SCIENCES SORBONNE UNIVERSITÉ

Examen en seconde session de 3I002
Licence 3 d'informatique

C. Constantin, B. Lefebvre, J. Lejeune, A. Miné, D. Mercadier, Y. Thierry-Mieg

Juin 2018

2 heures – **Tout document papier autorisé**
Tout appareil électronique interdit

- Le barème est sur 23 points et est donné à titre indicatif. La note finale sera bornée à 20 points.
- Dans le code Java demandé vous ne gérerez pas les imports.
- Lors de la correction il sera tenu compte de la simplicité et de lisibilité de vos réponses.
- Il vous est conseillé de lire attentivement les exercices entièrement avant de composer.

L'objectif de ce sujet est de programmer un mini-jeu de stratégie simplifié de type « StarCraft » ou « Age of Empire » où les joueurs doivent gérer des unités (soldats, bâtiments, citoyens, ...) sur une carte. Une carte est vue comme un ensemble de cases placées selon des coordonnées x et y. Chaque unité du jeu occupe une case. Une case peut accueillir au plus une unité.

Interfaces. Pour implanter notre jeu, nous aurons besoin des interfaces suivantes :

- L'interface `IJoueur` qui représente un joueur de la partie. Un joueur possède un nom de type `String` faisant office d'identifiant et une collection d'unités qu'il contrôle.
- L'interface `IUnite` représente une unité de la partie. Elle possède un identifiant de type `String`, une référence vers la carte qui l'héberge, une références sur la case sur laquelle elle est positionnée, la référence vers son joueur propriétaire.
- L'interface `ICarte` représente une carte de la partie et se caractérise par une ordonnée maximale `maxY` et une abscisse maximale `maxX`, et des cases de coordonnées x entre 0 et `maxX` et y entre 0 et `maxY` (inclus).
- L'interface `ICase` représente une case de la carte. Elle possède comme propriétés ses coordonnées x et y dans la carte et une référence vers son unité occupante (ou `null` si la case est vide).

Le détail de ces interfaces est donné en figure 1. Notez que certaines méthodes (comment `getVoisins` et `bestPath`) n'ont pas été expliquées ci-dessus et seront décrites au fur et à mesure dans le reste de l'énoncé.

Exercice 1 – Les joueurs

Notions testées : Classe, map, délégation, itérateur

On souhaite coder une classe `Joueur` qui implante l'interface `IJoueur` :

- Le constructeur prendra comme seul argument le nom du joueur, de type `String`.
- Le nom du joueur doit être un attribut constant, initialisé au moment de la construction.
- On stockera les unités du joueur à l'aide d'une map, initialement vide, où la clé sera l'identifiant de l'unité et la valeur sera l'unité elle-même. On utilisera exclusivement des références de type `IUnite` pour référencer des unités (programmation vis à vis d'une interface).

```java
public interface IJoueur {
  String getNom();
  void putUnite(IUnite unite);
  IUnite removeUnite(String idUnite);
  IUnite get(String id);
}
```

```java
public interface IUnite {
  ICarte getCarte();
  String getID();
  IJoueur getProprietaire();
  ICase getPosition();
  void setPosition(ICase c);
}
```

```java
public interface ICase {
  int getX();
  int getY();
  IUnite getOccupant();
  void setOccupant(IUnite u);
  void removeOccupant();
  Collection<ICase> getVoisins();
}
```

```java
public interface ICarte {
  ICase get(int x, int y);
  int getMaxX();
  int getMaxY();
  List<ICase> bestPath(ICase from,
      ICase to);
}
```

FIGURE 1 – Interfaces.

- La méthode `removeUnite` doit respecter le même contrat que la méthode `remove` de l'interface `Map`, c'est à dire qu'elle renvoie une référence sur l'unité supprimée ou `null` si cette dernière n'était pas présente dans la map.

Question 1 1½ points

Donnez le code de la classe `Joueur`.

Solution:

```java
public class Joueur implements IJoueur{

	private final Map<String,IUnite> unites = new HashMap<>();
	private final String nom;

	public Joueur(String n) {
		this.nom=n;
	}

	public String getNom() { return nom;}


	public void putUnite(IUnite unite) {
		unites.put(unite.getID(), unite);
	}

	public IUnite removeUnite(String id) {return unites.remove(id);}

	@Override
	public IUnite get(String id) {return unites.get(id);}

```

```
}
```

Solution:
Barème :
- 20% `implements IJoueur`
- 10% déclaration des attributs
- 10% **`final`** sur nom
- 20% constructeur + instanciation correcte de la map
- 20% `putUnite`
- 10% `removeUnite`
- 10% `get`

Dans la suite, on souhaiterait pouvoir ajouter la capacité à un joueur d'itérer sur ses unités afin que le code suivant soit compilable :

```
IJoueur j = ....
for(IUnite u : j){
	//traitement sur u
}
```

Question 2 1 point

Que faut-il faire sur l'interface `IJoueur` pour que cela soit possible ? Modifiez en conséquence la classe `Joueur`. Vous renverrez un itérateur sur une nouvelle liste construite à partir des valeurs de la map.

Solution:
rajouter un **`extends`** `Iterable<IUnite>` dans `IJoueur`.
Dans la classe `Joueur`

```
@Override
	public Iterator<IUnite> iterator() {
		return new ArrayList<IUnite>(unites.values()).iterator();
	}
```

Barème :
- 25% pour **`extends`** `Iterable<IUnite>` dans `IJoueur`.
- 25% signature de la méthode `iterator()`
- 25% construction d'une nouvelle liste
- 25% délégation sur un iterator de la liste

N.B : on considérera par la suite que les modifications ont été correctement faites.

Exercice 2 – Les unités

Notions testées : héritage, extension d'interface, DP Factory

Nous allons dans un premier temps écrire une classe `Unite` implantant l'interface `IUnite`.
Lors de l'appel au constructeur `Unite(ICarte carte, IJoueur proprietaire, ICase position)` :

- on passera en argument la carte, le propriétaire de l'unité et la position initiale ;
- on construira l'identifiant de l'unité à partir du nom de son propriétaire, suivi d'une valeur entière qui sera calculée grâce à un compteur statique incrémenté à chaque appel du constructeur ;
- l'instance créée sera affectée au propriétaire (c'est à dire, ajoutée à la liste des unités gérées par le propriétaire) ;
- la case initiale passée en paramètre sera supposée vide avant l'appel au constructeur, et aura comme occupant l'instance créée quand le constructeur retourne.

Question 1 1½ points

Donnez le code de la classe `Unite`. Vous donnerez le code de tous les attributs, du constructeur et de la méthode `setPosition`. Le code des getters n'est pas demandé. La méthode `setPosition` change la position de l'unité. Vous prendrez garde à mettre à jour la case correspondant à l'ancienne position, qui devient vide, et celle de la nouvelle position, qui est supposée vide avant l'appel à `setPosition` et qui doit contenir l'unité au retour de la méthode.

Notez qu'une unité garde une référence vers sa position, tandis que la position garde une référence vers l'unité. Il faut à tout moment (notamment après la construction et après `setPosition`) respecter l'invariant suivant : pour toute référence non nulle u de type `IUnite` et c de type `ICase` :

$$\texttt{u.getPosition()} == \texttt{c} \Leftrightarrow \texttt{u} == \texttt{c.getOccupant()}$$

Solution:

```java
public class Unite implements IUnite {

	private static int cpt=0;
	private ICase position;;
	private IJoueur proprietaire;
	private final String id;
	private final ICarte carte;


	public Unite(ICarte carte, IJoueur proprietaire, ICase pos_init) {

		this.proprietaire=proprietaire;
		id=proprietaire.getNom()+"_unite_"+(cpt++);
		this.carte=carte;
		proprietaire.putUnite(this);
		setPosition(pos_init);
	}

	@Override
	public void setPosition(ICase c) {
		if(this.position != null) {//peut être null àla construction
			this.position.removeOccupant();
		}
		this.position=c;
		c.setOccupant(this);
	}

	//getters

}
```

Solution:
Barème :

10%	pour `implements IUnite`
20%	déclaration des attributs d'instances (hors `cpt`)
25%	déclaration compteur statique + construction correcte de l'id
25%	constructeur + initialisation correcte de l'état + attribution de l'unité au joueur
20%	méthode `setPosition` (dont 10% sur l'appel à `setOccupant` de la case)

Question 2 1 point

On suppose l'existence d'une classe `UtilTest` qui offre les méthodes suivantes :

- `public static ICarte getCarteTest()` qui renvoie une instance sur une carte de test. La carte renvoyée ne contient aucune unité.
- `public static IJoueur getJoueurTest()` qui renvoie une instance de joueur de test.

Donnez une classe de test JUnit nommée `TestInvariant` qui crée une unité pour le joueur de test en position (0,0) de la carte de test puis la déplace en position (1,1), tout en vérifiant que l'invariant ci-dessus est bien respecté.

Solution:

```
public class TestInvariant {

	@Test
	public void test() {
		ICarte carte =UtilTest.getCarteTest();
		IJoueur joueur = UtilTest.getJoueurTest();
		ICase pos1 = carte.get(0, 0);
		IUnite unite = new Unite( carte,joueur,pos1);
		assertTrue(unite.getPosition() == pos1);
		assertTrue(pos1.getOccupant() == unite);

		ICase pos2 = carte.get(1, 1);
		unite.setPosition(pos2);
		assertTrue(unite.getPosition() == pos2);
		assertTrue(pos2.getOccupant() == unite);
		assertTrue(pos1.getOccupant() == null);

	}
}
```

Solution:
Barème :

10%	pour `@Test`
20%	bonne initialisation du joueur
20%	changement de position
50%	pour les assertTrue (10% chacun)

On souhaite maintenant avoir des unités capables de se déplacer sur la carte. Pour cela nous allons définir une nouvelle interface `IMovableUnite` qui étend l'interface `IUnite`. Une unité mobile possède,

en plus des attributs d'une unité classique, un attribut entier indiquant sa vitesse et un attribut `ICase` indiquant sa destination. L'interface `IMovableUnite` devra offrir :

- un getter `getVitesse` renvoyant la vitesse de déplacement de l'unité ;
- un getter `getDestination` qui renvoie la case destination de l'unité ;
- une méthode `move` qui prend en argument une case destination et déplace l'unité vers cette case ;
- une méthode `stop`, sans argument, qui arrête le déplacement de l'unité.

On considérera qu'une `IMovableUnite` est immobile si sa position courante est égale à sa destination.

Question 3 1½ points

Donnez le code de l'interface `IMovableUnite`. Vous donnerez un comportement par défaut à la méthode `stop` en appelant la méthode `move` paramétrée avec la position courante.

Solution:

```
public interface IMovableUnite extends IUnite {

	public void move(ICase destination);

	public int getVitesse();

	public ICase getDestination();

	default public void stop() {
		move(getPosition());
	}

}
```

Solution:

Barème :

- 25% déclaration de l'interface avec `extends IUnite`
- 10% `getVitesse()`
- 10% `move()`
- 10% `getDEstination()`
- 20% signature `stop` avec `default`
- 25% corps de `stop`

Question 4 1 point

Sur le même principe que la classe `Unite`, donnez le code de la classe `MovableUnite` qui hérite de `Unite`, qui implante `IMovableUnite` et qui stocke les attributs vitesse et destination. Une unité déplaçable est immobile initialement. La vitesse de l'unité sera renseignée en argument du constructeur. Les autres arguments du constructeurs sont les mêmes que ceux du constructeur de la classe `Unite`. Vous ne coderez pas la méthode `move` (qui sera étudiée plus loin) ni les getters `getVitesse` et `getDestination`.

Solution:

```
public class MovableUnite extends Unite implements IMovableUnite {


```

```
        private ICase destination;
        private int vitesse;

        public MovableUnite(ICarte carte, IJoueur proprietaire, ICase pos_init,
            int santemax, int vitesse) {
                super(carte, proprietaire, pos_init, santemax);
                this.vitesse=vitesse;
                stop();
        }

        @Override
        public ICase getDestination() {
                return destination;
        }

        public int getVitesse(){
                return vitesse;
        }

        @Override
        public void move(ICase destination) {
                /*voir plus loin*/
        }

}
```

Solution:

Barème :

- 20% `extends Unite`
- 20% `implements IMovableUnite`
- 20% déclaration attributs
- 10% signature constructeur
- 20% appel à `super`
- 10% initialisation correcte des attributs

Question 5 1 point

A l'aide du design pattern static Factory, donnez le code d'une classe `UniteFactory` qui propose les méthodes suivantes :

- `buildMur`, qui construit un mur. Un mur est une unité immobile.
- `buildCavalier`, qui construit un cavalier. Un cavalier est une unité mobile ayant une vitesse de 2.

Dans les deux cas, la carte, le propriétaire et la position initiale sont renseignés dans les arguments des méthodes.

Solution:

```
public final class UniteFactory{
        public static IUnite buildMur(ICarte carte, IJoueur proprietaire, ICase
            pos_init){
          return new Unite(carte, proprietaire, pos_init);
```

```
        }

        public static IUnite buildCavalier(ICarte carte, IJoueur proprietaire,
            ICase pos_init){
          return new MovableUnite(carte, proprietaire, pos_init,2);
        }
}
```

Solution:
Barème :

- 20% les deux méthodes sont statiques
- 20% les deux méthodes renvoient une `IUnite`
- 30% `buildMur`
- 30% `buildCavalier`

Exercice 3 – Donner des ordres aux unités au cours de la partie

Notions testées : DP command, DP singleton

Pour exécuter une partie au cours du temps, nous allons ajouter la capacité aux unités de pouvoir recevoir des ordres. Pour cela nous allons appliquer le design pattern Command. Nous disposons des interfaces `ICommandableUnite` et `ICommand` suivantes :

```
public interface ICommandableUnite {
      public ICommand getNextCommand(); //obtenir la commande
      public void setNextCommand(ICommand c);// changer la commande
}
```

```
public interface ICommand {

      public void execute(); //exécute la commande
}
```

Nous faisons étendre `IUnite` de `ICommandableUnite` afin que toutes les unités possèdent une commande courante. Notez que `IMovableUnite` étend alors automatiquement `ICommandableUnite`. Afin de respecter cette nouvelle interface, nous supposons que le code suivant est ajouté à la classe `Unite` :

```
private ICommand nextCommand;

public void setNextCommand(ICommand c) {  this.nextCommand = c; }
public ICommand getNextCommand() { return nextCommand; }
```

L'exécution d'une partie se fait par tour. À chaque tour, nous parcourons l'ensemble des unités de tous les joueurs et nous exécutons leur commande courante. Nous allons donc coder la classe `Partie` qui :

- possède comme attribut une collection de `IJoueur` renseignée en argument du constructeur ;

- une méthode `executeTour()` qui permet d'exécuter un tour de la partie.

Question 1 1 point

Donnez le code d'une classe `Partie`. On rappelle qu'un joueur a la capacité d'itérer sur ses unités (cf. exercice 1, question 2).

Solution:

```
public class Partie {

	private List<IJoueur> joueurs;

	public Partie(List<IJoueur> joueurs) {
		this.joueurs=joueurs;
	}


	public void executeTour() {


		for(IJoueur j : joueurs) {
			for(IUnite unite : j) {
				unite.getNextCommand().execute();
			}
		}

	}


}
```

Solution:

Barème :

20% déclaration de la liste de IJoueur en attribut
20% constructeur + initialisation de l'attribut
30% parcours des unités pour tout joueur
30% appel à `execute` sur chaque commande de chaque unité

On souhaite à présent coder deux classes de commande implantant `ICommand` :

- `Nop`, qui ne fait rien.
- `Move`, permettant de se déplacer sur la carte.

Question 2 1 point

La commande `Nop` étant une constante, implantez cette commande selon le pattern Singleton.

Solution:

```java
public class Nop implements ICommand {

	private static Nop instance= null;

	private Nop() {

	}

	public static Nop getInstance() {
		if(instance == null) {
			instance = new Nop();
		}
		return instance;
	}

	@Override
	public void execute() { }
}
```

Solution:

Barème :

- 20% `implements ICommand`
- 20% attribut statique de type Nop
- 20% constructeur déclaré private
- 20% `getInstance` renvoie toujours la même instance
- 20% méthode `execute` avec corps vide

Question 3 1½ points

Donnez le code de la classe `Move` :

- Elle a comme attribut une unité déplaçable, renseignée au moment de la construction. Notez que la case destination peut être déterminée en appelant `getDestination` sur cette unité ; il est donc inutile de passer cette information au constructeur et de la maintenir dans un attribut de `Move`.
- L'appel à la méthode `execute()` déplace l'unité d'une ou plusieurs cases, en fonction de sa vitesse, vers sa destination, et met éventuellement à jour la commande à exécuter pour le prochain tour. Plus précisément :
  - ◇ Si la position courante est égale à la destination de l'unité (on supposera que la classe `ICase` implante une méthode `equals()` adaptée) alors la position de l'unité reste inchangée et sa commande courante est changée en `Nop`.
  - ◇ Sinon obtenir le chemin jusque la destination. Pour cela on utilisera la méthode `List<ICase> bestpath(ICase from, ICase to)` de l'interface `ICarte` qui renvoie le plus court chemin de la position `from` pour atteindre la position `to`. Un chemin est renvoyé sous forme d'une liste de cases contigües, de la case origine à la case destination :
    - · Si au tour courant la case `to` est inaccessible depuis `from`, la méthode `bestpath` renvoie une liste vide. Dans ce cas, l'appel à `execute` ne change pas la position de l'unité ni sa commande courante. Ceci simule le fait que l'unité attend qu'un chemin se libère.
    - · S'il existe un chemin, alors `bestpath` retourne une liste non vide. La nouvelle position de l'unité est la $x-$ième case du chemin, où x est le minimum entre la vitesse de l'unité et la taille du chemin moins un. La commande reste la même pour le tour suivant.

Solution:

```
public class Move implements ICommand {

	private IMovableUnite u;
	public Move(IMovableUnite unite) {
		this.u = unite;
	}


	@Override
	public void execute() {
		if(u.getDestination().equals(u.getPosition())) {
			u.setNextCommand(Nop.getInstance());
			return;
		}
		List<ICase> chemin = u.getCarte().bestpath(u.getPosition(), u.
		    getDestination());
		if(chemin == null) {
			return;
		}

		ICase pos = chemin.get(Math.min(u.getVitesse(),chemin.size()-1));
		u.setPosition(pos);
	}

}
```

Solution:

Barème :

- 10% `implements ICommand`
- 10% un attribut de type `IMovableUnite`
- 10% constructeur
- 20% prochaine commande == Nop si destination == Position
- 20% appel correct à bestpath
- 10% sort de la fonction si le chemin calculé par bestpath est null
- 20% mise à jour de la position en fonction de la distance restante à parcourir et la vitesse

Exercice 4 – La carte

Notions testées : Type générique, Interface fonctionnelle, DP Observer/Observable

Nous proposons la classe `Case` suivante qui est une implémentation de `ICase` :

```
public class Case implements ICase {

	private int x;
```

```
	private int y;
	private final ICarte carte;
	private IUnite occupant=null;//initialisée lors du premier appel au getter
	private List<ICase> voisins = null;

	public Case(int x, int y, ICarte c) {
		this.x=x;
		this.y=y;
		this.carte=c;
	}

	@Override public int getX() { return x; }

	@Override public int getY() { return y; }


	@Override public void setOccupant(IUnite u) { occupant=u; }

	@Override public void removeOccupant() {  setOccupant(null);  }


	@Override public IUnite getOccupant() { return occupant; }


	@Override
	public Collection<ICase> getVoisins() {
		if(voisins == null) {
			voisins= new ArrayList<>();
			if(y < carte.getMaxY() ) voisins.add(carte.get(x,y+1));//Nord
			if(x < carte.getMaxX() ) voisins.add(carte.get(x+1,y));//Est
			if(y > 0 ) voisins.add(carte.get(x,y-1));//Sud
			if(x > 0 )  voisins.add(carte.get(x-1, y));//Ouest
		}
		return voisins;
	}
}
```

Question 1 ½ point

Pourquoi la liste des voisins est-elle construite au moment du premier appel au getter plutôt que dans le constructeur ?

Solution:

Pour pouvoir construire les différentes cases itérativement au moment de la construction de la carte. si on construisait les liens vers les voisins au moment de la construction de la case, il se peut que certaine des cases voisines ne soient pas encore instanciées ce qui ajouterai des références null dans la liste des voisins.

Barème : 0 ou 100% à l'appréciation du correcteur

Nous allons maintenant coder la classe abstraite `AbstractCarteMatrice`, une implémentation partielle de la carte `ICarte` qui n'implémentera pas la méthode `List<ICase> bestpath(ICase from, ICase to)`. L'implémentation sera basée sur un tableau Java à deux dimensions où l'élément (l,c) du tableau référencera la case d'abscisse c et d'ordonnée l du terrain de jeu. L'instanciation des cases sera faite au moment de la construction de la carte.

Question 2 1½ points

Donnez le code de la classe `AbstractCarteMatrice`.

Solution:

```
public abstract class AbstractCarteMatrice implements ICarte {

	private ICase[][] cases;

	public AbstractCarteMatrice(int nbLigne, int nbColonne) {

		cases= new ICase[nbLigne][nbColonne];
		for(int y=0;y<nbLigne;y++) {
			for(int x=0;x<nbColonne;x++) {
				cases[y][x]=new Case(x,y, this);
			}
		}
	}
	@Override public ICase get(int x, int y) { return cases[y][x]; }

	@Override public int getMaxX() { return cases[0].length-1;  }

	@Override public int getMaxY() {  return cases.length-1; }
```

Solution:

Barème :

- 10% `implements ICarte`
- 10% `abstract` sur la classe
- 10% attribut `ICase[][]`
- 30% constructeur (instanciation de la matrice + instanciation de chaque case)
- 20% `get` (y sur les lignes et x sur les les colonnes)
- 10% `getMaxX`
- 10% `getMaxY`

Nous allons à présent coder une extension concrète de `AbstractCarteMatrice` afin d'implémenter la méthode manquante. Notre solution sera basée sur un graphe. Nous proposons l'interface `Graph<T>` suivante :

```
public interface Graph<T> {

 Set<T> nodes();
 Set<T> voisins(T n);
 void addLink(T n1, T n2);
 void removeLink(T n1, T n2);
 boolean islink(T n1, T n2);
 void addNode(T n);
 void removeNode(T n);

 default public List<T> AStar(T start, T goal){
   //algorithme A* permettant de calculer le plus court chemin entre deux noeuds
```

```
  final Map<T,Double> gScore = new HashMap<>();                                    13
  final Map<T,Double> fScore = new HashMap<>();                                    14
  //initialisation des variables                                                   15
  ...                                                                              16
  final List<T> openSet = new ArrayList<>();                                       17
  openSet.add(start);                                                              18
  while(!openSet.isEmpty()) {                                                      19
    Collections.sort(openSet, new OpenSetComparator<T>(fScore));                   20
    //le noeud dans openset ayant la valeur de fScore la plus faible               21
    T current = openSet.remove(0);                                                 22
    ....                                                                           23
    for(T neighbor : voisins(current)) {                                           24
       ...                                                                         25
       double tentativeGScore = gScore.get(current) + current.distance(neighbor);  26
       ....                                                                        27
     }//fin for                                                                    28
  }//fin while                                                                     29
  return Collections.emptyList(); //cas où pas de chemin possible                  30
 }//fin méthode AStar                                                             31
}// fin interface                                                                  32
```

Afin que l'instruction à la ligne 22 corresponde bien à ce qui est demandé par le commentaire en ligne 21, il est nécessaire que la liste `openSet` soit triée par ordre de `fScore` croissant.

Question 3 1½ points

Donnez la classe `OpenSetComparator<T>` implantant l'interface `Comparator<T>` et permettant de comparer les éléments de `openSet` vis à vis de ce critère de tri : par ordre croissant de `fScore`.

Solution:

```
public class OpenSetComparator<T> implements Comparator<T>(){                      1
                                                                                   2
      private Map<T, Double> fscore;                                               3
      public OpenSetComparator(Map<T, Double> fscore){this.fscore= fscore;}        4
                                                                                   5
      public int compareTo(T e1, T e2){                                            6
            return  fScore.get(c1).compareTo(fScore.get(c2))                       7
      }                                                                            8
}                                                                                  9
```

Solution:

Barème :

- 10% `implements Comparator<T>`
- 10% attribut `fscore`
- 20% constructeur
- 20% `get` (y sur les lignes et x sur les les colonnes)
- 10% signature de compareTo (argument forcément de type générique)
- 30% code de compareTo

Question 4 ½ point

Plutôt que de passer en argument à `Collections.sort` une instance de `OpenSetComparator` à la

ligne 20, il est possible de lui passer en argument un lambda. Réecrivez la ligne 20 de manière équivalente avec un lambda.

Solution:

```
(c1,c2)-> fScore.get(c1).compareTo(fScore.get(c2))
```

Solution:

Barème :

50% syntaxe des arguments avec flèche

50% même contenu que le corps de la fonction compareTo de la question précédente

Question 5 ½ point

Le compilateur indique une erreur à la ligne 26 car la méthode `distance` n'est pas définie sur le type `T`. Que faire pour corriger l'erreur ?

Solution:

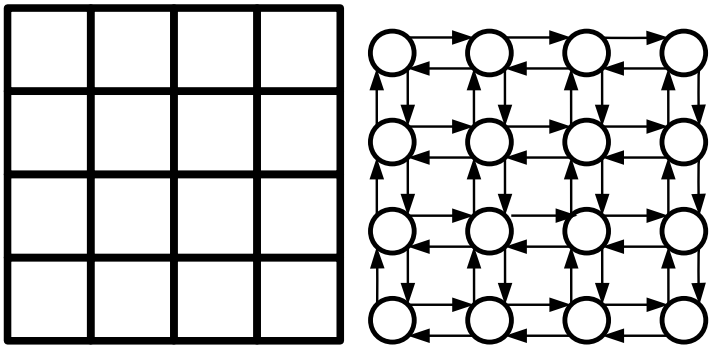
Pour corriger ceci, il faut modifier la déclaration de `Graph<T>` par `Graph<T extends Localisable<T>>`

où `Localisable<T>` serait une interface contenant une méthode `public double distance(T other);`

Barème : 0 ou 100 % à l'appréciation du correcteur

On supposera par la suite qu'il existe une classe `GraphImpl<T>` qui implante l'interface `Graph<T>`.

Pour calculer le plus court chemin dans une carte du jeu, nous pouvons représenter la carte comme un graphe où les nœuds du graphe sont les cases de la carte et deux nœuds sont reliés si leurs cases correspondantes sont voisines sur la carte. La figure 2 illustre cette représentation avec une carte vide de 4x4 cases. Comme chaque case ne peut comporter au plus qu'une unité, toute case qui est occupée par une unité devient inaccessible. Ceci a donc pour conséquence sur le graphe de supprimer les arcs entrants sur une case occupée par une unité. La figure 3 illustre la même carte que la figure 2 mais avec 5 unités.

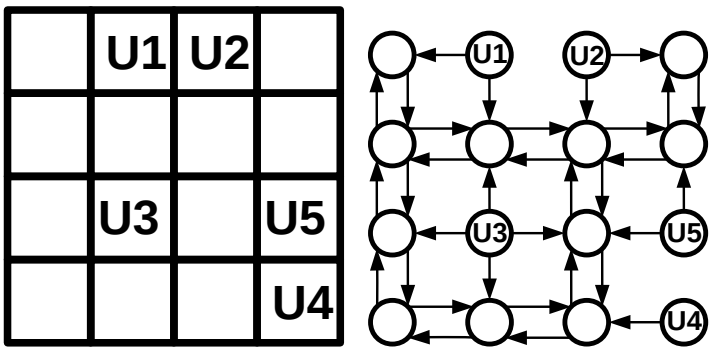


(a) Représentation carte (b) Représentation graphe

FIGURE 2 – Représentation d'une carte vide (sans unité)

Question 6 1½ points

Donnez le code de la classe concrète `CarteMatriceWithGraph` qui étend `AbstractCarteMatrice` et



(a) Représentation carte (b) Représentation graphe

FIGURE 3 – Représentation d'une carte avec 5 unités

délègue l'appel de la méthode `bestpath` à la méthode `AStar` du graphe. Dans le constructeur, une fois que la matrice a été instanciée, on ajoutera dans le graphe toutes les cases de la carte et les liens devront correspondre à une carte où toutes les cases sont vides.

Solution:

```
public class CarteMatriceWithGraph extends AbstractCarteMatrice {

	private Graph<ICase> graph;


	public CarteMatriceWithGraph(int nbLigne, int nbColonne) {
		super(nbLigne, nbColonne);
		graph=new GraphImpl<>();
		for(ICase c : this) {//j'avais ajouter un iterator dans la super
		    classe
			graph.addNode(c);
		}
		for(ICase c : this) {
			open(c);// ne peut pas etre fait en meme temps que addnode
			    car les liens doivent etre ajoute une fois que tous les
			    noeuds ont été declares dans le graphe.
		}
	}
	@Override
	public List<ICase> bestpath(ICase from, ICase to) {
		return graph.AStar(from, to);
	}

	private void open(ICase c) {
		for(ICase voisin : c.getVoisins()) {
			graph.addLink(voisin,c);
		}
	}
	private void close(ICase c) {
		for(ICase voisin : c.getVoisins()) {
			graph.removeLink(voisin,c);
		}
```

```
        }
}
```

On notera également que `ICase` doit étendre `Localisable<ICase>` pour respecter la contrainte de type de `Graph<T extends Localisable<T>>` il est alors possible de définir un code par défaut de la méthode `distance` en faisant appel aux getters de x et y offerts par l'interface.

Solution:

Barème :

- 10% `extends AbstractCarteMatrice`
- 10% attribut de type `Graph<ICase>`
- 10% appel au super constructeur
- 10% instanciation du graphe
- 20% ajout des noeuds dans le graphe
- 30% ajout des liens
- 10% méthode `bestpath`

Question 7 ½ point

Quel design pattern avons nous mis en application à la question précédente ?

Solution:

DP adapter

Barème : 0 % ou 100%

Nous souhaitons à présent que l'état de la carte se mette à jour en fonction du mouvement des unités, de leur création ou de leur suppression. Pour cela nous allons utiliser le pattern Observer/Observable. Ainsi chaque case est observée par la carte. Dès que la valeur de l'occupant de la case change, il faut que la carte soit notifiée pour qu'elle mette à jour l'état du graphe. Si la case est devenu vide, alors il faut ajouter les liens entrants sur la case, sinon il faut les supprimer. Vous disposez des deux interfaces suivantes :

- l'interface `IObserver` :

```
public interface IObserver {
  void update(IObservable source, Object arg);
}
```

- l'interface `IObservable` :

```
public interface IObservable {
       void addObserver(IObserver o);//ajoute un observer
       void notifyObservers(Object arg);//appel l'update de chaque observer avec l'
           argument arg
}
```

Lors de l'appel à `notifyObservers`, l'objet arg passé en paramètre est transmis à tous les observers via le deuxième argument de la méthode `update`. Dans notre cas, cet objet sera le nouvel occupant de la case (ou `null` si la case se vide).

Question 8 1½ points

Donnez le code de la classe abstraite `AbstractObservable` qui implante toutes les méthodes de l'interface `IObservable`.

Solution:

```
public abstract class AbstractObservable implements IObservable {

	private Collection<IObserver> observers = new ArrayList<>();

	@Override
	public void addObserver(IObserver o) {
		observers.add(o);
	}

	@Override
	public void notifyObservers(Object arg) {
		for(IObserver o : observers)
			o.update(this, arg);
	}
}
```

Solution:

Barème :

- 10% `implements IObservable`
- 10% classe abstraite
- 10% collection de IObserver déclaré en attribut
- 10% instanciation de l'attribut
- 20% `addObserver`
- 40% `notifyObservers`

Question 9 1 point

Quelles sont les modifications nécessaires à l'interface `ICase` et la classe `Case` ?

Solution:

- l'interface `ICase` doit étendre `IObservable`
- La classe `Case` :
  - ◇ doit étendre `AbstractObservable`
  - ◇ doit faire appel à `notifyObservers(u);` dans la méthode `setOccupant(IUnite u)`

Barème :

- 30% `ICase` doit étendre `IObservable`
- 30% `Case` doit étendre `AbstractObservable`
- 40% appel à `notifyObservers(u);` dans la méthode `setOccupant(IUnite u)`

Question 10 2 points

Sachant que l'abonnement aux cases se fera au moment de la construction de la carte donnez les modifications à apporter sur la classe `CarteMatriceWithGraph`.

Solution:

La classe `CarteMatriceWithGraph` :

- doit `implements IObserver`
- doit se déclarer comme observer de chaque case, on ajoute dans le constructeur

```
for(ICase c : this) {
       c.addObserver(this);
}
```

- doit implanter la méthode `update` dont le code est

```
public void update(IObservable o, Object arg) {
 if(! (o instanceof ICase)) throw new IllegalArgumentException(o+" is
    not an ICase");
 ICase c = (ICase) o;
 if(arg == null)
   open(c);
 else
  close(c);
}
```

Solution:

Barème :

20% `CarteMatriceWithGraph implements IObserver`
30% ajout du addObserver(this) pour chaque Case
50% code du update