Correction de l'examen de programmation par objets

Licence informatique - UE2241MBE – Janvier 2004

Responsable J.Ferber – Durée 2 h

Documents manuels (cours, polys, TD, TP) autorisés.
Livres et documents Web interdits.

Correction version 1.0

Attention: cette correction ne prend en compte que les deux premiers exercices

Exercice 1: Héritage et instanciation

Le graphe de classes (graphe UML) présenté figure 1 correspond à une hiérarchie de classes. L'hexagone I représente une interface.

Question : Dans la suite des instructions, indiquez pour chacune d'elles: 1) si elle passe ou si elle provoque une erreur à la compilation. 2) si on peut résoudre le problème de la compilation par un cast (et donnez ce cast), et si oui, si il reste une erreur à l'exécution ou non.

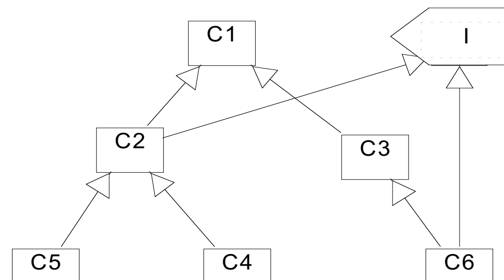


Figure 1

	Compilation	Correction	Execution
C1 a1 = new C1()	OK		
I a2 = new C4()	OK		
I a3 = new C6()	OK		
C3 a4 = new C6()	OK		
C3 a5 = a3	Erreur	C3 a5 = (C3) a3	OK
a5 = a4	OK		
a2 = a3	OK		
C2 a6 = new C2()	OK		
a4 = a6	Erreur	Pas de cast possible	
C4 a7 = a6	Erreur	C4 a7 = (C4) a6	Erreur
a6 = a2	Erreur	a6 = (C2) a2	Erreur

Notes:

a) S'il n'y a pas de cast, alors si OK à la compilation implique nécessairement OK à l'exécution.

b) Les interfaces sont des types. On ne peut pas faire I a2 = new I(), puique les interfaces n'ont pas d'instances, mais on peut faire I a2 = new C4(). De même a2 = a3 est OK puisque les types des variables correspondent. Il n'y a pas non plus d'erreur à l'exécution.

c) Pour a4 = a6, il n'y a pas de cast possible puisque C3 et C2 ne sont pas comparables.

d) Si on met un cast il peut y avoir une erreur à l'exécution. Cela est vrai pour les 2 dernières lignes.

Exercice 2: Interface graphique et événements

On voudrait réaliser un composant textuel **AfficheurTexte** qui hérite de **TextArea**. On rappelle que la particularité d'un **TextField** c'est d'avoir aussi deux méthode, **setText** et **getText** qui fonctionnent comme pour **TextArea** (cf ci-dessous) mais aussi d'invoquer la méthode **actionPerformed(ActionEvent e)** de tout 'listener' de ce **TextField**).

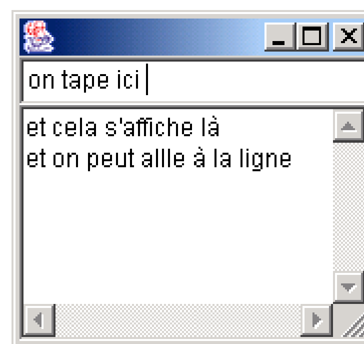


Figure 2

On désire réaliser l'interface graphique présentée figure 2.

Elle se compose d'une classe **TestAfficheur** qui hérite de **Frame** et de deux composants: un **TextField** (le composant du haut) et un **AfficheurTexte** (le composant du milieu).

Question 1: donnez le code de la classe **AfficheurTexte** de telle manière que l'on puisse écrire la définition des composants ainsi:

```
TextField tf = new TextField();
AfficheurTexte at = new AfficheurTexte();
tf.addActionListener(at);
```

Question 2: Donnez le code intégral de la classe **TestAfficheur** qui réalise l'interface graphique de la figure 2.

Note1: Un TextArea est un composant AWT qui implémente un petit éditeur de texte. Il est caractérisé par trois méthodes : **String getText()** qui retourne le contenu du **TextArea**, **void setText(String t)** qui écrase le contenu du **TextArea** avec la chaîne t, et **void append(String t)** qui ajoute la chaîne t à la fin du contenu du **TextArea**.

```java
class AfficheurTexte extends TextArea implements ActionListener {
	public void actionPerformed(ActionEvent e){
		TextField tf = (TextField) e.getSource();
		this.append(tf.getText()+'\n');
	}
}

public class TestAfficheur extends Frame {

	public static void main(String[] args){
		Frame f = new TestAfficheur();
		f.show();
	}

	TestAfficheur(){
		TextField tf = new TextField();
		AfficheurTexte at = new AfficheurTexte();
		tf.addActionListener(at);
		add(tf, BorderLayout.NORTH);
```

```
            add(at, BorderLayout.CENTER);
            setSize(412,312);
        }
}
```

Note : c'est le AfficheurTexte qui implémente le ActionListener et non la Frame comme on le fait souvent.

Exercice 3 : Bataille navale

Remarque générale : cet exercice sera noté en fonction de la qualité de la conception *objet* du logiciel. En particulier vous n'avez pas le droit d'utiliser l'opérateur **instanceof**.

On voudrait réaliser un jeu de bataille navale. Un jeu de bataille navale se compose d'un tableau et d'un ensemble de bateaux, chaque bateau se compose d'un ensemble de taille fixe d'élements. Un croiseur comprend 3 éléments, un escorteur 2 et un sous-marin un seul élément. Chaque élément est caractérisé par sa position et par son état: sain ou touché. Les sous-marins ont la possibilité de plonger. Lorsqu'ils plongent ils ne peuvent pas être touchés.

Un tableau contient un ensemble de bateaux. Un bateau est caractérisé par l'ensemble de ses éléments. Voici comment on instancie une flotte de bateaux (qui correspond à la figure 3):

Bateau b1 = new Croiseur(1,1, true); // un croiseur horizontal dont le premier élément est en 1, 1 (les coordonnées ont leur origine en 0,0).

Bateau b2 = new Escorteur(2,5,false); // un escorteur vertical dont le premier élément est en 2,5

Bateau b3 = new SousMarin(4,2,true); // un sous-marin en 4,2 et en surface (false = en plongée).

Tableau t1 = new Tableau(7,9);

t1.ajouterBateau(b1);

t1.ajouterBateau(b2);

t1.ajouterBateau(b3);

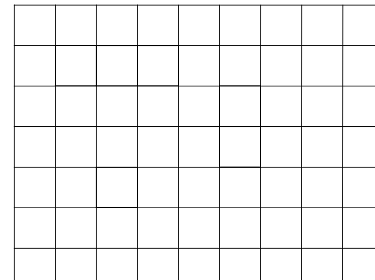


Figure 3

Question 1: donnez le diagramme UML (diagramme de classes) de cet exemple.

Question 2: donnez le code des classes ainsi définies (définition avec les attributs, les constructeurs et les méthodes **ajouterBateau** et **enleverBateau** dans Tableau).

Question 3: donnez le code de la méthode **int estTouche(int px, int py)** définie dans **Bateau** (et éventuellement dans ses sous-classes) qui retourne 0 lorsque le coup est dans l'eau, 1 si le coup tombe sur un élément touché, 2 si le coup tombe sur un élément touché pour la première fois, et 3 si le bateau est coulé (tous les éléments sont touchés). Si c'est un sous-marin en plongée, la méthode retourne toujours 0.

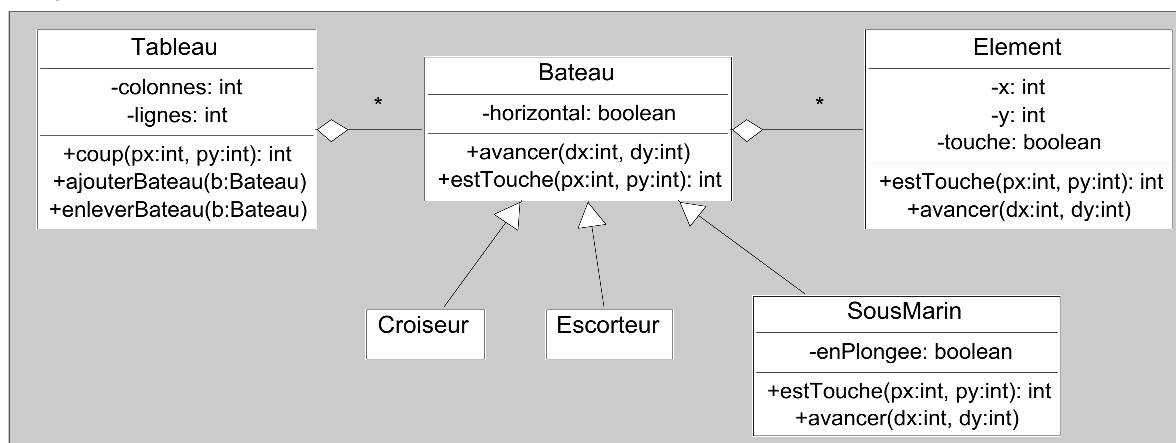
Note: vous pouvez ajouter des méthodes auxiliaires si nécessaires.

Question 4: donnez le code de la méthode **int coup(int px, int py)** définie dans **Tableau** qui retourne -1 si le coup est en dehors du tableau, 0 lorsque le coup est dans l'eau, 1 si le coup tombe sur un élément déjà touché, 2 si le coup tombe sur un élément touché pour la première fois et 3 si le bateau est coulé. Si le bateau est coulé, il est alors supprimé du tableau.

Question 5: on suppose que les bateaux peuvent avancer, sauf lorsque les sous-marins sont en plongée. Ecrire la méthode **boolean avancer(int dx, int dy)** dans **Bateau** (et éventuellement dans ses sous-classes) qui fait avancer le bateau d'une case dans la direction définie par dx et dy (la valeur de dx et dy est seulement de -1 ou 1, ce qui implique que les bateaux peuvent avancer en diagonale.

Voici une des corrections possibles (toutes les classes sont données ensembles). Lisez les notes et recommendations à la fin du fichier.

Diagramme UML




```
class Element {
        int x, y;
        boolean touche=false;
        Element(int x, int y){
                this.x = x; this.y = y;
        }

        void avancer(int dx, int dy){
                x = x+dx;
                y = y+dy;
        }
        int estTouche(int px, int py){
                if ((x == px) && (x == py)){
                        if (touche)
                                return 1;
                        else {
                                touche = true;
                                return 2;
                        }
                } else return 0;
        }
}

class Bateau {
        boolean horizontal=true;
        Element[] elements;
        Bateau(int x, int y, boolean horiz, int taille){
                horizontal=horiz;
                elements=new Element[taille];
                for(int i=0; i<taille;i++){
                        if (horizontal)
                                elements[i]=new Element(x+i,y);
                        else
                                elements[i]=new Element(x,y+i);
                }
```

```
	}

	int estTouche(int px, int py){
		int cpt = 0; int res = 0;
		for (int i=0;i<elements.length;i++){
			int r = elements[i].estTouche(px,py);
			if (r == 1) {
				res = 1; cpt++;
			}
			else if (r == 2) {
				res = 2; cpt++;
			}
		}
		if (cpt == elements.length)
			return 3;
		else
			return res;
	}
	void avancer(int dx, int dy){
		for (int i=0;i<elements.length;i++)
			elements[i].avancer(dx,dy);
	}
}

class Croiseur extends Bateau {
	Croiseur(int x, int y, boolean horiz){
		super(x,y,horiz,3);
	}
}

class Escorteur extends Bateau {
	Escorteur(int x, int y, boolean horiz){
		super(x,y,horiz,2);
	}
}

class SousMarin extends Bateau {
	boolean enPlongee;
	SousMarin(int x, int y, boolean plongee){
		super(x,y,true,1);
		enPlongee = plongee;
	}
	int estTouche(int px, int py){
		if (enPlongee)
			return 0;
		else return (super.estTouche(px,py));
	}
```

```
        void avancer(int dx, int dy){
                if (!enPlongee){
                        super.avancer(dx,dy);
                }
        }
}

class Tableau {
        int colonnes;
        int lignes;
        Vector bateaux;

        Tableau(int l, int c){
                bateaux = new Vector();
        }

        void ajouterBateau(Bateau b){
                if (!bateaux.contains(b))
                        bateaux.add(b);
        }

        void enleverBateau(Bateau b){
                bateaux.remove(b);
        }

        int coup(int px, int py){
                if ((px < 0) || (px >= colonnes) || (py < 0) || (py >= lignes))
                        return 0;
                else {
                        for (int i=0;i<bateaux.size();i++){
                                Bateau b = (Bateau) bateaux.get(i);
                                int r = b.estTouche(px,py);
                                if (r != 0){
                                        if (r == 3) enleverBateau(b);
                                        return r;
                                }
                        }
                        return 0;
                }
        }
}
```

Variante

- On peut aussi ne pas considérer les éléments comme des entités et tout gérer à partir d'un tableau de booléens associé aux bateaux.

- Ici on calcule à chaque appel de **estTouche** le fait qu'un bateau soit coule ou non. On peut le calculer une fois pour toutes et laisser cette information dans un attribut '**boolean coule**' dédié à cette effet.

Notes

- Notez le fait que l'on utilise pleinement la classe **Element**. La tendance des débutants en objet est de considérer de telles classes comme de simples structures de données passives. Au contraire, c'est en gérant pleinement la classe **Element** comme une entité à part entière que l'on simplifie des méthodes telles que **estTouche** ou **avancer**.
- Les méthodes **estTouche** et **avancer** sont redéfinies dans **SousMarin** et font appel au code standard. Cela simplifie le code.

Recommandations

Voici un ensemble de recommandations qui s'adressent non seulement à ceux qui n'ont pas réussi l'examen, mais aussi à ceux qui l'ont passé. Car pratiquement aucun étudiant n'a réellement appliqué ces recommandations dans leur intégralité.

1. L'écriture d'un programme est un art (au sens ancien du métier artisanal) qui réclame un certain « tour de main » (même si ce tour de main est évidemment très intellectuel en informatique). La programmation est certainement l'une des rares activités industrielle qui s'exprime encore comme un art (au même titre qu'un architecte par exemple). Comme dans toute qualité nécessitant un tour de main (pensez aux compagnons du tour de France) un travail bien fait se reconnaît au fait qu'il a été conçu intelligemment (et non pas codé à toute vitesse dans tous les sens) et que sa structure présente une certaine harmonie. Utilisez votre sens esthétique pour déterminer si un programme est correct ou non : un « beau » programme a plus de chance de tourner et d'être évolutif que du code jeté à la va vite..
2. Il faut que le code des classes soit cohérent avec le diagramme. Pensez aussi à l'implémentation des relations. Pour une association ou une agrégation avec une cardinalité '*', utilisez un Vector. Si au contraire le nombre des éléments est fixe, utilisez un tableau.
3. Il faut toujours rechercher la généricité dans un programme. Ecrivez toujours un programme comme si vous deviez le modifier et l'étendre le lendemain pour lui ajouter des fonctionnalités. En particulier : évitez les structures de données inutiles, toutes les techniques 'ad hoc' que l'on n'utilise que pour résoudre un tout petit cas particulier.
4. Entre autre : ne JAMAIS mettre des constantes en dur dans le code ! (dans l'exemple précédent, ne pas faire des choses du genre : for(int i=0 ;i < 7 ; i++). Le code ne sera jamais extensible…
5. Recherchez la simplicité et la lisibilité de votre programme. Un bon programme est un programme simple. Moins il y a de lignes de code, tout en étant lisible (pas d'astuces qui diminuent en fait la lisibilité du code), plus votre programme a de chances d'être extensible et donc rapidement fonctionnel. La simplicité et la lisibilité sont les clefs de la bonne programmation.
6. En programmation objet : toujours rechercher à utilisez la hiérarchie des classes et l'envoi de message. C'est un mécanisme très puissant. En particulier n'utilisez JAMAIS un champ 'type' (ou 'nature' ou 'sorte') avec des méthodes qui font des tests sur ces types, utilisez toujours l'héritage à la place.
7. Ne pas confondre héritage avec composition. L'héritage répond à la question 'sorte de' (un chien est une sorte de animal), alors que l'agrégation (et la composition) répond à la question 'partie de' (la tête d'un chien est une partie de ce chien).
8. Essayez que vos objets agissent par eux mêmes : pensez que chaque objet est un petit être qui agit par lui même. N'agissez pas sur les objets comme s'il s'agissait de structures de données passives, demandez aux objets d'agir !

J'espère que ces recommendations vous permettront de mieux réussir dans vos examens futurs, mais aussi tout simplement d'écrire des programmes mieux conçus et plus « objet ».