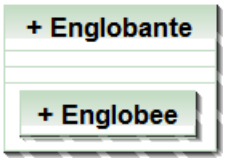
2.1.1.3. parameter

Le stéréotype "parameter" indique que la source accède à la cible via une référence reçue en paramètre

2.1.2. Les classes internes

Il est possible de déclarer une classe dans une autre classe (ou une interface)[2]. Une classe interne a un accès complet sur tous les attributs des ses classes englobantes. Les classes internes nous seront utiles pour traiter les événements. En UML les classes internes sont représentées par un cercle avec une croix mais on peut aussi les trouver simplement imbriquées.

Figure 1.8. Classes imbriquées




```
public class Englobante {
 public class Englobee {

 }
}
```

[2]Java est très riche, il permet des classes internes statique de classe, des classes internes de méthodes, des Innerclass de classe celle que nous étudierons et des classes anonyme de méthode.

Il existe d'autre possibilités pour qualifier les associations comme les classes d'association et le qualifieurs d'association.

2.2. Exercices

2.2.1. stéréotype d'association

Pour chaque problème suivant, proposer un diagramme de classe UML qui utilise un des stéréotypes précédents ?

- Une fabrique de voiture fabrique des voitures.
- Pour cuire un civet on utilise une casserole.
- Pour chauffer une maison, on fait un feu.

2.2.2. Classe interne

Les classes internes n'ont de réel intérêt que pour rendre concret un type abstrait : une interface ou une classe abstraite en java. Comme nous ne les connaissons pas encore l'exemple suivant est un peu tiré par les cheveux.

Nous souhaitons créer une voiture avec une pile. La voiture dispose d'une pile. Si la pile est vide, la voiture s'arrête. La voiture dispose de la méthode roule(int n) qui consomme des unités de pile. La seule classe UML visible est Voiture. Proposer une modélisation UML et le code java associé.

3. Héritage - classes abstraites - interface - polymorphisme

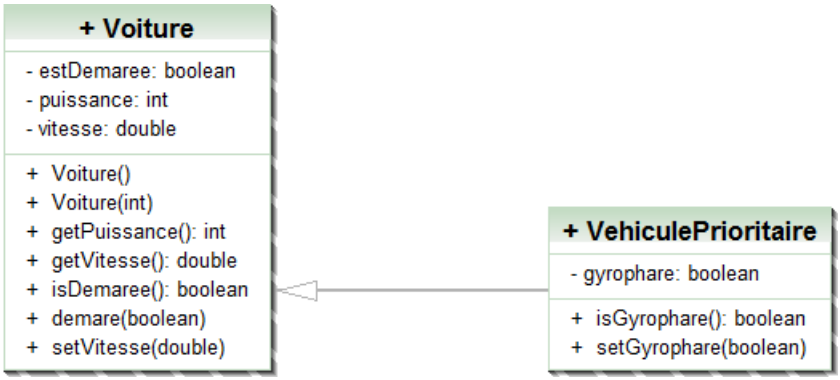
L'héritage est un moyen de réutiliser du code, à l'image d'un arbre généalogique, une classe fille hérite des caractéristiques de sa classe mère. L'héritage va nous permettre d'utiliser des classes abstraites : des classes dont une partie du code n'est pas encore connu mais dont la signature des méthodes est connue. Le classes abstraites ont une forme extrême, des classes sans aucun code et avec uniquement des méthodes statiques : les interfaces. L'héritage va aussi nous ouvrir la voie du polymorphisme, la capacité que possède un objet à changer de type à l'exécution.

3.1. Cours et exemples

3.1.1. Héritage

L'héritage est représenté en UML avec une flèche triangulaire. La classe cible est la classe mère, l'autre est la classe fille. On parle encore de superclasse pour la classe mère, et de sous-classe pour la classe fille. L'héritage peut-être traduit par "est une sorte de". La classe fille "est une sorte de" classe mère. Sur l'exemple qui suit la classe mère est Voiture et la classe fille est VehiculePrioritaire. En java, il n'y a pas d'héritage multiple une classe fille n'a qu'une classe mère. Une classe mère par contre peut-avoir plusieurs classes filles.

Figure 1.9. Héritage



La notion d'héritage est traduite en java par le mot clef extends.

```
public class VehiculePrioritaire extends Voiture {
 private boolean gyrophare;

 public boolean isGyrophare() {
  return gyrophare;
 }

 public void setGyrophare(boolean gyrophare) {
  this.gyrophare = gyrophare;
 }
}
```

L'héritage permet d'enrichir, la classe VehiculePrioritaire possède les méthodes et les attributs (public ou protected) de la classe Vehicule plus ses propres attributs.

L'héritage permet aussi de spécialiser, de *redéfinir* le comportement de la classe mère en *redéfinissant* ses méthodes. En java, la classe Object est la mère de toute classe, elle possède la méthode

```
public String toString()
```

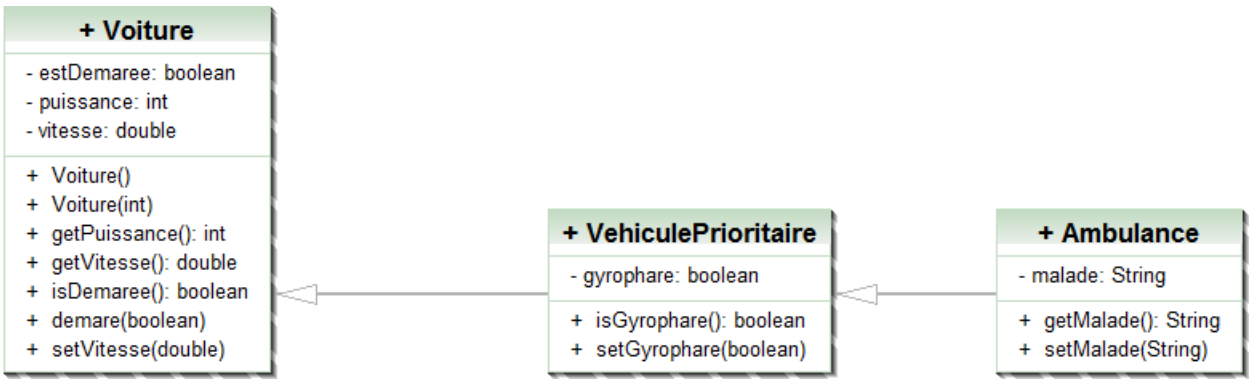
qui affiche le nom de la classe et la référence mémoire, vous avez déjà redéfini cette méthode pour obtenir un affichage plus convivial.

Enfin l'héritage permet de réutiliser du code sans en avoir la source (si la classe n'est pas final), pour hériter de Voiture, seul son *.class* est utile, le *.java* ne l'est pas.

Lors de la construction d'un objet de type d'une classe fille un constructeur de la classe mère est appelé en java le mot clef *super* à la même sens que *this* dans la classe fille. super(...) permet de faire un appel explicite à un constructeur de la classe mère, super.attribut ou super.methode() permet d'utiliser un attribut ou une méthode de la classe mère.

Par défaut, *super()* est utilisé, c'est donc le constructeur par défaut si il existe ou alors le constructeur sans paramètre de la classe mère qui est appelé.

Figure 1.10. Chaînage des constructeurs



Si le code de Voiture() est

```
public Voiture() {
  this(5);
 }
```

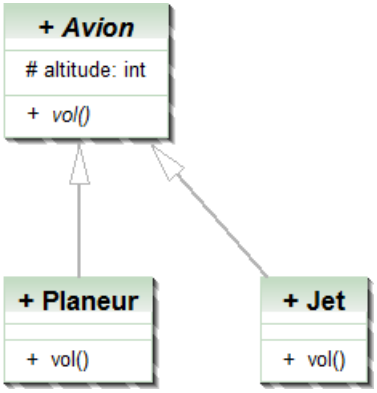
new Ambulance() conduira à la création d'une ambulance de 5 chevaux. Ambulance() est le constructeur par défaut, il existe car il n'y a pas d'autre constructeur dans Ambulance. Ambulance() fait appel implicitement (super()) à VehiculePrioritaire() qui lui même fait appel au constructeur sans paramètre Voiture(). Voiture() n'est pas un constructeur par défaut et il fait appel (this(5)) à une surcharge Voiture(int).

3.1.2. Classe abstraite

Une classe est dite abstraite si l'on ne connaît pas le code d'au moins une de ses méthodes. Attention une méthode sans code n'est pas une méthode dont le code est vide. maMethode(); n'a pas de code, maMethode(){ } est une méthode dont le code est vide. Une classe abstraite ne peut-être instanciée, alors à quoi sert-elle ? Elle permet de définir un comportement pour toutes ses classes filles sans en connaître le code. Toutes les classes filles sous peine de rester abstraite doivent implémenter la méthode abstraite.

En UML une classe abstraite et ses méthodes abstraites sont signalées par la propriété {abstract} est ou par un nom en italique.

Figure 1.11. Classe Abstraite



Le code java associé est le suivant :

```
public abstract class Avion {
 protected int altitude;

 public abstract void vol() ;

}
```

```
ublic class Planeur extends Avion {

 @Override
 public void vol() {
  // TODO Auto-generated method stub
  altitude +=100;
  if (altitude > 1000) altitude=1000;
 }

}
```

```
public class Jet extends Avion {

 @Override
 public void vol() {
  // TODO Auto-generated method stub
  if (altitude > 1000) altitude=10000;
 }

}
```

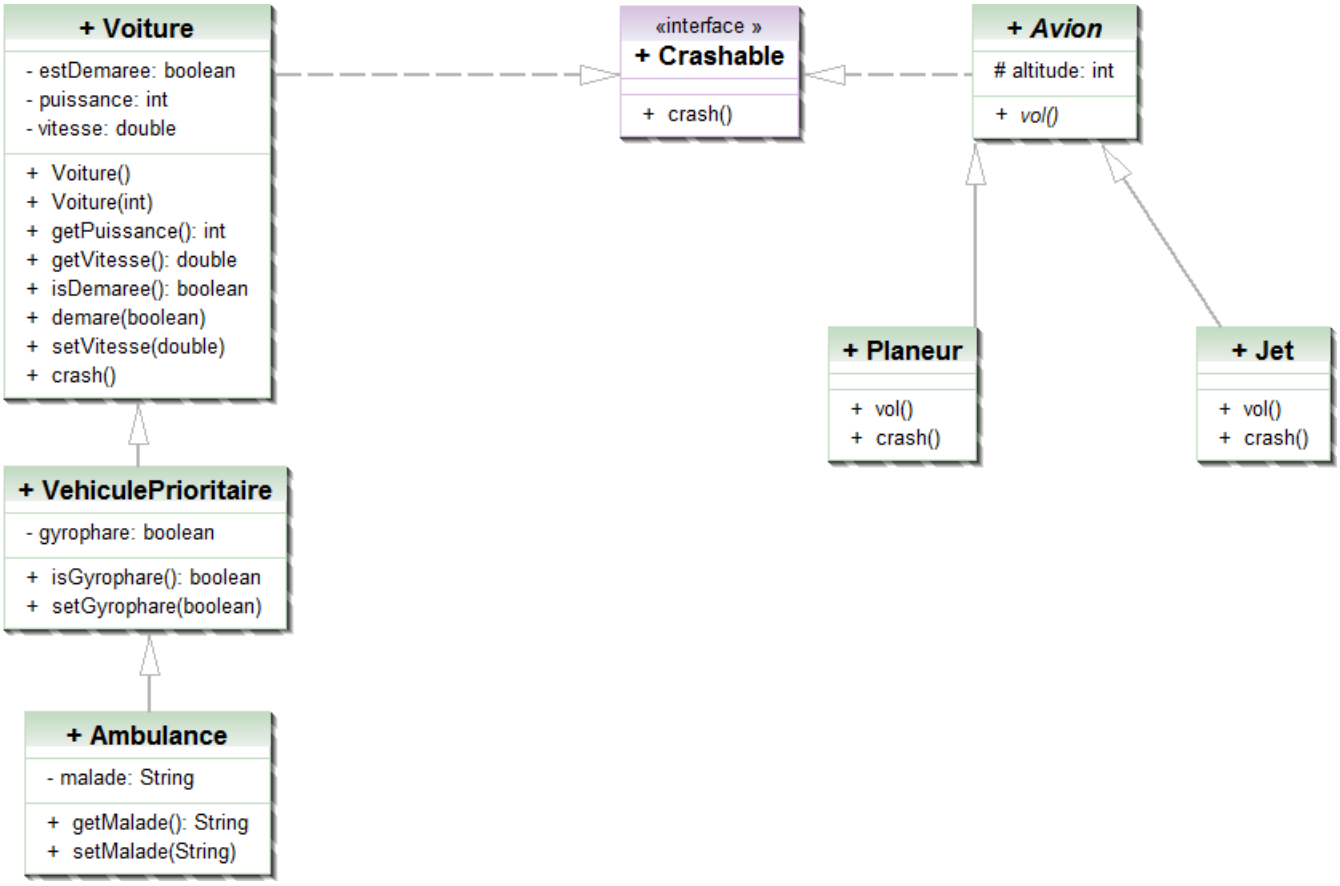
Tous les avions vols mais on ne sait pas comment. Par contre, on sait comment un Jet ou un Planeur volent.

3.1.3. Interface

Une interface est une classe abstraite pure. Elles sont très utiles en java qui ne possède pas d'héritage multiple mais qui permet d'implémenter plusieurs interfaces. Vous avez déjà rencontré l'interface comparable<T> qui contient la méthode int compareTo(T o). Pour rendre des objets comparables, il suffit d'implémenter l'interface comparable

à savoir donner un code à la méthode compareTo(T o). En UML une interface est un des deux stéréotypes de classe, l'autre est "utility" : une classe utilitaire est une classe qui ne contient que des attributs et des méthodes statiques. L'interface est représentée soit avec l'annotation "interface" soit avec une "lolipop". Le fait de réaliser (implements) une interface est noté avec un flèche en pointillés à tête triangulaire.

Figure 1.12. Interface



Le code java associé est

```
public interface Crashable {
 public void crash();
}
```

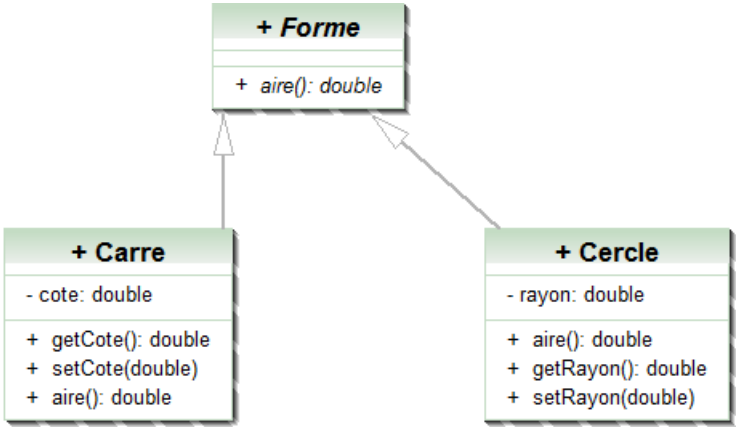
Voiture a du être modifier pour réaliser l'interface Crashable comme suit :

```
public class Voiture implements Crashable{
 ...

 @Override
 public void crash() {
  // TODO Auto-generated method stub
  ...
 }
 ...

}
```

3.1.4. Polymorphisme

L'héritage est un moyen de réaliser le polymorphisme en java. Le polymorphisme est le moyen pour un objet de changer de type à l'exécution. L'idée est d'autoriser le même code à être utilisé avec différents types. Vous l'avait fait en utilisant Collections.sort(). Soit l'exemple suivant :

Figure 1.13. Polymorphisme



la méthode de Forme aire() est abstraite, elle est implémentée dans Carre et dans Cercle de deux manière différente. Nous pouvons calculer l'aire totale de plusieurs formes comme suit :

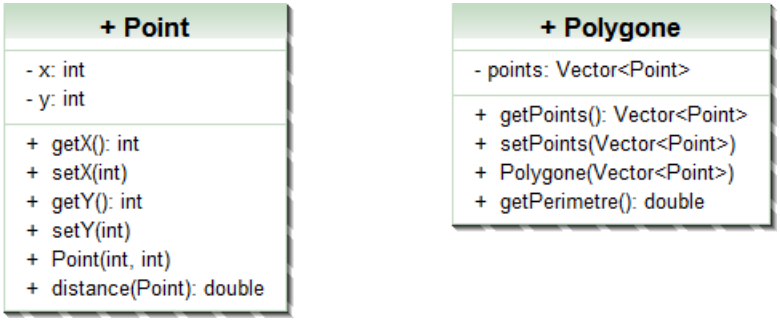
```
float aireTotal(Forme[] tabl) {
   float s=0;
   for(int i = 0; i < tab1.length; i++) {
      s += tabl[i].aire(); // le programme sait automatiquement quelle fonction appeler
   }
   return s;
}

// ...
Carre c1 = new Carre(); c1.setCote(10);
Carre c2 = new Carre(); c2.setCote(20);
Cercle cer1 = new Cercle(); cer1.setRayon(1);
Forme[] tableau = { c1, c2, cer1};
int a = aireTotal(tableau);
```

La méthode aireTotal() attend un tableau de Forme et exécute le même algorithme quelque soit la Forme. Nous avons donc un algorithme qui peut s'appliquer à n'importe qu'elle sorte de Forme. Il nous fout donc pouvoir transformer tout objet d'un classe fille de Forme en un objet de type Forme (upcating ou surclassement). A une référence d'un type donné, soit Forme, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous classe directe ou indirecte de Forme. Forme f1 = new Carre() est possible, de même que Forme f2 = new Cercle(). f1 et f2 ne dispose que des méthode de Forme mais peuvent retrouver leur type d'origine. f1= (Carre) f1 et f2=(Cercle) f2 on parle de downcasting. Le downcasting permet de liberer les fonctionnalités cachées.

3.2. Exercices

Nous allons partir de la classepoint et de la classe polynome, c'est cette dernière que nous allons faire évoluer.

Figure 1.14. Le polygone



3.2.1. Héritage

Proposer un diagramme de classe qui contient la classe Carre, la classe Rectangle, la classe Triangle et la classe Hexagone. La méthode double perimetre() doit être optimisée dans carré et dans rectangle, comme se nomme se procédé. Nous souhaitons que Carre possède deux constructeurs quel principe est utilisé ?.

3.2.2. Classe abstraite

Nous souhaitons savoir si une figure a un centre de symétrie (boolean isSymetrie()), mais nous ne savons pas le calculer pour un "Polygone" comment faire ?

3.2.3. Interface

Nous souhaitons trier des polygones, celui qui a le plus de ségments est le plus grand. Nous diposons de l'interface comparable (I*nterface Comparable<T>*) qui ne contient qu'une méthode *int compareTo(T o)* et de la méthode de classe Collection.sort() qui permet de trier une collection (public static <T extends Comparable<? super T>> void sort(List<T> list)[3]). Comment faire pour trier un un "*Carre*", un "*Rectangle*" et un "*Triangle*".?

3.2.4. Polymorphisme

Écrire une méthode double *perimetreTotal(Vector<Polygone> figures)* qui calcul la somme des périmètres d'un vecteur de "*Polygone*". Écrire une méthode de test qui utilise un "*Carre*", un "*Rectangle*" et un "*Triangle*".

[3]La signature est un peu compliquée, elle exprime des contraintes sur le type T générique mais pour nous il faut simplement que chaque élément de la liste doit implémenter comparable.

Chapitre 2. Travaux pratiques

1. Introduction à la programmation

1.1. 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
: une association simple représente une relation sémantique durable entre deux classes

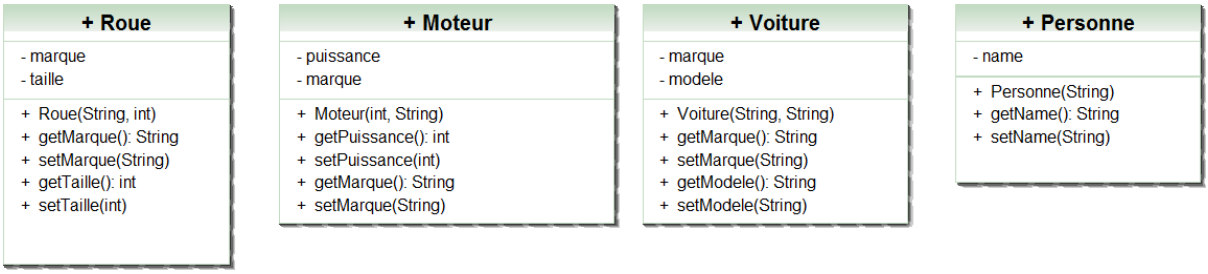
L'agrégation partagée
: Les agrégations sont des assocaitions non symétriques particulières qui signifient "contient" ou "est composé de" Pour l'agrégation partagée, les cycles de vie sont indépendants, les objets sont créés et détruit séparément. Un élément ne peut appartenir qu'a un seul agrégat composite

L'agrégation de composition ou composition
: Les cycles de vie sont liés, la destruction de l'agrégat composite entraine la destruction de tous ses ensembles. Le composite est responsable du cycle de vie de tous ses éléments.

1.1.1. Les classes d'origine

Coder les classes *Voiture*, Personne, *Moteur* et Roue du diagramme de classe UML suivant.

Figure 2.1. Classe isolées



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

1.1.2. Association simple

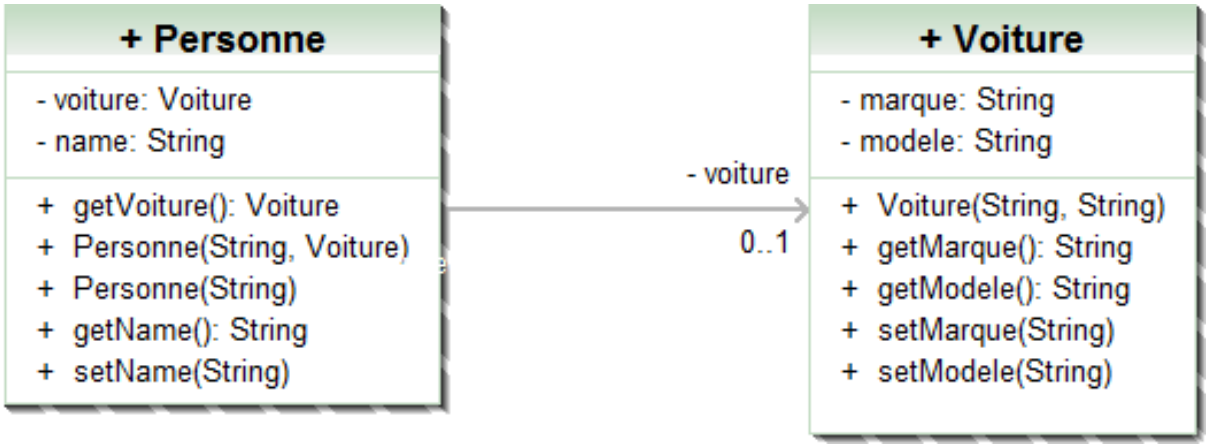
La communication entre classe se fait en rajoutant des variables d'instance. Si une voiture veut voir une personne alors la classe Voiture doit contenir une variable d'instance de type Personne, l'inverse et bien entendu vrai et les deux sens de visibilité peuvent cohabiter.

Le sens de la visibilité n'est pas le seul paramètre, tout comme pour les MCD (Modèle Conceptuel de Données) des cardinalités peuvent être exprimées, les cardinalités multiples seront impléméntées sous forme de collections ou de tableaux.

Faite en sorte qu'une personne possède au plus une voiture et que la voiture ne connaisse pas sont possesseur. Le digramme UML suivant doit pouvoir vous aider.

Une fois les classes modifiées, et en utilisant une classe munie d'un main faîte en sorte que la personne "Brard" puisse posséder une ferrari de modèle "pas cher". La personne "brard" peut-elle posséder deux voitures et peut-elle changer le modéle de sa voiture ? Si la personne "brard" ou la voiture disparait, l'autre disparait-il aussi ?

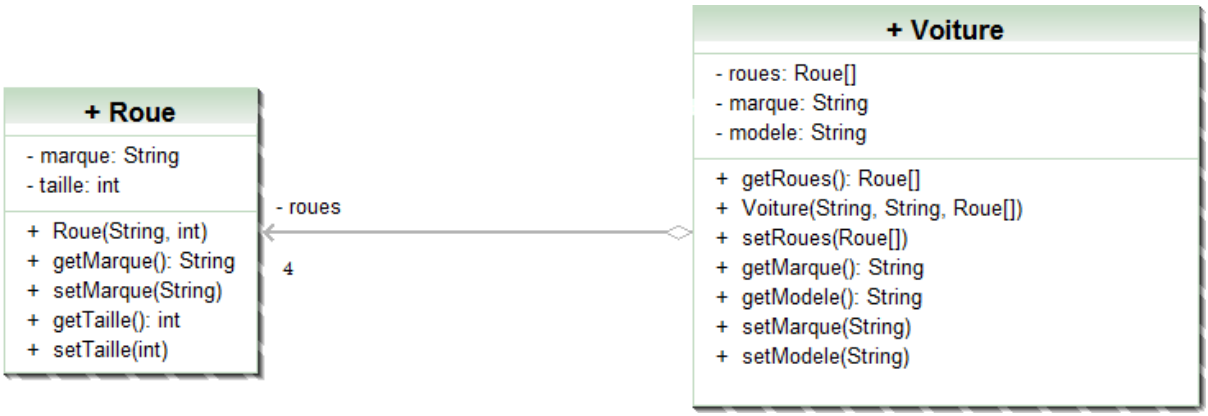
Figure 2.2. Association simple



1.1.3. Agrégation (partagée)

L'association partagée correspond à une association "groupe-élément", "tout-partie" où la disparition du groupe n'entraîne pas la disparition de l'élément. Modifier vos classes avec le diagramme suivant puis tester en faisant en sorte que la voiture Ferrari précédente ait quatre roues.

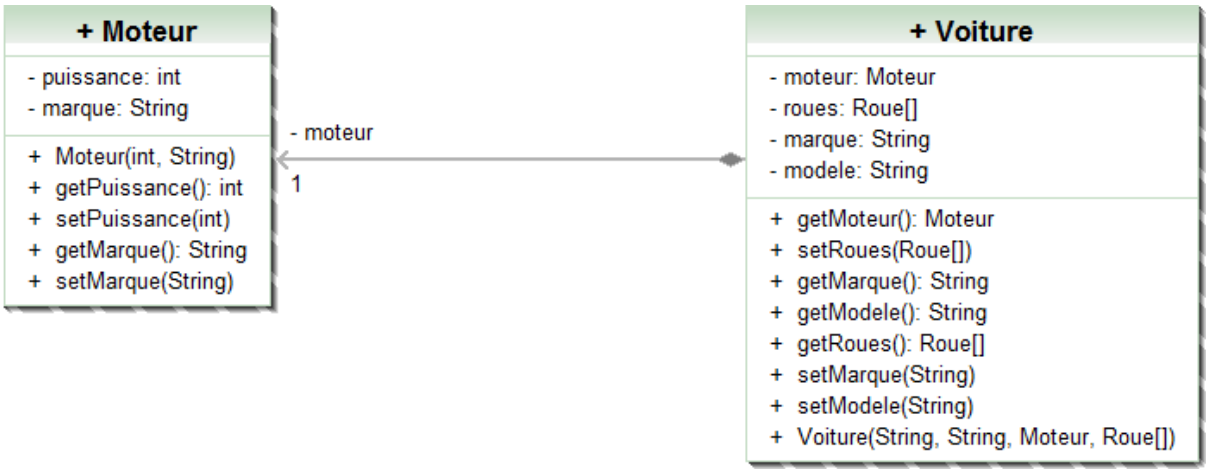
Figure 2.3. Agrégation partagée



1.1.4. La 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 2.4. Agrégation de composition

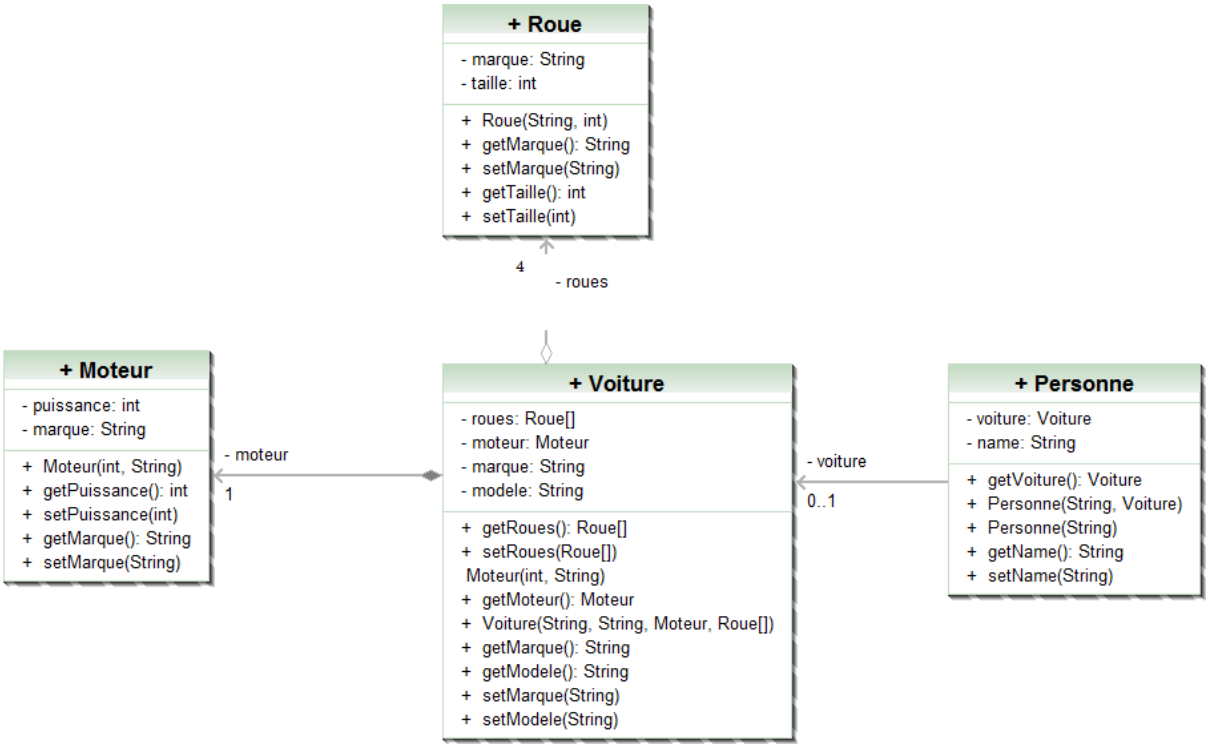


Nous avons modifier le constructeur de voiture pour que la voiture soit munie à la construction d'un moteur. Dans ce constructeur un nouveau moteur doit être construit (*new*) à l'image de celui passé en paramétre.

Créer un moteur (initial), puis une voiture, qui posséde une copie de ce moteur enfin modifier le moteur initiale, celui de la voiture est-il modifié.

Au final nous avons réalisé tout ceci :

Figure 2.5. Diagramme de classe complet



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

1.2. Héritage, classe abstraite et interface

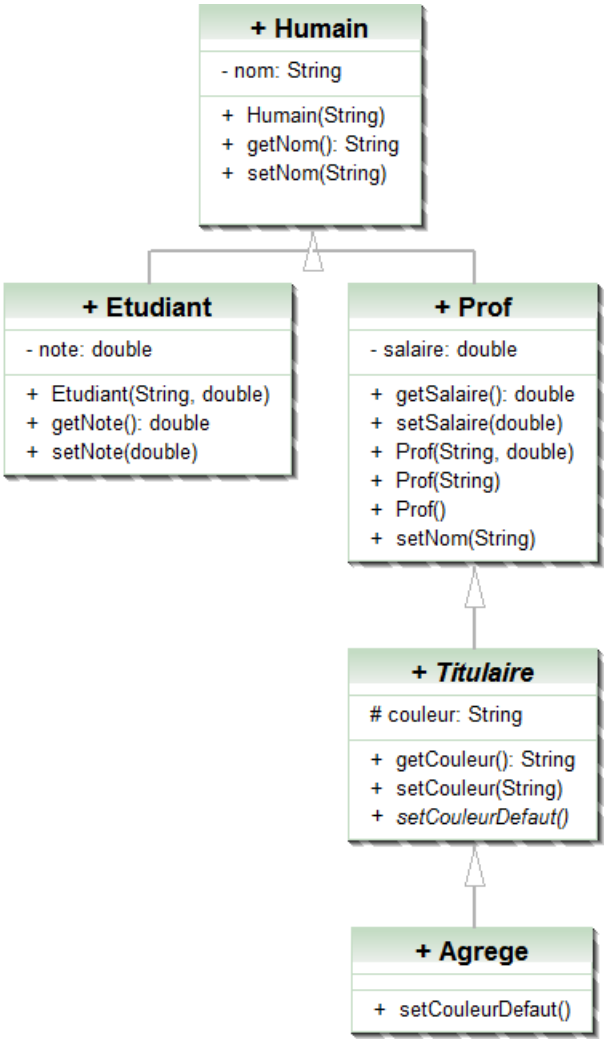
Nous allons étudier comment réutiliser des classes.

1.2.1. 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 de la classe mères. Chacune des classes filles peut donc posséder les mêmes caractéristiques que sa classe 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 2.6. 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 [1] 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()[2] créé un *prof* avec un *salaire* de 0 et sans *nom*(*null* ou *""*), le constructeur *Prof(String nom)* créé un *prof* de nom

[1] Si une classe n'a pas de constructeur, elle a un constructeur par défaut parcontre dès qu'une classe à un constructeur, elle n'a plus de constructeur par défaut. Le constructeur sans parmètre n'est pas le constructeur par défaut.
[2] Prof() est ici le constructeur sans paramètre de la classe Prof.

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.* La méthode setNom de Prof redéfinie celle d'Humain et impose que le nom soit mis en majuscule (toUpperCase).

Tester votre code.

1.2.2. 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 *setCouleurDefaut* (*public abstract void setCouleurDefaut();*), l'attribut *couleur* peut être déclaré *protected* (# en UML) pour pouvoir être modifié dans les classes héritées, il peut aussi être accesible via un setter public.

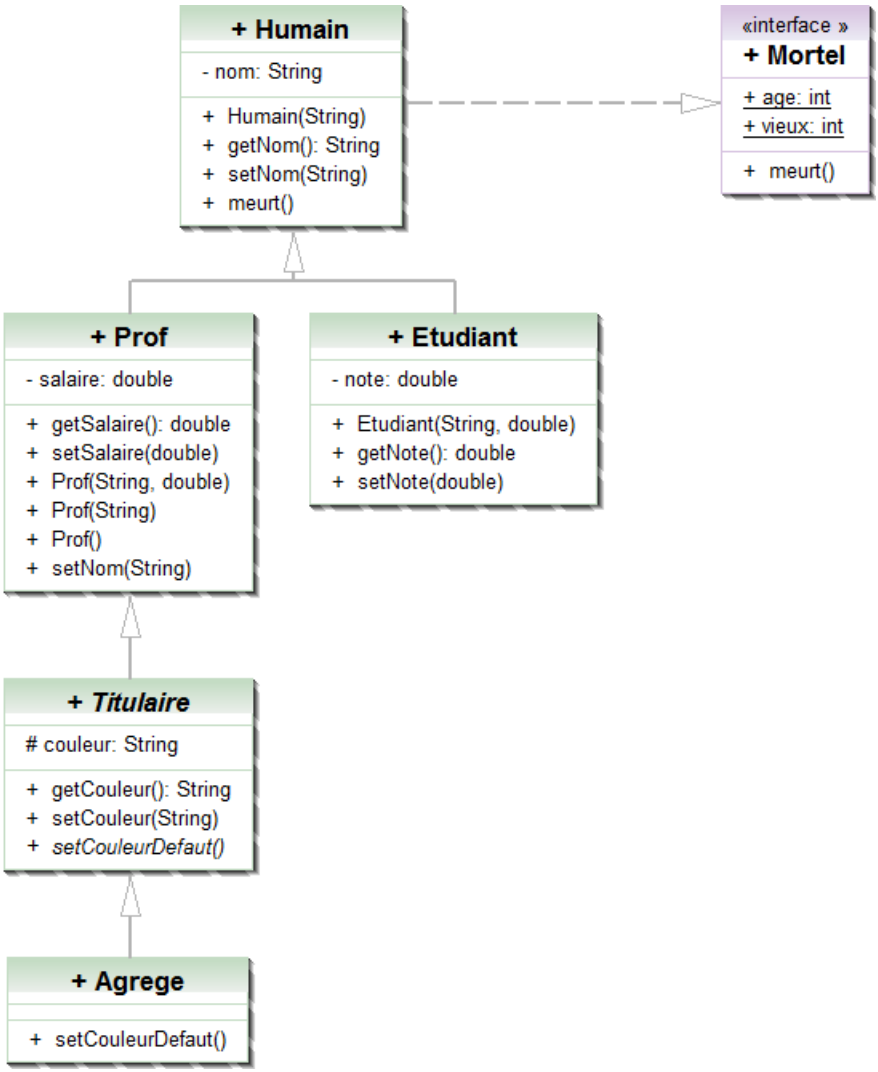
Créer la classe *Agrege*, les agrégés sont "roses". Tester votre code.

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 (écran un message à l'écran).

Figure 2.7. Interface



1.2.3. 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.

1.3. Swing

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

Figure 2.8. Structure d'une JFrame

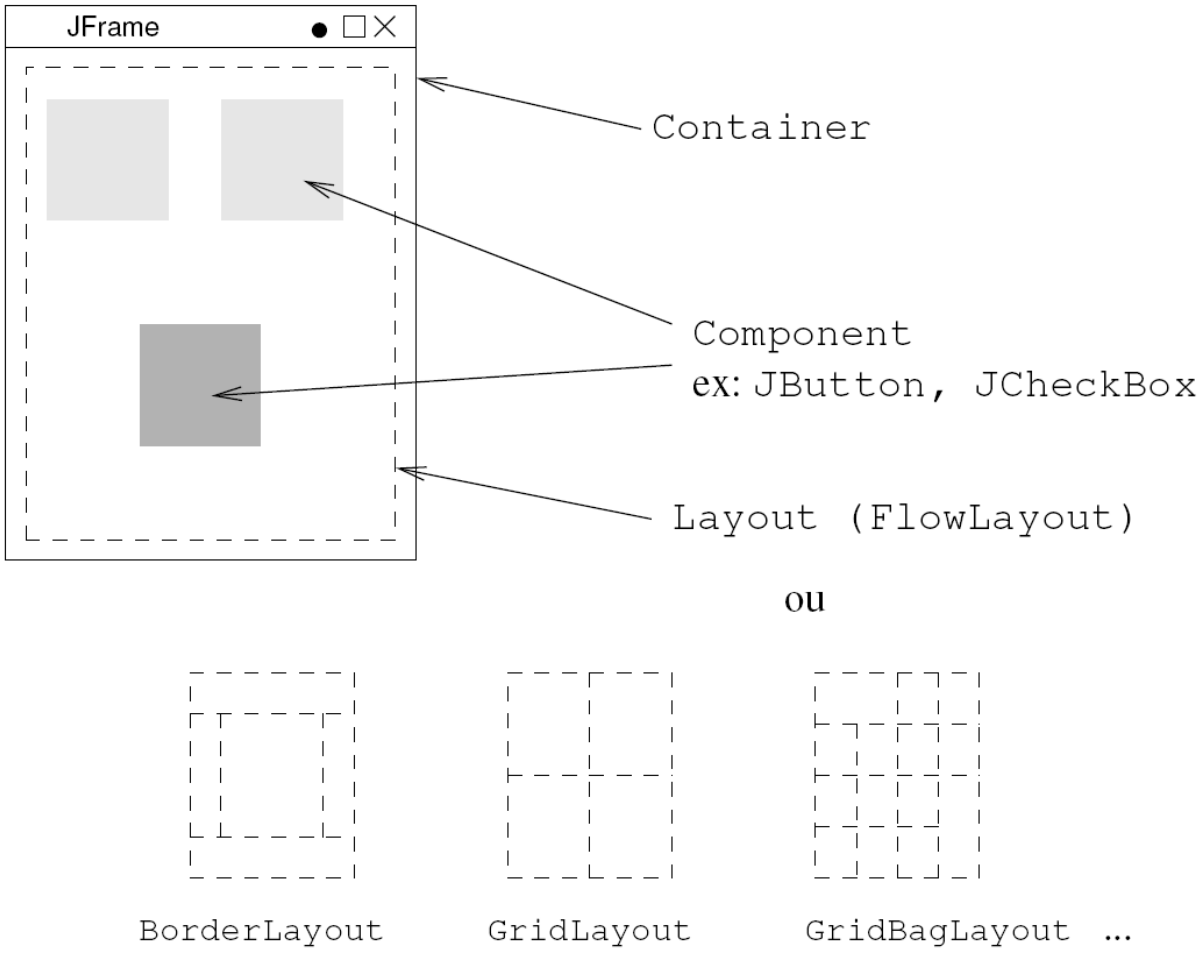
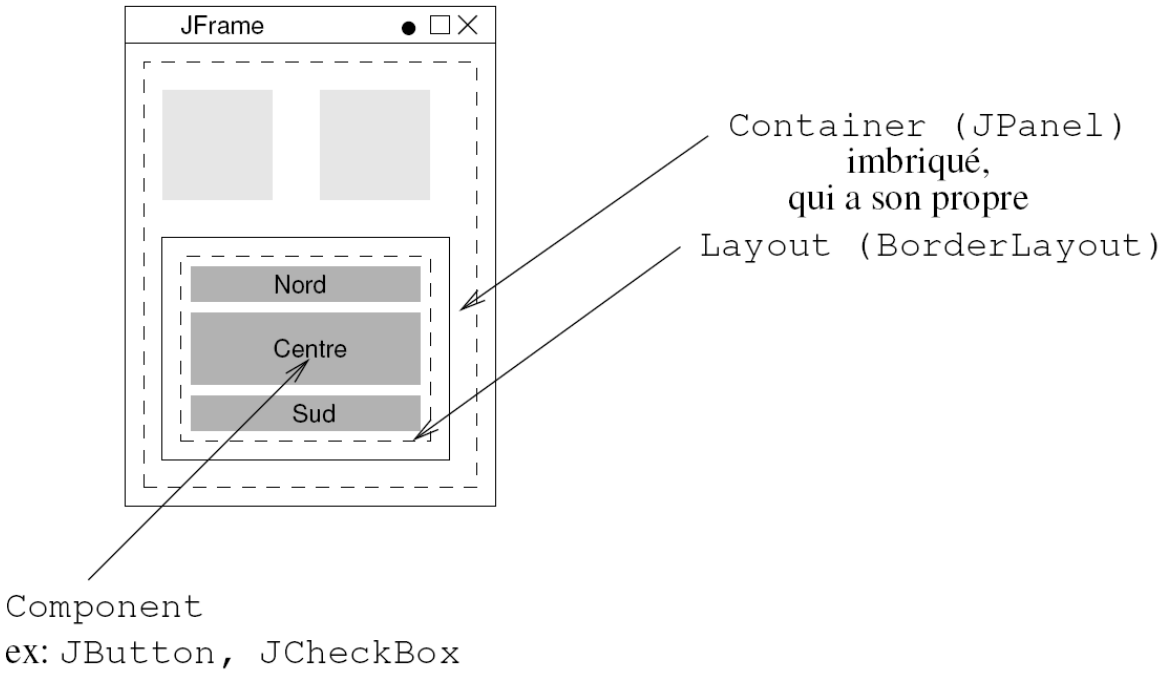


Figure 2.9. Méthodologie (imbrications)



1.3.1. 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 :

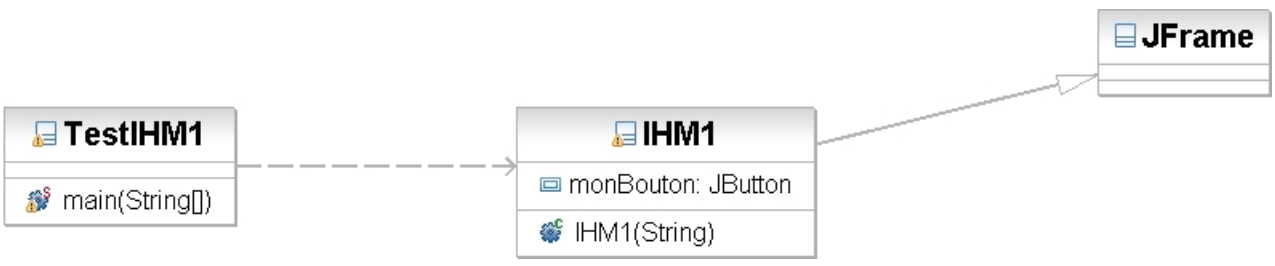
    ```
    f.pack() ;
    ```

- 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 2.10. IHM1



```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class IHM1 extends JFrame
{
 JButton monBouton;
 // contructeur de IHM1
 public IHM1(String titre)
 {
  super(titre);
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  this.setLayout(new FlowLayout());
  monBouton = new JButton("Click here !!!");
  this.add(monBouton);
  this.setSize(400,400);
  this.setVisible(true);
 }
}
```

```
public class TestIHM1 {
public static void main(String[] args)
{
IHM1 monIHM = new IHM1("Ma frame à moi");
}
}
```

Dans la première solution nous avons utilisé la JFrame comme classe mère mais nous pouvons aussi l'utiliser comme tous les Components (Container compris) en variables d'instances :

Figure 2.11. IHM2


```
import java.awt.FlowLayout;

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

public class IHM2 {
 JFrame f;
 JButton monBouton;
 // contructeur de IHM2
 public IHM2(String titre)
 {
 f = new JFrame(titre);
 f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 f.setLayout(new FlowLayout());
 monBouton = new JButton("Click here !!!");
 f.add(monBouton);
 f.setSize(400,400);
 f.setVisible(true);
 }
}
```

```
public class TestIHM2
{
public static void main(String[] args)
{
IHM2 monIHM = new IHM2("Ma frame à moi");
}
}
```

1.3.2. Réalisations

1.3.2.1. Fenêtre simple

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

Figure 2.12. 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

1.3.2.2. Disposition de composants graphiques dans une JFrame

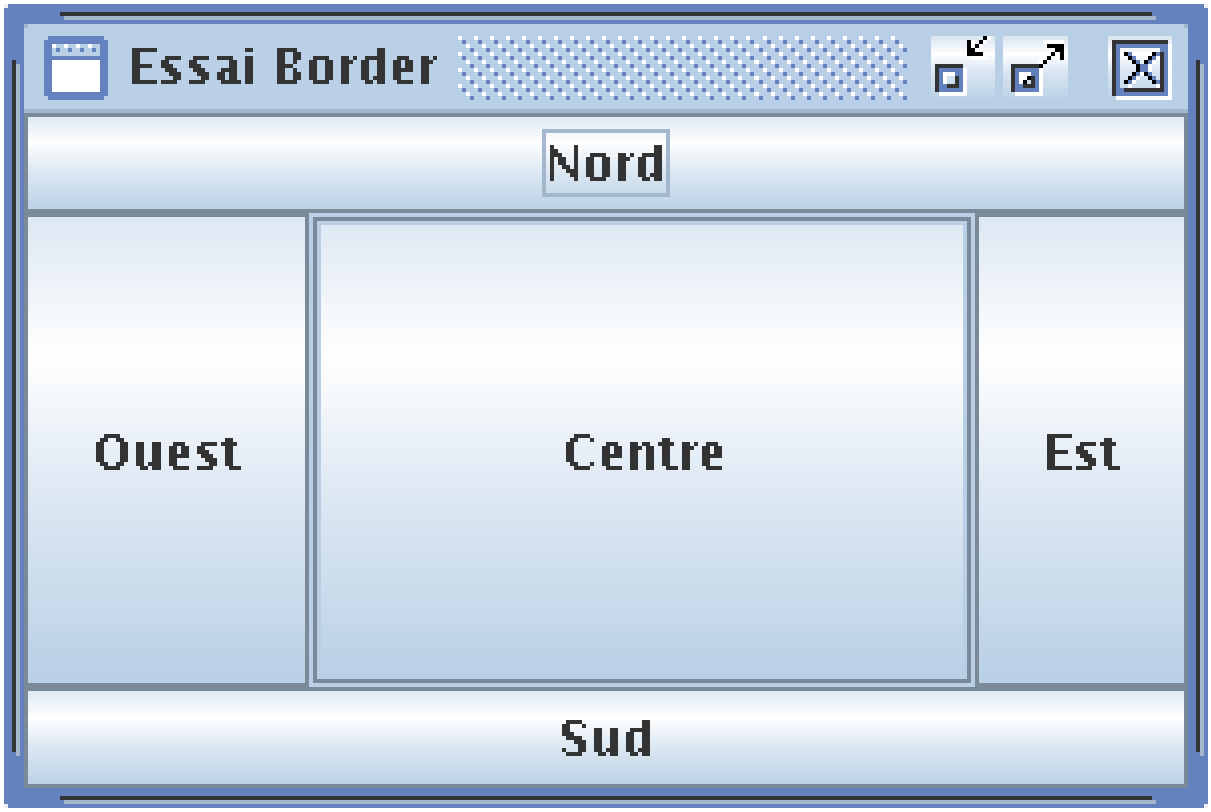
Dans la partie qui suit nous allons enrichir notre JFrame en ajoutant différents composants, la méthodologie suivante sera utilisée :

- 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 différents éléments (JButton, JComboBox ...) à la JFrame
- dimensionner la JFrame (à l'aide des méthodes setSize(int,int) ou pack())
- la faire afficher (à l'aide de setVisible(boolean)).

1.3.2.2.1. BorderLayout avec des composants JButton

Nous allons reproduire la JFrame suivante :

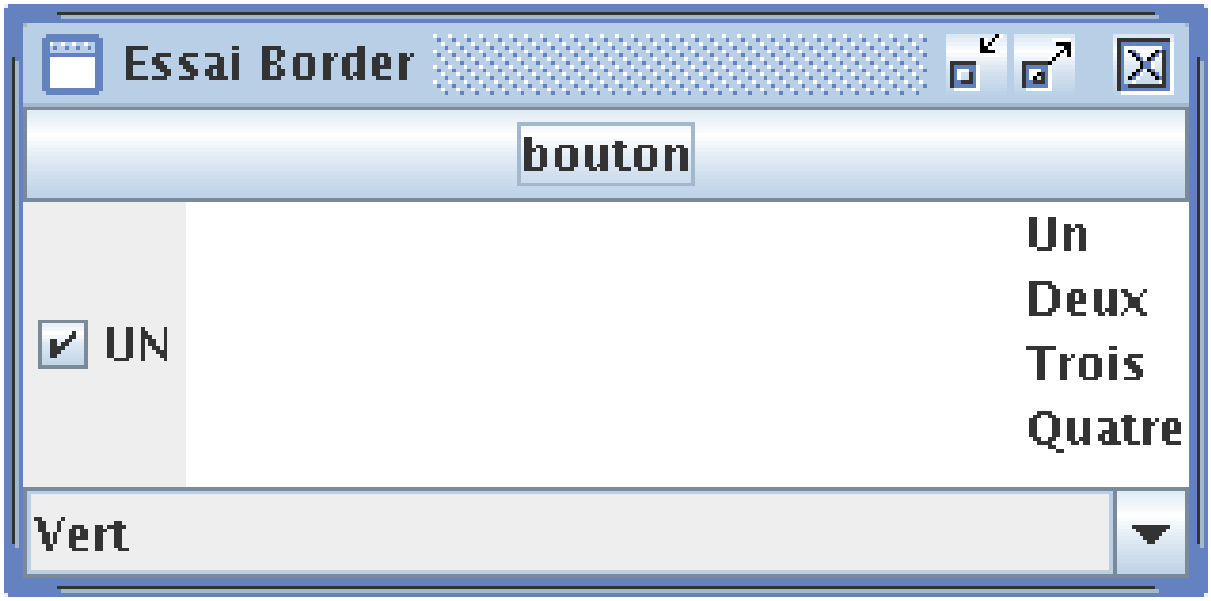
Figure 2.13. BorderLayout et JButton



1.3.2.2.2. BorderLayout avec d'autres composants

Nous allons reproduire la JFrame suivante :

Figure 2.14. 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

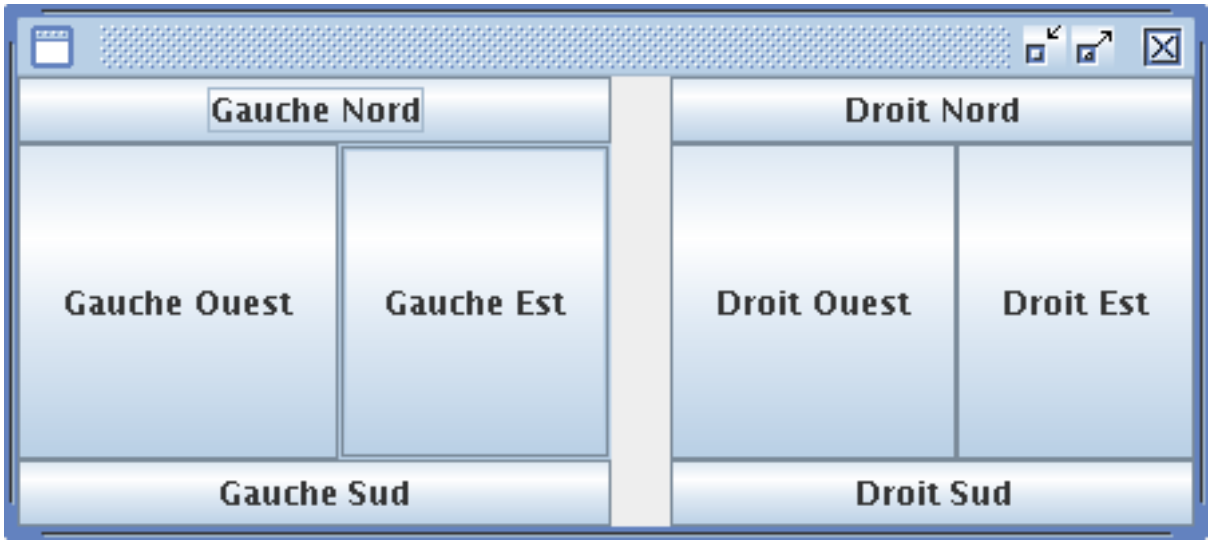
1.3.2.2.3. GridLayout

Reproduire la figure :

Figure 2.15. GridLayout

1.3.2.2.4. BorderLayout avec deux Panels

Figure 2.16. BorderLayout avec DEUX JPanels



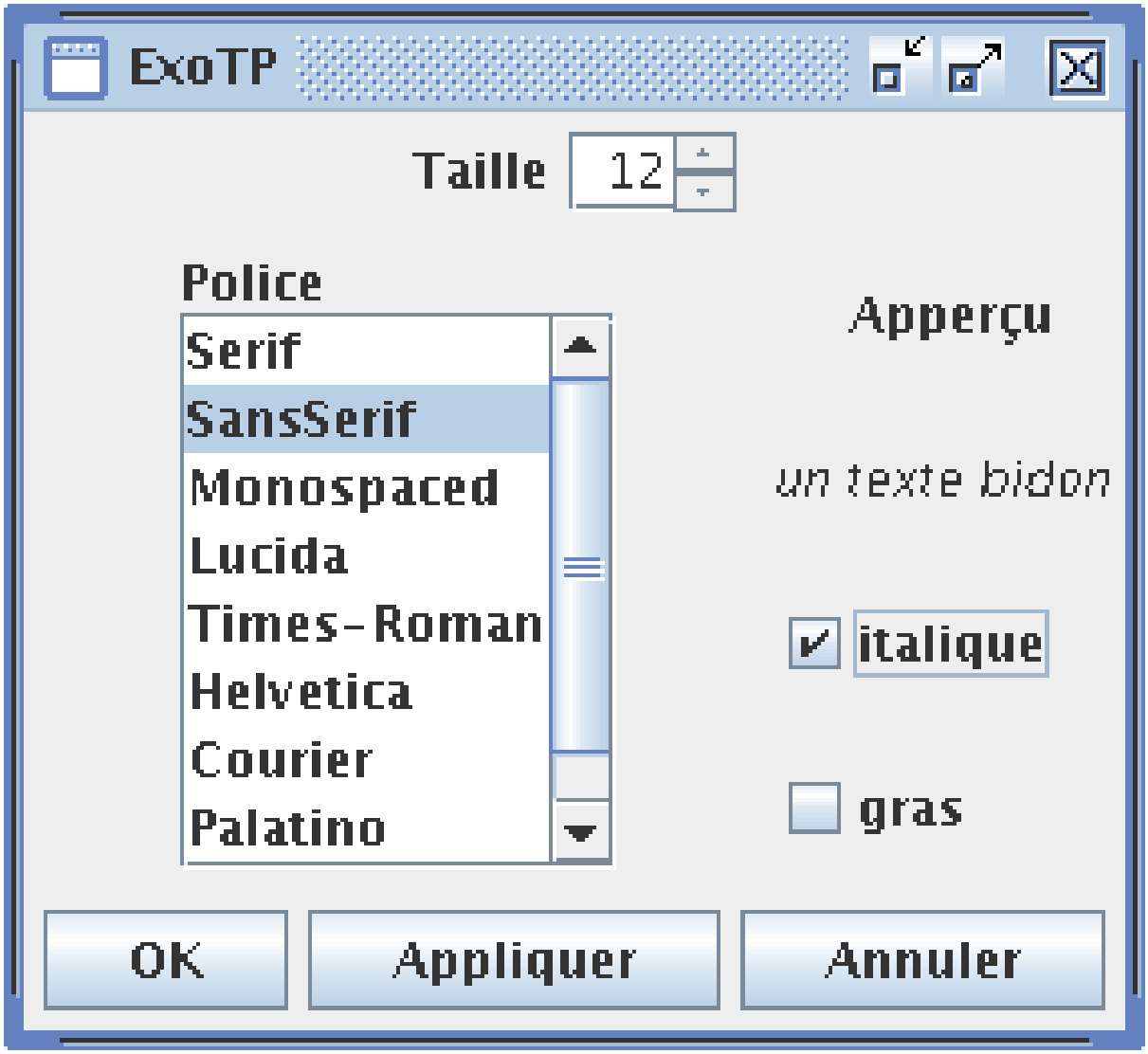
Reproduire la figure (dimension de la JFrame 650x200) :

- 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

1.3.2.3. Exercice récapitulatif

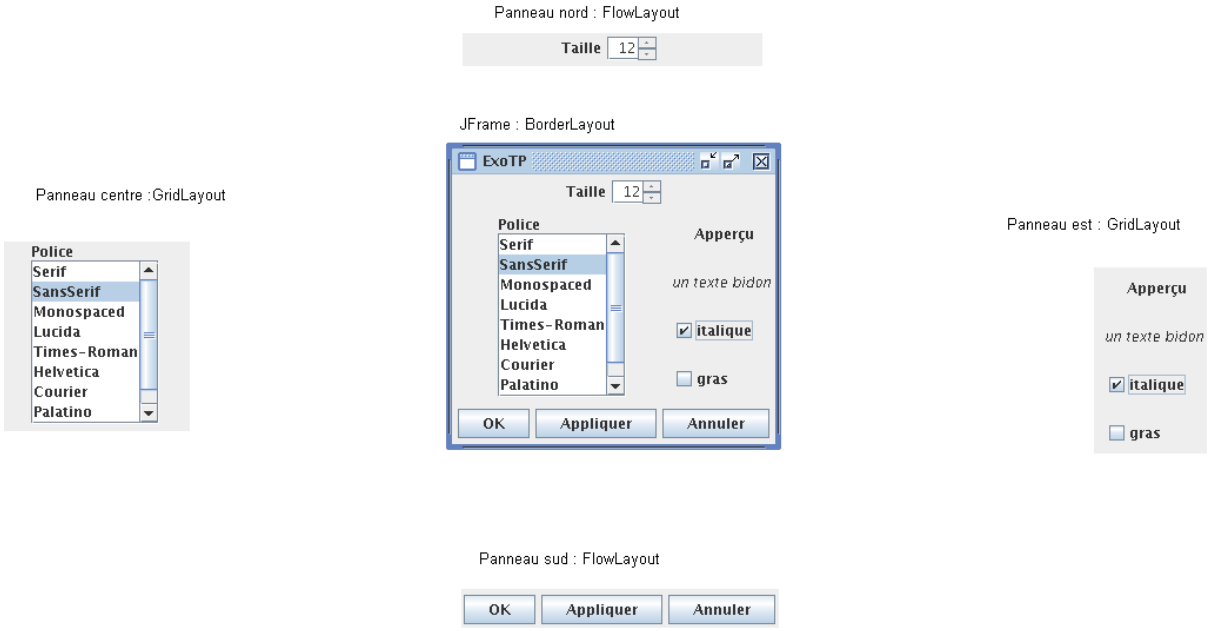
Reproduire à l'identique la JFrame donnée :

Figure 2.17. JFrame à reproduire



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

Figure 2.18. Découpage de la JFrame en Panels



Les ascenseurs sont obtenus avec des *JScrollPane*, les fontes avec *Font* et le compteur avec *JSPinner*. La liste des polices système est obtenue avec le code suivant :

```
String fontes[] = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
```

.

1.4. Evénements

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

1.4.1. 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 :

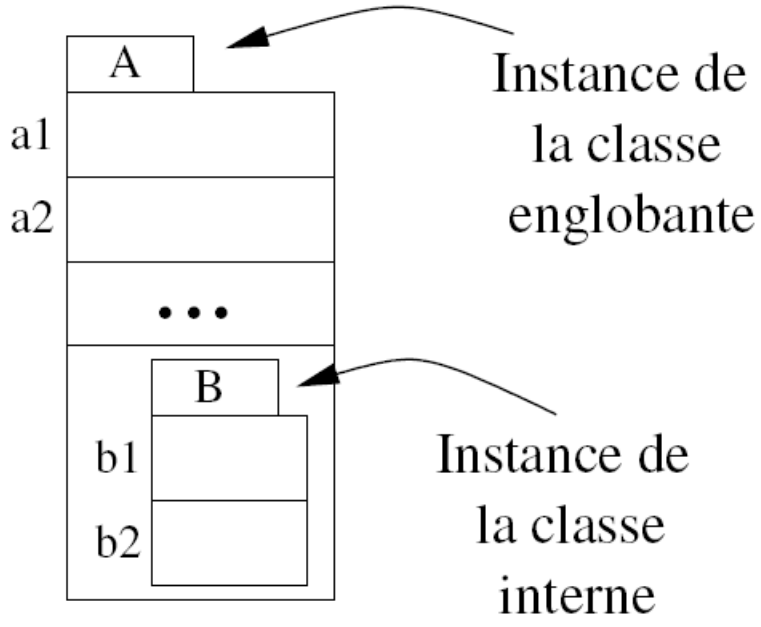
```
public class Englobante
{
class Interne
{
} // fin class Interne
}// fin classe Englobante
```

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 2.19. Dépendance entre instances



1.4.2. Architecture des classes

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

1.4.2.1. Base

La méthodologie est la suivante :

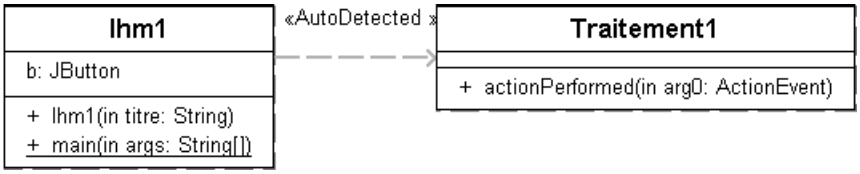
- Créer une classe *IHMn* qui crée l'ihm (par exemple, une *JFrame* et le positionnement de composants dans celle-ci)

- 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 2.20. 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);
  }

}
```

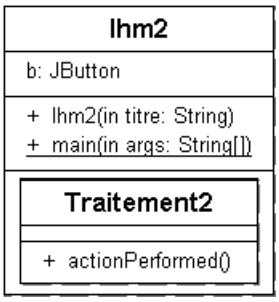
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

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.

1.4.2.2. Classe internes

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

Figure 2.21. 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.

1.4.2.3. 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 2.22. 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.

1.4.3. Tri des diérents Events

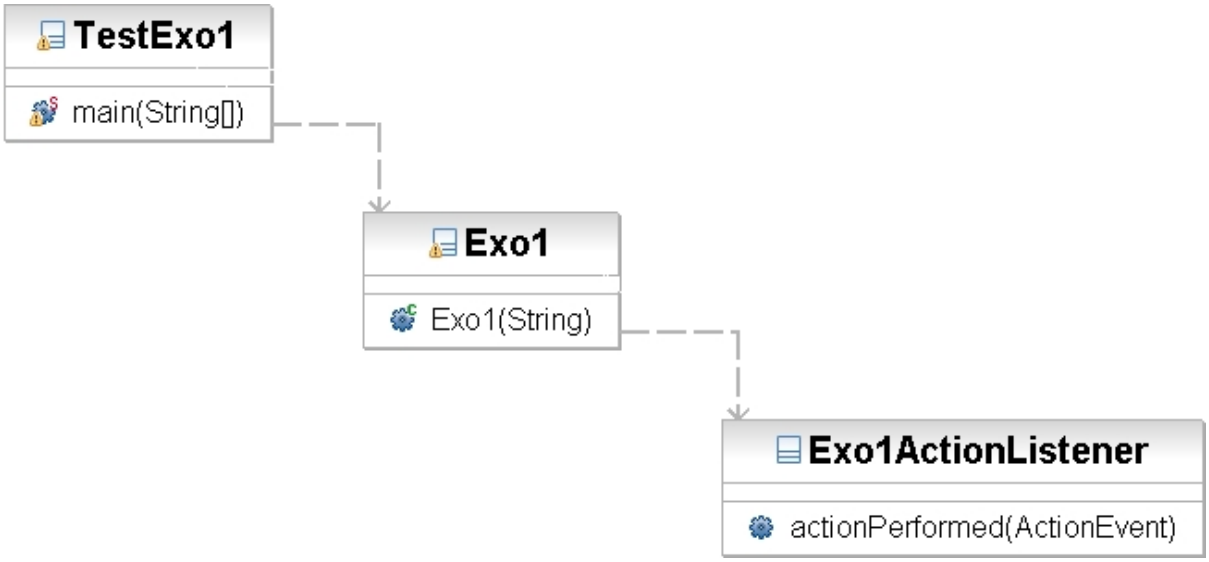
Si une application comporte des Components générant plusieurs Event de même nature (cas de plusieurs boutons, générant chacun un ActionEvent), on utilise la méthode getSource() pour connaître le responsable de l'événement.

1.4.4. Réalisations

1.4.4.1. Un bouton simple avec une classe externe pour gérer l'événement

Le but est de produire une *JFrame* avec un *JButton* qui permet de quitter l'application. Il vous faut en tout trois classe : la JFrame, la classe de traitement qui implémente *ActionListener* et une classe "lanceur" (avec un main).

Figure 2.23. Une action sur un bouton avec une classe externe



Indications :

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

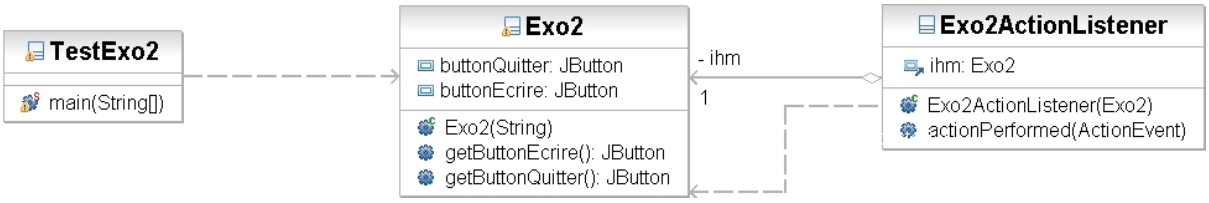
  ```
  public void actionPerformed(ActionEvent e)
  ```

)
- 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.

1.4.4.2. Deux boutons avec une classe externe

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.

Figure 2.24. Une action sur deux boutons avec une classe externe



Indications :

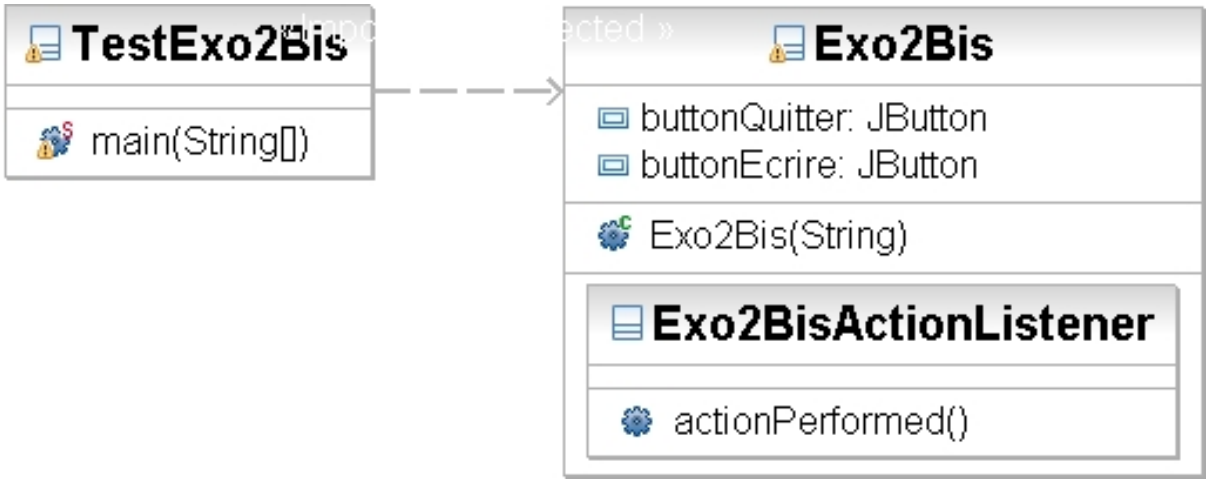
- Dans la JFrame les boutons apparaisent comme attributs car ils sont utilisés hors du constructeur.
- Des getters sont nécessaires sur les boutons pour permettre leur accès en dehors de l'IHM.
- Dans la classe de traitement qui implémente actionListener, il faut que la JFrame soit visible, ici elle est reçue via le constructeur.
- Dans la méthode *actionPerformed*, on peut savoir qui est le responsable de l'événement avec la méthode *getSource()* appliquée à un objet événement (ActionEvent). Ensuite l'utilisation de tests d'égalités permet de définir l'identification du responsable.

Comme nous venons de le voir le prix de la réutilisation du code aloudie le code, nous pouvons nous simplifier la vie avec des classes internes

1.4.4.3. Deux boutons avec une classe interne

Nous allons reproduire l'exercice précédent avec une classe interne pour gérer les événements. L'utilisation de la classe interne nous permet l'économie des getters, de l'attribut de type JFrame dans la classe de traitement et de son constructeur.

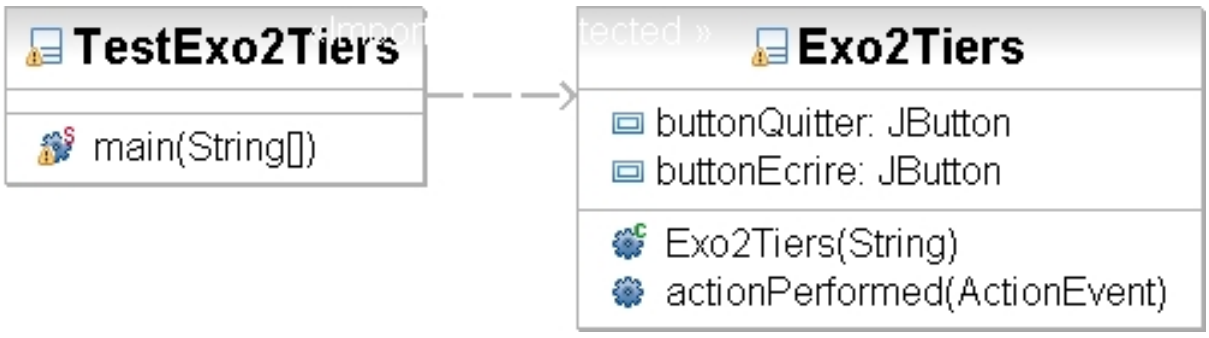
Figure 2.25. ne action sur deux boutons avec une classe interne



1.4.4.4. Deux boutons en implémentant directement l'interface dans la JFrame

Une classe peut à la fois hériter d'une classe mère et implémenter des interfaces, nous allons ici heriter de JFrame et implémenter l'interface ActionListener. Cette solution offre pour avantage de faire disparaître la classe interne. L'inconvéniant est la lisibilité du code surtout si deux interfaces ont les mêmes méthodes.

Figure 2.26. Une action sur deux boutons en implémentant directement l'interface



1.4.4.5. Actions plus complexes

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

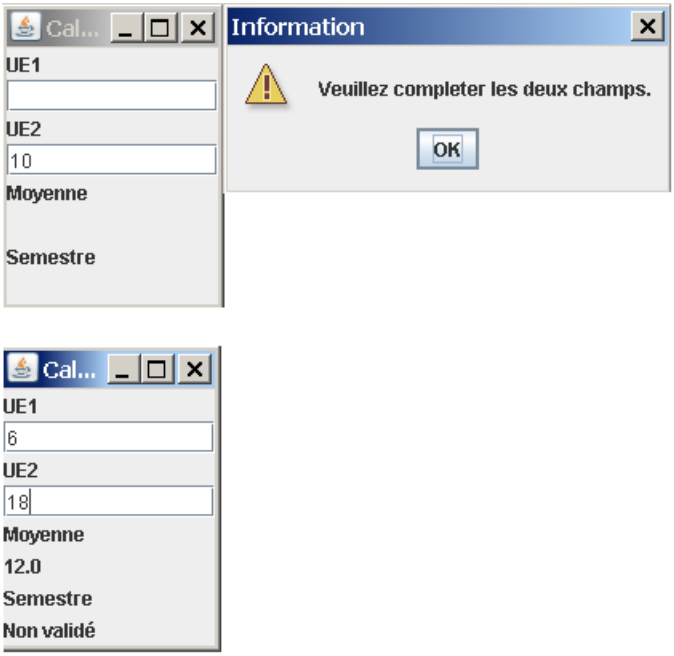
Indications :

- 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.

1.4.4.6. Exercice récapitulatif

Réaliser le calculateur de semestre suivant :

Figure 2.27. 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) est vide (*.equals("")*) alors une fenêtre de dialogue doit être affichée

(JOptionPane.showMessageDialog). Cette fenêtre doit-être liée à la fenêtre qui la lancée. Si la classe externe est Semestre sont objet courant est vu dans las classe interne comme *Semestre.this*.

1.5. Dessiner en java

2. Programmation d'un "shoot them up"

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

2.1. 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.

2.1.1. 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 (setPreferedSize).
2. Créer une classe TestCercleFrame qui hérite de JFrame et 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`

2.1.2. 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

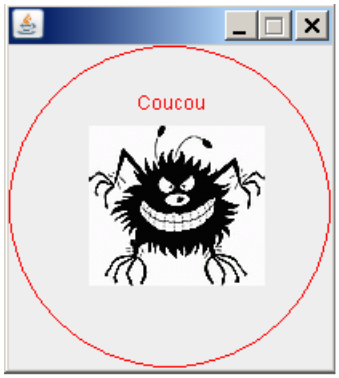
Nous pouvons améliorer la lecture de notre son en le lançant dans un nouveau thread. Le thread est créé avec la classe interne suivante :

```
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 2.28. TestJFrame



2.1.3. 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.

2.1.3.1. 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.

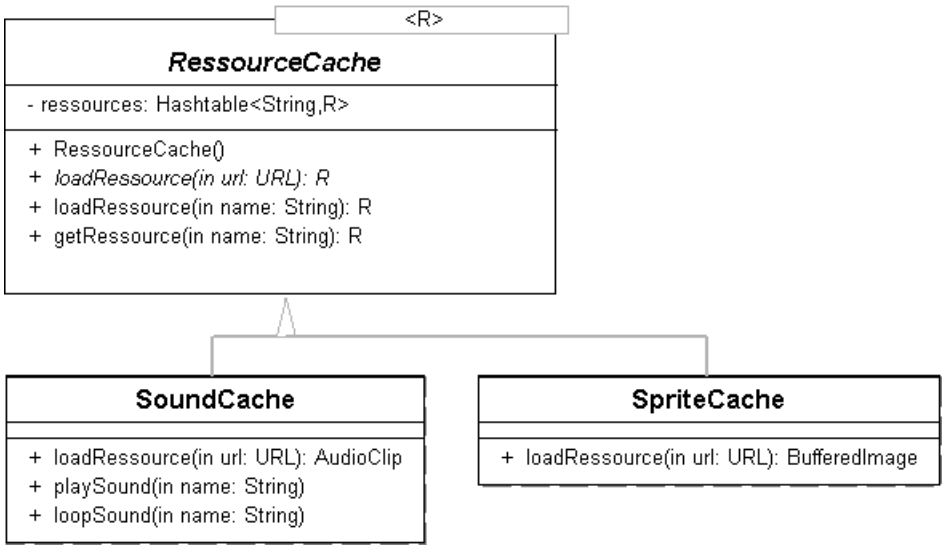
Le code RessourceCache est le suivant :

```
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 2.29. Gestionnaire de ressources



2.1.3.2. 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.*

2.1.3.3. 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 } });

2.2. 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) :

2.3. Définir les acteurs (Entity)

2.3.1. Les monstres

2.3.2. Le joueur

2.3.3. Les tirs

2.4. Définir le jeux

2.5. A vous de continuer