Paradigmes de Progammation I
Travaux pratiques

Professeur : B. Le Charlier

Assistants : Michaël Petit, Isabelle Pollet

Avril 2000

1 Introduction

Ce document rassemble les différents énoncés relatifs à la partie Orienté Objet des travaux pratiques du cours de Paradigmes de Programmation (Première Partie) ainsi que certains renseignements pratiques concernant l'organisation de ces TPs.

Ces TPs sont constitués, d'une part, de quelques séances d'exercices en salle et, d'autre part, de la remise d'un travail en deux parties. La première partie de ce travail consiste en la définition et en l'implémentation d'un **type logique** et de quelques instanciations de celui-ci, tandis que la seconde partie s'attarde sur l'élaboration d'une **interface graphique** simple pour celui-ci. Ces différentes implémentations seront bien sûr réalisées en Java.

Le but de ces TPs est de mettre en pratique les différentes notions abordées au cours. Il ne s'agit donc absolument pas de passer des jours de recherche dans diverses sources de documentation sur le langage Java.

Pour éviter cet écueil, nous imposons, dans la réalisation du travail, les contraintes suivantes :

- l'implémentation relative à la première partie doit entièrement reposer sur le sous-langage *VTF* de Java décrit dans le document *Définition du Langage Vas-T'y-Frotte*; on adjoint cependant à ce langage les opérations de casting et le test `instanceof` ;
- l'implémentation relative à la seconde partie doit entièrement reposer sur le sous-langage *VTF* (augmenté des opérations de casting, du test `instanceof` et des interfaces) et sur le sous-ensemble des classes Swing qui sera présenté lors du dernier cours théorique.

Il est évident que le respect ou non de ces contraintes intervient dans la note attribuée au travail.

Si, par hasard, vous avez des questions ou des remarques, vous pouvez

- soit les poser directement aux séances d'exercices (!!!),
- soit les envoyer par mail (`ipo@info.fundp.ac.be` ou `mpe@info.fundp.ac.be`, selon le groupe d'exercices auquel vous appartenez).

2 Planning et groupes de TP

Voici le planning des séances d'exercices relatives à la partie "Orienté Objet" du cours de Paradigmes de Programmation (Première Partie).

Samedi 1/04	Exercices en salle
Jeudi 13/04	Cours (partie graphique)
Mardi 18/04	Exercices en salle et présentation de la première partie du TP
Samedi 29/04	Questions/Réponses (première partie du TP + cours)
Samedi 06/05	Date limite de remise de la première partie du TP
	Exercices en salle et présentation du TP2
Samedi 13/05	Questions/Réponses (seconde partie du TP + cours)
Samedi 20/05	Date limite de remise de la seconde partie du TP

Quelques remarques concernant ce planning :

- usuellement, vous serez repartis en deux groupes d'exercices, cependant le samedi 06/05 les deux groupes fusionneront en un seul (suite à un problème d'horaire);
- deux séances de questions/réponses sont prévues au planning (le samedi 29/04 et le samedi 13/05); ces séances sont prévues, d'une part, pour vous permettre de poser des questions sur le TP en cours et, d'autre part, pour éventuellement réexpliquer et/ou illustrer certaines parties du cours de votre choix; plus précisément, si vous souhaitez des explications sur une partie déterminée du cours, faites-le nous savoir, au plus tard, lors de la séance précédant chacune de ces séances de questions/réponses; dans le cas contraire, un fois les questions sur le TP épuisées, nous considèrerons la séance d'exercices terminée.

Pour réaliser les TPs, nous vous demandons de former des groupes de une, deux ou trois personnes. Nous attribuerons la cote du TP indépendamment du nombre d'étudiants formant le groupe de TPs. Nous vous demandons de nous communiquer la liste des groupes pour le mardi 18/04.

Ces groupes doivent être formés en respectant la contrainte suivante : *tous les étudiants d'un groupe de TPs donné doivent appartenir au même groupe d'exercices* .

3 Quelques petits exercices sur Java

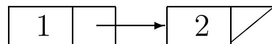
Exercice 1 (Pour se rappeler la syntaxe)

Réaliser une classe Java permettant de représenter une liste simplement chaînée d'entiers. Cette classe doit posséder des méthodes permettant

- d'ajouter un entier en début de liste,
- de tester la présence d'un entier dans une liste (version récursive),
- d'afficher une liste à l'écran (version itérative faisant appel au package MyCrt).

Cette classe doit appartenir au package `Liste`.

Ecrire ensuite un petit programme construisant la liste ci-dessous et affichant son contenu à l'écran.



Exercice 2 (Constructeurs)

Considérons le petit programme Java ci-après, quelle sera la valeur affichée à l'écran ?

```
class  A
{
  int val;
};

class B extends A
{
  B() {val = 100;}
  B(int v) {val = val + v;}
};

public class Const
{
  public static void main(String[] args)
  {
    A x = new B(5);
    MyCrt.MyIO.Write(x.val);
  }
}
```

Exercice 3 (Appels de méthodes)

Considérons les définitions de classes suivantes.

```
class  A
{
  A autre;

  void meth(A v) { autre = new A(); }
};

class B extends A
{
  int val;

  void meth(A v) { super.meth(v); val = val + 1; }
  void meth(B v) { autre = v; }
};
```

Décrire (représenter) les situations résultant des morceaux de programmes ci-après (si celles-ci ont un sens).

1. ```
   A x = new A();
   A y = new A();
   x.meth(y);
   ```
2. ```
   A x = new A();
   A y = new B();
   x.meth(y);
   ```
3. ```
   A x = new A();
   B y = new B();
   x.meth(y);
   ```
4. ```
   B x = new B();
   B y = new B();
   x.meth(y);
   ```
5. ```
   B x = new B();
   B y = new A();
   x.meth(y);
   ```
6. ```
   A x = new B();
   B y = new B();
   x.meth(y);
   ```

4 Exercice introductif

Les files sont des structures relativement simples principalement caractérisées par deux opérations :

- ajout d'un élément à la fin de la file,
- retrait (départ) d'un élément en tête de file.

On demande de réaliser pas à pas les étapes suivantes.

4.1 Spécification

Il s'agit de spécifier le type logique "file d'éléments de type t". On veut que ce type dispose de 5 primitives : une primitive de création, une primitive d'ajout, une primitive de retrait, une primitive permettant de tester si la file est vide et une primitive permettant de tester l'égalité de deux files.

On demande de donner 3 spécifications :

1. une spécification par type abstrait (approche valeur),
2. une spécification par "modèle de haut niveau" (approche valeur),
3. une spécification selon l'approche objet.

4.2 Implémentation

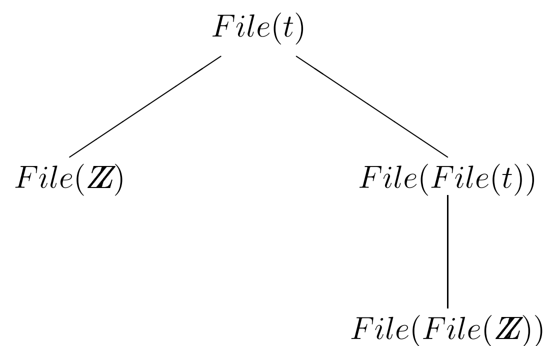
On demande de fournir une implémentation en Java pour le type logique spécifié à l'étape précédente. Cette implémentation appliquera les principes de simulation de la généricité expliqués au cours et permettra, en outre, la lecture au clavier et l'affichage à l'écran d'un élément de $File(t)$.

Il s'agit donc, premièrement, de définir précisement des conventions de représentation pour un objet de type $File(t)$ et de spécifier les différentes méthodes permettant la représentation du type logique et, deuxièmement, d'implémenter cette représentation.

Plus précisément, on demande d'implémenter :

1. une représentation du type générique $File(t)$,
2. une représentation de l'instanciation $File(\mathbb{Z})$,
3. une représentation du type générique $File(File(t))$,
4. une représentation de l'instanciation $File(File(\mathbb{Z}))$.

Il s'agit donc d'implémenter la hiéarchie ci-après.



Remarque : en ce qui concerne les lectures au clavier et les affichages à l'écran, on se contente d'une interface "commande" réalisée à l'aide du package MyCrt qui sera fourni et dont la spécification est donnée dans le document *Spécification et Vérification de Classes Java Abstraites (Génériques) et Concrètes : Exemples* .

5 TP : première partie

Comme signalé dans l'introduction, cette partie du TP consiste en la définition et en l'implémentation de types logiques et de quelques instanciations de ceux-ci. Nous avons porté notre choix sur deux types logiques “usuels” : l' **ensemble ordonné** et la **fonction**.

Ce TP se décompose en trois parties principales :

1. spécification des deux types logiques mentionnés ci-avant,
2. implémentation de ces deux types logiques et de quelques instanciations de ceux-ci (en respectant une hiérarchie donnée),
3. implémentation d'une “application” manipulant ces types.

5.1 Types Logiques

Cette section décrit, de manière incomplète, les deux types logiques à implémenter. La première partie de l'exercice consiste justement à compléter cette description.

5.1.1 Ensemble ordonné

On considère des ensembles finis d'éléments de type t. Le type t est muni, d'une part, d'une relation d'égalité et, d'autre part, d'un ordre total strict symbolisé par “ $<$”.

Pour rappel, une relation binaire $<$ sur un espace E est une relation d'ordre strict si et seulement si

- $<$ est anti-réflexive (i.e. $\forall x : x \in E : \neg(x < x)$),
- $<$ est transitive (i.e. $\forall x, y, z : x, y, z \in E : (x < y \wedge y < z) \Rightarrow x < z$),
- $<$ est anti-symétrique (i.e. $\forall x, y : x, y \in E : \neg(x < y \wedge y < x)$).

En outre, un ordre strict est dit total si tous les éléments de l'espace sont “comparables entre-eux” (i.e. $\forall x, y : x, y \in E : x < y \vee y < x$).

On désire que le type logique **ensemble ordonné** (que l'on note $Ens(t)$) dispose des “primitives” suivantes : créer un ensemble, ajouter un élément à un ensemble, retirer un élément donné d'un ensemble, tester si un ensemble est vide, tester l'appartenance

d'un élément à un ensemble, tester l'égalité de deux ensembles, donner le maximum d'un ensemble (relativement à la relation $<$), donner le minimum d'un ensemble et, enfin, déterminer le successeur d'un élément d'un ensemble dans cet ensemble.

Les signatures exactes de ces primitives sont données dans ce "morceau de type abstrait".

- **Sortes**
  t : paramètre
  $Ens(t)$: ensemble d'éléments de type t
  $bool$: booléen
- **Signatures**
  $Vide :\longrightarrow Ens(t)$
  $Ajout : Ens(t) \times t \longrightarrow Ens(t)$

  $Appartient : Ens(t) \times t \longrightarrow bool$
  $EstVide : Ens(t) \longrightarrow bool$
  $Egale : Ens(t) \times Ens(t) \longrightarrow bool$
  $Min : Ens(t) \longrightarrow t$
  $Max : Ens(t) \longrightarrow t$
  $Succ : Ens(t) \times t \longrightarrow t$

  $Enlever : Ens(t) \times t \longrightarrow Ens(t)$

On demande de

1. compléter la spécification par type abstrait donnée ci-avant (i.e. donner les axiomes),
2. donner une spécification selon l'approche objet (cohérente avec la spécification fournie au point précédent).

5.1.2 Fonction de domaine fini

Le second type logique auquel nous nous intéressons est le type **fonction de domaine fini**.

Plus précisément, on considère deux types paramètres t_1 et t_2 qu'on suppose munis tous deux, d'une part, d'une relation d'égalité et, d'autre part, d'un ordre total strict (ces ordres sont respectivement notés $<_1$ et $<_2$) et on s'intéresse aux fonctions de domaines finis (inclus dans t_1) et à valeurs dans t_2.

On désire que le type logique **fonction de domaine fini** (que l'on note $Fonc(t_1, t_2)$)

dispose des “primitives” suivantes : créer une fonction, mettre à jour la valeur d'une fonction en un point, retirer un élément du domaine de définition d'une fonction, donner le domaine d'une fonction, donner l'image d'un point par une fonction et tester l'égalité de deux fonctions.

Les signatures exactes de ces primitives sont données dans le “morceau de type abstrait” ci-dessous.

- **Sortes**
  t_1, t_2 : paramètres
  $Ens(t_1)$: ensemble d'éléments de type t_1
  $Fonc(t_1, t_2)$: fonction de t_1 dans t_2
  $bool$: booléen

- **Signatures**
  $Vide :\longrightarrow Fonc(t_1, t_2)$
  $MiseAJour : Fonc(t_1, t_2) \times t_1 \times t_2 \longrightarrow Fonc(t_1, t_2)$

  $Im : Fonc(t_1, t_2) \times t_1 \longrightarrow t_2$
  $Egale : Fonc(t_1, t_2) \times Fonc(t_1, t_2) \longrightarrow bool$
  $Dom : Fonc(t_1, t_2) \longrightarrow Ens(t_1)$

  $Retire : Fonc(t_1, t_2) \times t_1 \longrightarrow Fonc(t_1, t_2)$

On demande de

1. donner une spécification du type $Fonc(t_1, t_2)$ par modèle de haut niveau,
2. donner une spécification selon l'approche objet (cohérente avec la spécification fournie au point précédent).

5.2 Représentation et instanciations

Cette partie du TP s'attarde sur l'implémentation en Java des deux types logiques spécifiés à l'étape précédente. Pour réaliser cette implémentation, on applique les principes de simulation de la généricité expliqués au cours. On demande, en outre, d'ajouter aux deux types logiques une primitive d'affichage à l'écran.

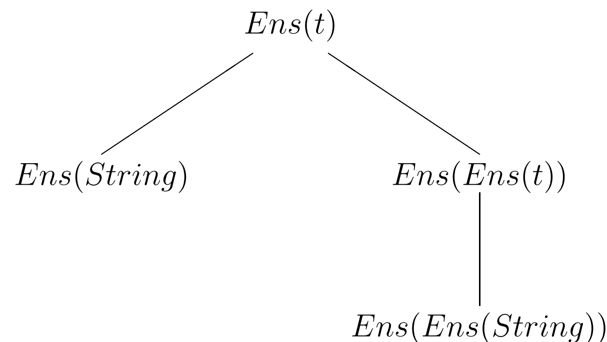
5.2.1 Ensemble ordonné

Il s'agit, premièrement, de définir précisement des conventions de représentation pour un objet de type $Ens(t)$ et de spécifier les différentes méthodes permettant la représentation du type logique et, deuxièmement, d'implémenter cette représentation.

Plus précisément, on demande d'implémenter :

1. une représentation du type générique $Ens(t)$,
2. une représentation de l'instanciation $Ens(String)$,
3. une représentation du type générique $Ens(Ens(t))$,
4. une représentation de l'instanciation $Ens(Ens(String))$.

Il s'agit donc d'implémenter la hiéarchie ci-après.



Attention : nous vous demandons de veiller à exploiter l'existence d'une relation d'ordre lors de votre choix de représentation (l'existence d'une relation d'ordre doit, en effet, permettre d'optimiser certaines recherches). Nous vous demandons, en outre, de justifier vos choix de représentation.

Remarque 1 : comme pour l'exercice introductif, en ce qui concerne les affichages à l'écran, on se contente d'une interface "commande" réalisée à l'aide du package `MyCrt`.

Remarque 2 : on peut induire de l'ordre total sur t, un ordre total sur $Ens(t)$. Le principe de cet ordre est celui des ordres lexicographiques (alphabétiques). Il est évident qu'un tel ordre vous sera indispensable pour implémenter $Ens(Ens(t))$. La définition de cet ordre doit donc figurer dans votre rapport. Soulignons, au passage, que l'inclusion n'est pas un ordre total.

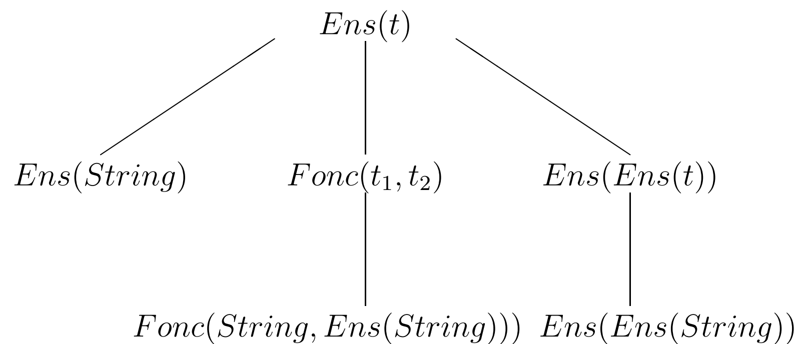
5.2.2 Fonction de domaine fini

En ce qui concerne la représentation de $Fonc(t_1, t_2)$, nous ne vous laissons en fait que très peu de choix. En effet, nous vous imposons de voir les éléments de $Fonc(t_1, t_2)$ comme des éléments particuliers de $Ens(t_1 \times t_2)$. Autrement dit, nous vous demandons de voir la représentation de ce type logique comme une classe héritant de la classe Java représentant $Ens(t)$.

Cependant, comme pour le type logique précédent, il s'agit de définir précisement des conventions de représentation pour un objet de type $Fonc(t_1, t_2)$, de spécifier les différentes méthodes permettant la représentation du type logique et d'implémenter cette représentation.

On demande, en outre, d'implémenter l'instanciation $Fonc(String, Ens(String))$.

La hiérarchie "complète" à implémenter devient donc :



Cette hiérarchie n'est évidemment pas nécessairement "exhaustive" et des classes intermédaires peuvent apparaître.

5.3 Application

Pour terminer, nous vous demandons de réaliser une petite "application" manipulant les différentes classes construites au cours du TP. Cette "application" se décompose en deux éléments principaux : une classe permettant de représenter un *Dictionnaire* et un programme de test manipulant des instances de cette classes.

5.3.1 Dictionnaire

La dernière classe que nous vous demandons d'implémenter doit permettre de représenter un dictionnaire. Usuellement, un dictionnaire permet de trouver la ou les définitions d'un mot (si, si ...). Partant de cette constatation, nous pouvons voir un dictionnaire comme une fonction associant à un mot un ensemble de définitions.

Dans cette optique, si on décide de représenter les "mots" et les "définitions" par des `String`, on peut construire une classe permettant de représenter des dictionnaires en étendant la classe représentant $Fonc(String, Ens(String))$.

Nous souhaitons, en outre, que cette classe, soit garnie de champs contentant certaines informations "redondantes", à savoir

- un dictionnaire orthographique correspondant au dictionnaire principal (il s'agira tout simplement d'un élément de $Ens(String)$),
- un dictionnaire des anagrammes correspondant au dictionnaire principal (il s'agira d'un élément de $Fonc(String, Ens(String))$).

Un dictionnaire doit disposer des "primitives" suivantes :

- créer un dictionnaire (vide),
- ajouter une définition dans un dictionnaire,
- "compléter" un dictionnaire (i.e. construire le dictionnaire orthographique et le dictionnaire des anagrammes),
- donner les définitions d'un mot dans un dictionnaire,
- vérifier l'orthographe d'un mot dans un dictionnaire,
- donner les anagrammes d'un mot dans un dictionnaire,
- donner l'ensemble des ensembles des sous-mots d'un mot dans un dictionnaire (chaque ensemble de sous-mots correspondant à une longueur fixée pour les mots lui appartenant : un ensemble pour les mots à 1 lettre, un ensemble pour les mots à 2 lettres, etc.).

Nous vous demandons de spécifier (spécification selon l'approche objet) et implémenter la classe décrite ci-dessus.

5.3.2 Programme de test

Nous vous demandons d'écrire un programme de test qui crée et remplit un dictionnaire "complet" à partir d'une instance quelconque de la classe `NimporteQuoi` (voir 5.3.3) et présente, ensuite, à l'utilisateur un menu textuel lui permettant d'exécuter à la demande chacune des fonctionnalités suivantes :

- afficher les définitions d'un mot introduit par l'utilisateur;
- vérifier l'orthographe d'un mot introduit par l'utilisateur;
- afficher les anagrammes d'un mot introduit par l'utilisateur;
- afficher tous les mots définis dans le dictionnaire;
- calculer et afficher l'ensemble des ensembles de sous-mots d'un mot introduit par l'utilisateur;
- quitter l'application.

Après l'exécution d'une fonctionnalité, le menu est réaffiché et l'utilisateur peut choisir une autre fonctionnalité.

5.3.3 `Vocabulaire`: les définitions sans peine

Dans notre grande bonté, nous vous fournissons des classes prédéfinies succeptibles de fournir sur demande une série de couples mot/définition.

Ces classes peuvent (doivent) être utilisées lors de la construction du dictionnaire intervenant dans le programme de test.

La spécification de ces classes, groupées dans le package `Vocabulaire`, est donnée ci-après.

Deux classes sont utilisées : `Definition` et `NimporteQuoi`. Une instance de la classe publique `Definition` représente un couple mot/définition. Outre les constructeurs, la classe est caractérisée par deux méthodes publiques (`getMot` et `getDefinition`) permettant d'obtenir respectivement le mot et la définition d'un couple.

```
package Vocabulaire;

public class Definition {
// Cette classe publique représente une définition de mot.
// Le mot et sa définition sont de type String.
```

```
    Definition();
    // Création d'une instance de la classe.

    Definition(String m, String d);
    // Création d'une instance de la classe à partir d'un mot
    // et de sa définition.

    public String getMot();
    // renvoie le mot défini.

    public String getDefinition();
    // renvoie la définition du mot.
}
```

Une instance de la classe publique `NimporteQuoi` renferme un certain nombre de définitions et fournit sur demande des instances d'objets de la classe `Definition`. Les définitions doivent être demandées séquentiellement par l'utilisateur de la classe. La classe fournit pour cela les méthodes définies ci-dessous.

```
package Vocalubaire;

public class NimporteQuoi{
// les instances de cette classe publique renferment des définitions et
// les fournissent séquentiellement sur demande.

    public NimporteQuoi();
    // Crée une instance et initialise les définitions renfermées.
    // La définition courante est la première (s'il y a au moins une
    // définition).

    public void init();
    // réinitialise la définition courante. Celle-ci sera alors la première
    // des définitions (s'il existe au moins une définition).

    public Definition getDefinition();
    // renvoie la définition courante et définit la définition courante
    // comme étant la définition suivante.
    // Précondition: la définition courante existe (on n'a pas lu
    // plus de définitions qu'il n'y en a).

    public boolean plusDeDefinition();
    // renvoie false si la définition courante existe (si on n'a pas lu
```

```
// plus de définitions qu'il n'y en a), true sinon.
```

5.4 La classe `String`

Comme vous l'avez remarqué, ce TP mentionne en de nombreux points la classe `String`. Les opérations prédéfinies sur cette classe, hormis l'égalité et la concaténation, ne sont pas reprises dans VTF. Cependant, pour réaliser ce TP, certaines opérations supplémentaires sur les `String` sont nécessaires. C'est pourquoi, nous vous fournissons l'extrait de la spécification de la classe `String` ci-après. Les méthodes reprises dans cet extrait devraient être suffisantes pour réaliser le TP.

Extrait de la spécification de la classe `String` du package `java.lang` :

```
public final class String {

public String()
This constructor initializes a newly created String object so that it
represents an empty character sequence.

public String(String value)
This constructor initializes a newly created String object so that it
represents the same sequence of characters as the argument; in other
words, the newly created string is a copy of the argument string.

public boolean equals(Object anObject)
The result is true if and only if the argument is not null and is a
String object that represents the same sequence of characters as this
String object. Overrides the equals method of Object.

public int length()
The length of the sequence of characters represented by this String
object is returned.

public int compareTo(String anotherString)
The character sequence represented by this String object is compared
lexicographically to the character sequence represented by the argument
string. The result is a negative integer if this String object
lexicographically precedes the argument string. The result is a positive
integer if this String object lexicographically follows the argument
string. The result is zero if the strings are equal; compareTo returns 0
exactly when the equals method would return true.
If anotherString is null, an error occurs.
```

```
public String substring(int beginIndex, int endIndex)
The result is a newly created String object that represents a subsequence
of the character sequence represented by this String object; this subsequence
begins with the character at position beginIndex and ends with the character
at position endIndex-1. Thus, the length of the subsequence is endIndex-beginIndex.
If beginIndex is negative, or endIndex is larger than the length of this String
object, or beginIndex is larger than endIndex, then an error occurs.

Examples:
"hamburger".substring(4, 8) returns "urge"
"smiles".substring(1, 5) returns "mile"

public String concat(String str)
If the length of the argument string is zero, then a reference to this String
object is returned. Otherwise, a new String object is created, representing a
character sequence that is the concatenation of the character sequence represented
by this String object and the character sequence represented by the argument string.
If str is null, an error occurs.

Examples:
"cares".concat("s") returns "caress"
"to".concat("get").concat("her") returns "together"
}
```

5.5 Modalités

Et, pour terminer, quelques informations “pratiques” ...

5.5.1 Contenu du rapport

Voici, en résumé et schématiquement, ce que nous désirons trouver dans votre rapport.

- Spécifications des Types Logiques
  - Ensemble ordonné : type abstrait et spécification selon l’approche objet
  - Fonction de domaine fini : modèle de haut niveau et spécification selon l’approche objet
- Représentation/Implémentation des Types Logiques
  - Conventions de représentation

  - Justification des choix de représentation pour les ensembles
  - Description globale de votre hiérarchie de classes
  - Définitions supplémentaires (ordres induits, par exemple)
  - Critiques/commentaires relatifs à votre implémentation
- Application
  - Spécification de la classe “Dictionnaire”
  - Critiques/commentaires
- Commentaires “pratiques”
  - Découpe en packages
  - Description du programme de test
- Listing complet du code documenté (chaque méthode doit être garnie d’une précondition et d’une postcondition).

5.5.2 Format des fichiers

Au niveau des fichiers, nous désirons recevoir uniquement les fichiers de code (i.e. les fichiers `.java`). Nous vous demandons de nous remettre un fichier “zippé” contenant tous ces fichiers et rien qu’eux.

Ce fichier doit porter un nom de la forme `JavaG??.zip` où `??` représente le numéro de votre groupe. L’extraction de ce fichier doit engendrer dans le répertoire courant la création d’un répertoire `G??` contenant vos fichiers.

Votre programme de test doit porter le nom `TestG??.java`.

Au niveau de la découpe en package, nous vous demandons d’inclure vos différents package et classes dans un package nommé `G??`. La classe du programme de test se trouvera au premier niveau de ce package.

5.5.3 Divers

Les codes peuvent être remis soit par mail soit sur disquette.

Si vous optez pour le mail, veillez à placer la séquence `TP Orienté Objet (Charleroi)` dans le “subject” du mail.

Tous les documents relatifs à ce TP sont disponibles à l’adresse
`http://www.info.fundp.ac.be/~mpe/ParadigmeOO`.

6 Petits exercices sur AWT

Exercice 1

Ecrire une classe java permettant de créer une fenêtre présentant un seul bouton et telle que cliquer sur ce bouton provoque la fermeture de cette fenêtre.

Ecrire un petit programme test créant trois fenêtres de ce type.

Exercice 2

Ecrire une classe java permettant de créer une fenêtre présentant un seul bouton et telle que

- cliquer sur la "croix de fermeture" provoque la fermeture de la fenêtre,
- cliquer sur le bouton provoque la création (et l'affichage) d'une autre fenêtre présentant, quant à elle, un champ de texte et un label et telle que
  - cliquer sur la "croix de fermeture" provoque la fermeture de la fenêtre,
  - rentrer un mot dans le champ de texte provoque son "affichage" dans le label.

Ecrire un programme de test créant un fenêtre du premier type.

Une première variante de cet exercice doit permettre l'existence simultanée d'un nombre quelconque de fenêtres du second type (créées à partir de la première fenêtre), tandis qu'une seconde variante permettera l'existence, à un moment donné, d'au plus une fenêtre du second type.

Exercice 3

L'exercice consiste à écrire quelques classes Java réalisant une interface pour la mini-application décrite ci-après.

La mini-application demandée "gère" une liste de String et offre deux fonctionnalités :

- ajout d'un String dans la liste,
- affichage de la liste.

L'interface de cette mini-application est réalisée via trois fenêtres.

1. Une première fenêtre (appelée fenêtre initiale) présente trois boutons : un bouton permettant de quitter l'application, un bouton permettant d'appeler la "fenêtre d'ajout" et un bouton permettant d'appeler la "fenêtre d'affichage".

2. Une deuxième fenêtre (appelée fenêtre d'affichage) contient une liste présentant le contenu "actuel" de la liste gérée par l'application.

3. Une troisième fenêtre (appelée fenêtre d'ajout) contient un champ de texte permettant l'introduction d'un nouveau mot dans la liste gérée par l'application.

Cette mini-application utilise la classe `ListeString` du package `Liste` qui permet la représentation et le parcours de listes de String et vérifie la spécification ci-après

Le parcours d'une liste est réalisé via un "élément courant". Cet "élément courant", s'il est initialisé, correspond soit à un élément de la liste soit la valeur spéciale $finDeliste$ qui indique la fin du parcours de la liste.

`ListeString()`
Construit une liste vide de String.

`void debParcours()`
Pré :
Post : Le contenu de la liste est inchangé et l'élément courant correspond au premier élément de la liste si celui-ci existe et à $finDeListe$ sinon.

`String getString()`
Pré : L'élément courant, `e`, est initialisé et est différent de $finDeListe$.
Post : Le contenu de la liste est inchangé; l'élément courant est le successeur de e_0; `getString` $= e_0$.

`boolean finDeParcours()`
Pré : L'élément courant est initialisé.
Post : Le contenu de la liste est inchangé; l'élément courant est inchangé;
`finDeparcours` $= vrai$ si et seulement si l'élément courant correspond à $finDeliste$.

`void ajoutString(String s)`
Pré : le contenu de la liste est $(v_1, ..., v_n)$
Post : le contenu de la liste est $(s, v_1, .., v_n)$

7 TP : seconde partie

La seconde partie du TP consiste à implémenter une interface graphique simple relative à la classe `Dico` réalisée lors de la première partie du TP. On utilise (contrairement à ce qui avait été anoncé) pour cela un sous-ensemble des classes graphiques de Java AWT (Abstract Window Toolkit) en combinaison avec VTF.

L'interface graphique peut différer d'un groupe à l'autre selon les goûts de chacun. Un certain nombre de contraintes sont toutefois imposées. Elles sont décrites dans la section 7.1. Afin de réaliser cette interface, vous aurez besoin d'éléments non vus au cours. Ceux-ci sont décrits à la section 7.2.

7.1 Description de l'interface

L'interface doit être composée de trois fenêtres : une fenêtre initiale et deux fenêtres relatives aux fonctionnalités demandées.

Lors du démarrage du programme, le dictionnaire est initialisé à partir du package `Vocabulaire` décrit à la section 5.3.3 et la fenêtre initiale est affichée.

7.1.1 Fenêtre initiale

La fenêtre principale doit contenir trois boutons permettant :

- d'ajouter des couples "(mot, définition)" dans le dictionnaire (une fenêtre permettant cette opération est alors affichée);
- de consulter le dictionnaire (une fenêtre permettant cette opération est alors affichée);
- de quitter l'application.

7.1.2 Fenêtre d'ajout

La fenêtre d'ajout d'un mot au dictionnaire doit contenir :

- un champ de texte pour saisir le mot;
- un champ de texte pour saisir la définition;

- un bouton permettant de valider le mot et sa définition (afin qu'ils soient ajoutés au dictionnaire) et d'introduire le mot et la définition suivants;
- un bouton permettant d'abandonner la saisie en cours et de retourner à la fenêtre principale.

7.1.3 Fenêtre de consultation

La fenêtre de consultation doit contenir :

- une liste contenant tous les mots définis dans le dictionnaire et permettant de sélectionner un mot dont on souhaite consulter les définitions;
- une zone de texte dans laquelle les définitions du mot sélectionné seront affichées une par une;
- un bouton permettant d'afficher la première définition du mot sélectionné dans la zone de texte;
- un bouton permettant d'afficher la définition suivante du mot sélectionné dans la zone de texte;
- un bouton permettant de quitter la fenêtre et de retourner à la fenêtre principale.

7.2 Classes AWT

Outre les classes et méthodes d'AWT présentées au cours, nous vous permettons d'utiliser les éléments suivants.

7.2.1 Composant graphique `List`

Voici l'extrait de la spécification du composant dont vous aurez besoin.

```
List extends Component

List()
     Creates a new scrolling list.
add(String)
     Adds the specified item to the end of scrolling list.
addActionListener(ActionListener)
```

```
     Adds the specified action listener to receive action events
     from this list.
getSelectedItem()
     Get the selected item on this scrolling list.
removeAll()
     Removes all items from this list.
```

7.2.2 Composant graphique `TextField`

La méthode suivante est ajoutée à la spécification donnée.

```
getText()
     Gets the text that is presented by this text field.
```

7.2.3 Composant graphique `TextArea`

Voici l'extrait de la spécification du composant dont vous aurez besoin.

```
TextArea inherits Component

TextArea()
     Constructs a new text area.
TextArea(int, int)
     Constructs a new empty TextArea with the specified
     number of rows and columns.
TextArea(String)
     Constructs a new text area with the specified text.
setText(String)
     Sets the text that is presented by this text component to
     be the specified text.
```

7.2.4 Layout graphiques

Nous vous permettons d'utiliser le layout supplémentaire : `GridLayout`. Voici un extrait suffisant de la spécification de celui-ci.

```
GridLayouts simply make a bunch of Components have equal size,
displaying them in the requested number of rows and columns. A
GridLayout places components in a grid of cells. Each component
takes all the available space within its cell, and each cell is
```

```
exactly the same size. If you resize the GridLayout window, you'll
see that the GridLayout changes the cell size so that the cells
are as large as possible, given the space available to the
container.

GridLayout(int, int)
     Creates a grid layout with the specified number of rows and columns.
GridLayout(int, int, int, int)
     Creates a grid layout with the first two specified number of rows
     and columns. The last two numbers specify the number of blank
     pixels between two adjacent cells (horizontal and vertical "gaps").
```

Un GridLayout dessine les composants graphiques du container auquel il est attaché en les rangeant dans les cellules dans l'ordre dans lequel ils ont été ajoutés au container, en remplissant complètement chaque ligne avant de passer à la suivante.

En ce qui concerne les autres layout présentés au cours (`CardLayout`, `FlowLayout` et `BorderLayout`), vous pouvez utiliser les constructeurs suivants.

```
CardLayout()
     Creates a new card layout with gaps of size zero.
CardLayout(int, int)
     Creates a new card layout with the specified
     horizontal and vertical gaps.

FlowLayout()
     Constructs a new Flow Layout with a centered alignment
     and a default 5-unit horizontal and vertical gap.
FlowLayout(int)
     Constructs a new Flow Layout with the specified alignment
     and a default 5-unit horizontal and vertical gap.
FlowLayout(int, int, int)
     Creates a new flow layout manager with the indicated alignment
     and the indicated horizontal and vertical gaps.

The value of the alignment argument must be one of FlowLayout.LEFT,
FlowLayout.RIGHT, or FlowLayout.CENTER.

BorderLayout()
     Constructs a new border layout with no gaps between components.
BorderLayout(int, int)
     Constructs a border layout with the specified gaps between
     components.
```

7.2.5 Errata

Voici deux erreurs présentes dans le document donné au cours théorique.

1. La méthode `setActionCommand(String)` n'est pas définie pour la classe `TextField`.
2. La classe `WindowAdapter` n'est pas abstraite et, par conséquent, la méthode `windowClosing(WindowEvent)` ne peut être abstraite. Il s'agit en fait d'une méthode vide. Le type du paramètre fourni pour cette méthode était en outre erroné.

7.3 Modalités

Et, pour terminer, quelques informations "pratiques" ...

7.3.1 Contenu du rapport

Le rapport doit contenir une description des différentes classes supportant l'interface graphique proposée et décrire les interactions entre ces classes.

Vous pouvez évidemment ajouter à cela toute critique que vous jugerez pertinente ...

Le rapport contiendra également Listing complet du code documenté.

7.3.2 Format des fichiers

Au niveau des fichiers, nous désirons recevoir uniquement les fichiers de code (i.e. les fichiers `.java`). Nous vous demandons de nous remettre un fichier "zippé" contenant tous ces fichiers et rien qu'eux.

Ce fichier doit porter un nom de la forme `GraphJavaG??.zip` où `??` représente le numéro de votre groupe. L'extraction de ce fichier doit engendrer dans le répertoire courant la création d'un répertoire `G??` contenant vos fichiers.

Votre programme de test doit porter le nom `TestGraphG??.java`.

Théoriquement, vous ne devriez plus avoir à modifier les classes relatives à la première partie du TP, cependant, comme certains groupes, n'avaient pas terminé l'implémentation de la classe `Dico`, nous vous demandons d'intégrer dans le fichier `GraphJavaG??.zip`, les classes relatives à la première partie du TP.

Au niveau de la découpe en package, nous vous demandons d'inclure vos différents package et classes dans un package nommé `G??`. Les classes relatives à la partie graphique se trouveront dans un "sous-package" différent de celui (ou de ceux) regroupant les classes relatives à la première partie du TP. La classe du programme de test se trouvera au premier niveau de du package `G??`.

7.3.3 Divers

Les codes peuvent être remis soit par mail soit sur disquette.

Si vous optez pour le mail, veillez à placer la séquence `TP Orienté Objet` dans le "subject" du mail.