3I002 - Programmation Orientée Objet en Java : Examen Mai 2017

Université Pierre et Marie Curie - 2 heures

Barème *indicatif* sur 20 points.

L'épreuve se déroule sur machine, dans l'environnement Eclipse a priori, mais sans que ce soit imposé. Nous demandons d'écrire un certain nombre de fichiers Java. Le langage utilisé est Java 8. Les fichiers fournis sont évalués automatiquement et notés grâce à une série de tests JUnit 4 exécutés automatiquement. Le serveur prend en entrée vos sources (la copie de l'étudiant), et produit un compte-rendu détaillé de l'exécution (et éventuellement de l'échec) de la compilation et des tests. Elle calcule également une note en suivant le barème précisé dans chaque question (points pour la compilation et pour chaque test validant une partie de votre réponse). La correction étant automatique, il est impératif de respecter les noms de package, les noms de classes, et les interfaces imposées par l'énoncé, sous peine de voir vos fichiers-réponses rejetés par le compilateur Java avant même le lancement des tests ! La correction automatique utilise la plateforme CodeGradX de Christian Queinnec.

Important : Nommez votre projet : examMai2017 de façon à ce que vos sources se trouvent dans `~/workspace/examMai2017/src/`

A la fin de l'épreuve déconnectez-vous normalement, un script viendra collecter vos sources directement sur le compte temporaire qui vous a été attribué pour l'épreuve.

Le sujet est composé de quatre exercices indépendants.

Tous les sources reproduits dans cet énoncé sont disponibles dans une archive, à télécharger ici : https://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2016/ue/3I002-2017fev/examen/src.tgz.

Décompressez ce fichier dans votre projet,

```
cd ~/workspace/examMai2017/ ; tar xvzf src.tgz
```

Pensez à faire "File->refresh" sous Eclipse, et à ajouter JUnit (Build Path -> configure build path -> Libraries -> Add library -> JUnit 4). Certains fichiers ne compilent pas initialement, c'est normal il vous faudra les corriger au fil de l'examen.

Exercice 1 : Discussions en ligne (7,5 points)

Important : toutes les classes de cet exercice seront placées dans le package `forum`

Notions testées : Observer, Decorator.

On considère un système simulant des discussions en ligne (ou *chat*) entre participants (les *Chatters*).

Le système est constitué de deux types d'entités, les forums matérialisés par l'interface `IForum` représentent des discussions actives. Chaque forum a un nom (le sujet de la discussion), et un certain nombre de participants (des `IChatter`).

Les *chatters* peuvent s'abonner (avec `rejoindre`) ou se désabonner (avec `quitter`) d'un forum.

Le fait de `poster` un message sur un forum doit le rendre visible de tous les participants. Pour cela, chaque `IChatter` dispose d'une méthode de notification `messageRecu()`. Si l'on `posterMessage()` sur un forum, celui-ci sera transmis à tout `IChatter` ayant rejoint le forum.

IForum.java

```
package forum;

public interface IForum {
      /**
```

```
     * Le nom de ce forum, typiquement le sujet de discussion.
     * @return le nom du forum
     */
    String getName();
    /**
     * Permet de poster ou écrire un message sur le forum.
     * Tous les participants actifs en seront notifiés.
     * @param auteur Le nom de l'auteur du message, peut être arbitraire (pas né
         cessairement le nom d'un IChatter participant).
     * @param message Le contenu du message
     * @return vrai si le message a bien été posté, faux sinon.
     */
    boolean posterMessage (String auteur, String message);

    /**
     * Permet à un IChatter de rejoindre le forum.
     * A partir du moment où il a rejoint le forum, il sera notifié @linkto{IChatter.
         messageRecu} de tout message posté dessus.
     * On en peut pas rejoindre deux fois le même forum (return false).
     * @param chatter la personne à abonner au forum
     * @return vrai si le chatter à effectivement réussi à rejoindre le forum
     */
    boolean rejoindre (IChatter chatter);
    /**
     * Le nombre de participants actifs, i.e. ayant actuellement rejoint avec succès ce
         forum.
     * @return un entier >= 0
     */
    int nbParticipants();
    /**
     * Permet à un IChatter ayant précédemment rejoint le forum de s'en désabonner.
     * Il ne sera plus notifié des posts sur ce forum.
     * @param chatter le chatter qui souhaite quitter le forum
     * @return vrai si le chatter était effectivement abonné, faux sinon
     */
    boolean quitter (IChatter chatter);
}
```

IChatter.java

```
package forum;

public interface IChatter {
    /**
     * Le nom de ce chatter, utilisé comme nom d'auteur pour les message qu'il envoie.
     * @return une chaine non vide, désignant de façon unique ce chatter
     */
    String getName();
    /**
     * Signale que quelqu'un a posté un message sur l'un des IForum auquel le chatter
         participe actuellement.
     * Comme le chatter peut être abonné à plusieurs forums, on lui précise le nom du
         forum ou ce message a été posté.
     * @param forum nom du forum où a été posté ce message
     * @param auteur nom de l'auteur du message
     * @param message contenu du message
     */
    void messageRecu (String forum, String auteur, String message);

}
```

IChatter, IForum

a) (0,75 points) Réalisez l'interface `IChatter` dans une classe `forum.SimpleChatter`. Vous ajouterez un constructeur `public SimpleChatter(String name)`.

La notification se contentera d'afficher un message sur la sortie standard incluant : le nom du récepteur du message, le nom de l'auteur du message, le nom du forum où a été posté le message, et le contenu du message.

Le test fourni ici spécifie le résultat attendu.

TestSimpleChatter.java

```java
package forum;
public class TestSimpleChatter {
	public static void main(String[] args) {
		SimpleChatter a = new SimpleChatter("alice");
		a.messageRecu("forum", "bob", "message");
		// affiche sur une ligne :
		//(chez alice) bob@forum> message
	}
}
```

> Barème :
> implements IChatter : 30%
> getName + ctor : 20 %
> Test format de la string : 50%

Fournir `src/forum/SimpleChatter.java`.

b) (1,75 points) Réalisez l'interface `IForum` dans une classe `forum.Forum`. Vous ajouterez un constructeur `public Forum(String name)`.

Ce test spécifie le résultat attendu. Attention, on ne veut pas de double abonnement.

TestForum.java

```java
package forum;
public class TestForum {
	public static void main(String[] args) {
		IChatter alice = new SimpleChatter("alice");
		IChatter bob = new SimpleChatter("bob");
		IChatter charlie = new SimpleChatter("charlie");

		IForum forum = new Forum("Java");

		forum.rejoindre(alice);
		forum.posterMessage("alice", "Il n'y a personne ?");
		forum.rejoindre(charlie);

		forum.posterMessage("alice", "Vive les examens ! Surtout en POBJ !");
		forum.posterMessage("charlie", "Tu parles, Python c'est tellement mieux !");
		forum.rejoindre(bob);
		forum.posterMessage("bob", "Ouais c'est trop cooool java.");
		forum.posterMessage("charlie", "Si au moins le prof connaissait Ruby...");

		forum.quitter(alice);
		forum.quitter(bob);
	}
}
/** Trace attendue :
```

```
(chez alice) alice@Java> Il n'y a personne ?
(chez alice) alice@Java> Vive les examens ! Surtout en POBJ !
(chez charlie) alice@Java> Vive les examens ! Surtout en POBJ !
(chez alice) charlie@Java> Tu parles, Python c'est tellement mieux !
(chez charlie) charlie@Java> Tu parles, Python c'est tellement mieux !
(chez alice) bob@Java> Ouais c'est trop cooool java.
(chez charlie) bob@Java> Ouais c'est trop cooool java.
(chez bob) bob@Java> Ouais c'est trop cooool java.
(chez alice) charlie@Java> Si au moins le prof connaissait Ruby...
(chez charlie) charlie@Java> Si au moins le prof connaissait Ruby...
(chez bob) charlie@Java> Si au moins le prof connaissait Ruby...
**/
```

Fournir `src/forum/Forum.java`.

Forum.java

```java
package forum;

import java.util.HashSet;
import java.util.Set;

/** "Sujet" ou "Observable" du DP Observer.
 */
public class Forum implements IForum {

	// NB : utilisation d'un Set pour les observateurs
	private Set<IChatter> participants = new HashSet<>();
	private String name;

	public Forum(String name) {
		this.name = name;
	}

	@Override
	public String getName() {
		return name;
	}

	@Override
	public int nbParticipants() {
		return participants.size();
	}

	@Override
	public boolean rejoindre(IChatter chatter) {
		// API Set: return false s'il existe déjà
		return participants.add(chatter);
	}

	@Override
	public boolean quitter(IChatter chatter) {
		// API Set: return false s'il n'existe pas
		return participants.remove(chatter);
	}

	@Override
	public boolean posterMessage(String auteur, String message) {
		// textbook méthode "notifier" du DP
```

```
            for (IChatter c : participants) {
                  c.messageRecu(getName(), auteur, message);
            }
            return true;
      }

}
```

Barème :
implements IForum : 10 %
name + getName : 10%
nbParticipants : 20 % augmente avec rejoindre ok, diminue avec quitter
rejoindre : 20 % augmente nbParticipants, sans effet si déjà abonné, impact sur poster.
quitter : 20 % réduit nbParticipants, sans effet si pas abonné, impact sur poster.
poster : 20 % tout le monde est notifié, contenu des string ok

Décorations de IForum

Cette conversation a quelques soucis auxquels on va remédier en améliorant les forums par des décorations.
Le résultat global attendu à la fin de **c), d), e)** est le suivant.

TestDecoration.java

```java
package forum;

import java.util.Arrays;
import java.util.Collection;

public class TestDecoration {
      public static void main(String[] args) {
            IChatter alice = new SimpleChatter("alice");
            IChatter bob = new SimpleChatter("bob");
            IChatter charlie = new SimpleChatter("charlie");

            IForum forum = new Forum("Java");

            forum = new ForumAnnonce(forum);

            Collection<String> bad = Arrays.asList("Python","Ruby","Scala");
            forum = new ForumProtege(forum, bad);

            forum.rejoindre(alice);
            forum.posterMessage("alice", "Il n'y a personne ?");
            forum.rejoindre(charlie);

            forum.posterMessage("alice", "Vive les examens ! Surtout en POBJ !");
            forum.posterMessage("charlie", "Tu parles, Python c'est tellement mieux !");
            forum.rejoindre(bob);
            forum.posterMessage("bob", "Ouais c'est trop cooool java.");
            forum.posterMessage("charlie", "Si au moins le prof connaissait Ruby...");

            forum.quitter(bob);
            forum.quitter(alice);
            forum.posterMessage("charlie", "Java 8 c'est le Scala du pauvre");
      }
```

```
}
/**
(chez alice) Java@Java> alice rejoint la discussion.
(chez alice) alice@Java> Il n'y a personne ?
(chez alice) Java@Java> charlie rejoint la discussion.
(chez charlie) Java@Java> charlie rejoint la discussion.
(chez alice) alice@Java> Vive les examens ! Surtout en POBJ !
(chez charlie) alice@Java> Vive les examens ! Surtout en POBJ !
(chez alice) Java@Java> bob rejoint la discussion.
(chez charlie) Java@Java> bob rejoint la discussion.
(chez bob) Java@Java> bob rejoint la discussion.
(chez alice) bob@Java> Ouais c'est trop cooool java.
(chez charlie) bob@Java> Ouais c'est trop cooool java.
(chez bob) bob@Java> Ouais c'est trop cooool java.
(chez alice) Java@Java> bob quitte la discussion.
(chez charlie) Java@Java> bob quitte la discussion.
(chez charlie) Java@Java> alice quitte la discussion.
**/
```

c) (1 point)

Pour commencer, dans la classe **abstraite** `ForumDecorator` réalisez un décorateur abstrait de l'interface `IForum` : toutes les méthodes de `IForum` sont implémentées par délégation sur un attribut **privé** typé `IForum`, passé au constructeur `public ForumDecorator(IForum deco)`. Le comportement d'un forum décoré est donc identique à celui d'un forum normal.

Les points indiqués en **gras** (classe abstraite, attribut privé) sont des contraintes qui seront testées.

TestForumDecorator.java

```java
package forum;

public class TestForumDecorator {
	public static void main(String[] args) {
		Forum f = new Forum("Java");
		IForum decore = new ForumDecoreConcret(f);
		// on peut utiliser f ou decore de façon identique, cf tests précédents.
	}
}

/** Une classe concrète, joue le rôle de décorateur concret.
 */
class ForumDecoreConcret extends ForumDecorator {
	/**
	 * ForumDecorator a donc un constructeur prenant un IForum en paramètre.
	 * @param decore le forum à décorer.
	 */
	public ForumDecoreConcret(IForum decore) {
		super(decore);
	}
}
```

ForumDecorator.java

```java
package forum;

// DP Decorator : decorator abstrait
// à générer sous eclipse en 3 étapes :
// créer la classe,
```

```java
// ajouter l'attribut decore typé IForum,
// Source->generate Delegate methods, tout cocher sur decore -> finish
public abstract class ForumDecorator implements IForum {
	// private par contrainte d'énoncé
	// contourné dans pas mal de copies avec un getDeco() inélégant et inutile, mais non sanctionné.
	// préférez l'utilisation de super.méthode dans les decorateurs concrets à un accès direct à decore.
	private IForum decore;

	public ForumDecorator(IForum decore) {
		this.decore = decore;
	}

	@Override
	public String getName() {
		return decore.getName();
	}

	@Override
	public int nbParticipants() {
		return decore.nbParticipants();
	}

	@Override
	public boolean rejoindre(IChatter chatter) {
		return decore.rejoindre(chatter);
	}

	@Override
	public boolean quitter(IChatter chatter) {
		return decore.quitter(chatter);
	}

	@Override
	public boolean posterMessage(String auteur, String message) {
		return decore.posterMessage(auteur, message);
	}
}
```

10% implements IForum

10% class abstraite

20% ctor

10% attribut déclaré private

50% = 10% par méthode déléguée correctement

Fournir `src/forum/ForumDecorator.java`.

d) (1 point)

Implantez la classe `ForumAnnonce` qui étend `ForumDecorator` et ajoute des messages artificiels pour notifier les partipants des départs et arrivées des autres participants.

A chaque fois qu'un participant rejoint ou quitte **avec succès** le forum, un message avec pour auteur le nom du forum lui-même est généré, de la forme "bob quitte la discussion." ou "alice rejoint la discussion." Le chatter qui vient de rejoindre la discussion reçoit également la notification.

Fournir `src/forum/ForumAnnonce.java`.

ForumAnnonce.java

```java
package forum;

// un Decorateur concret
public class ForumAnnonce extends ForumDecorator {

	public ForumAnnonce(IForum decore) {
		super(decore);
	}

	@Override
	public boolean rejoindre(IChatter chatter) {
		// super.methode dans les deco concrets
		if (super.rejoindre(chatter)) {
			// poste seulement sur succès
			posterMessage(getName(), chatter.getName()+" rejoint la discussion.");
			return true;
		}
		return false;
	}

	@Override
	public boolean quitter(IChatter chatter) {
		// super.methode dans les deco concrets
		if (super.quitter(chatter)) {
			// poste seulement sur succès
			posterMessage(getName(), chatter.getName()+" quitte la discussion.");
			return true;
		}
		return false;
	}


}
```

10% extends ForumDecorator

10% aucun attribut (pas de copie du decore)

20% les 3 méthodes qu'il ne fallait pas redéfinir ne le sont pas.

20% rejoindre continue de rejoindre, quitter de quitter (invocation de super.)

20% le message additionnel est bien posté sur rejoindre/quitter

20% si rejoindre ou quitter échoue, pas de notifications

e) (1,25 point)

Implantez la classe `ForumProtege` qui étend `ForumDecorator` et permet d'éviter qu'il n'y ait des gros mots dans la conversation. On lui passe un ensemble de mots qu'il **copie** dans un attribut à la construction : `public ForumProtege(IForum decore, Collection<String> interdits)`.

A chaque fois qu'on poste un message, il teste si l'un des mots interdits est contenu dans le message : on utilisera simplement la méthode `boolean contains(String)` de la classe `String`.

Si le message contient l'un des mots interdits, le message est refusé, il ne sera pas diffusé sur le forum (et la méthode `posterMessage()` rend faux).

Fournir `src/forum/ForumProtege.java`.

ForumProtege.java

```java
package forum;

import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

public class ForumProtege extends ForumDecorator {

	private Set<String> bad ;

	public ForumProtege(IForum decore, Collection<String> bad) {
		super(decore);
		// NB : copie dans l'attribut, demandé par énoncé
		this.bad = new HashSet<>(bad);
	}

	@Override
	public boolean posterMessage(String auteur, String message) {
		for (String w : bad) {
			if (message.contains(w)) {
				return false;
			}
		}
		// super.methode dans les deco concrets
		return super.posterMessage(auteur, message);
	}

}
```

10% extends ForumDecorator
10% aucun attribut typé IForum (pas de copie du decore)
20% l'ensemble passé a bien été copié
20% les 4 méthodes qu'il ne fallait pas redéfinir ne le sont pas.
20% valeur de retour de poster true/false
20% le filtrage a bien lieu

Perroquet

On considère un Chatter un peu particulier, qui écoute un ou plusieurs forums, et répète tout ce qu'il entend dans un forum cible, en s'attribuant les messages qu'il répète.

Son constructeur `public Perroquet(String name, IForum cible)` prend donc un `IForum` à cibler en plus de son nom.

f) (1 point) Implémentez cette classe `Perroquet`.

On devra pouvoir réaliser le test suivant:

TestPerroquet.java

```java
package forum;

public class TestPerroquet {
	public static void main(String[] args) {
		IChatter alice = new SimpleChatter("alice");
		IForum f1 = new Forum("Java");
		f1.rejoindre(alice);
```

```
            IForum f2 = new Forum("Prog");

            IChatter charlie = new Perroquet("charlie", f2);
            f1.rejoindre(charlie);

            IChatter bob = new SimpleChatter("bob");
            f2.rejoindre(bob);

            // question 1.g)
            // f2.rejoindre(charlie);

            f1.posterMessage("alice", "J'ai la solution en O(1) !");
            f2.posterMessage("bob", "Menteur charlie, c'est Alice qui a trouvé !");
      }
}
/** Trace :
(chez alice) alice@Java> J'ai la solution en O(1) !
(chez bob) charlie@Prog> J'ai la solution en O(1) !
(chez bob) bob@Prog> Menteur charlie, c'est Alice qui a trouvé !
**/
```

Perroquet.java

```
package forum;

// Un sujet qui transmet les notifications, très utile pour mettre en place des chaines
    de notifications.
public class Perroquet implements IChatter {

      private String name;
      private IForum cible;
      public Perroquet(String name, IForum cible) {
            this.name = name;
            this.cible = cible;
      }
      @Override
      public String getName() {
            return name;
      }
      @Override
      public void messageRecu(String forum, String auteur, String message) {
            // patch du StackOverflow si on est abonné au forum cible
            if (! forum.equals(cible.getName())) {
                  cible.posterMessage(name, message);
            }
      }
}
```

10% extends ForumDecorator
10% aucun attribut typé IForum (pas de copie du decore)
20% l'ensemble passé a bien été copié
20% les 4 méthodes qu'il ne fallait pas redéfinir ne le sont pas.
20% valeur de retour de poster true/false
20% le filtrage a bien lieu

Fournir `src/forum/Perroquet.java`.

g) (0,75 point)

Que se passe-t-il si dans le scénario précédent on abonne “charlie” au forum cible `f2` ? Modifiez le code du Perroquet pour éviter ce problème, chaque message plagié par le perroquet ne doit être affiché qu’une seule fois dans le forum cible, et ce quels que soient ses abonnements actuels. La modification doit avoir pour effet que le Perroquet refuse de plagier les messages postés sur son forum cible (quel qu’en soit l’auteur).

> 50 % le scénario décrit ne provoque pas de StackOverflow
>
> 50 % le patch correspond bien à ce qui est demandé (filtrage par nom de forum)

Exercice 2 : Combinaison d’itérateurs (3 points)

Important : toutes les classes de cet exercice seront placées dans le package `dico`

On considère la classe `Dictionnaire` qui pour diverses raisons stocke deux listes distinctes de mots correspondant aux mots communs (table, chaise...) et aux noms propres (Turing, Charlemagne...).

Dictionnaire.java

```
package dico;

import java.util.ArrayList;
import java.util.List;

public class Dictionnaire {
      protected List<String> motsCommuns ;
      protected List<String> nomsPropres ;

      public Dictionnaire(List<String> motsCommuns, List<String> nomsPropres) {
            this.motsCommuns = new ArrayList<String>(motsCommuns);
            this.nomsPropres = new ArrayList<String>(nomsPropres);
      }
      public List<String> getMotsCommuns() {
            return motsCommuns;
      }
      public List<String> getNomsPropres() {
            return nomsPropres;
      }
}
```

On souhaite pouvoir éxécuter ce test, qui itère sur toutes les entrées du dictionnaire, sans faire de copies inutiles.

TestDictionnaire.java

```
package dico;

import java.util.Arrays;

public class TestDictionnaire {
      public static void main(String[] args) {
            DictionnaireComplet dico = new DictionnaireComplet(Arrays.asList("chaise","table
                "), Arrays.asList("Charlemagne","Turing"));
            for (String w: dico) {
                  System.out.println(w);
            }
```

```
    }
}
/** trace attendue:
chaise
table
Charlemagne
Turing
**/
```

a) (2 points) Dans une nouvelle classe **dico.Iterators**, écrivez une méthode `public static <T> Iterator<T> concat (Iterator<? extends T> it1, Iterator<? extends T> it2)`.

Cette méthode générique doit produire un `Iterator` qui concatène logiquement le contenu accessible par chacun des itérateurs qui lui sont passés. Cet itérateur rend donc tous les éléments accessibles par $it1$ puis (quand il n'y en a plus dans $it1$) tous les éléments accessibles par $it2$.

On demande de réaliser une nouvelle classe qui implémente `Iterator<T>` et d'en rendre une instance dans la méthode static `concat`. La classe portera en attribut les deux itérateurs sur lesquels elle s'appuie pour répondre à `next` et `hasNext`. Les tests ne concerneront que ces deux méthodes.

Fournir `src/dico/Iterators.java`.

Corrigé :

Iterators.java

```java
package dico;

import java.util.Iterator;

/** NB: classe abstraite non demandée, mais logique sur une classe utilitaire ne portant que des méthodes static.
 * Empeche de faire des "new Iterators", totalement inutiles.
 */
public abstract class Iterators {

	public static <T> Iterator<T> concat (Iterator<? extends T> it1, Iterator<? extends T> it2) {
		// on peut aussi faire une classe séparée
		// force à passer les iterateurs it1/it2 et à les copier en attribut
		// e.g. return new ConcatIterator<T>(it1,it2);
		// ils sont visibles ici dans le contexte de la classe anonyme définie dans la méthode elle-même.
		return new Iterator<T> (){
			@Override
			public boolean hasNext() {
				return it1.hasNext() || it2.hasNext();
			}

			@Override
			public T next() {
				if (it1.hasNext()) {
					return it1.next();
				} else {
					// NB : on laisse déborder le cas échéant
					// c'est la faute de l'utilisateur
					return it2.next();
				}
			}
		};
	}
```

```
}

// version classe séparée
//class ConcatIterator<T> implements Iterator<T> {
// private Iterator<? extends T> it1;
// private Iterator<? extends T> it2;
//
// public ConcatIterator(Iterator<? extends T> it1, Iterator<? extends T> it2) {
//  this.it1 = it1;
//  this.it2 = it2;
// }
// ... le reste comme la version anonyme
```

Barème :

La classe et sa méthode static existent 10%

L'itérateur rendu est bien défini dans ce package (pas dans java.util) 10%

tests hasNext cas aux limites 20 %

test itération correct 50 % (-20% si c'est le mauvais ordre)

test avec des sous types Animal/Chien/Elephant. 10%

b) (1 point)

Écrivez la classe `dico.DictionnaireComplet` qui `extends Dictionnaire`, de façon à pouvoir passer le test décrit plus haut, et en utilisant la méthode définie en **a)**.

Fournir `src/dico/DictionnaireComplet.java`.

Corrigé :

DictionnaireComplet.java

```java
package dico;

import java.util.Iterator;
import java.util.List;

public class DictionnaireComplet extends Dictionnaire implements Iterable<String> {

	public DictionnaireComplet(List<String> motsCommuns, List<String> motsPropres) {
		super(motsCommuns, motsPropres);
	}

	@Override
	public Iterator<String> iterator() {
		return Iterators.concat(motsCommuns.iterator(), nomsPropres.iterator());
	}

}
```

Barème :

La classe extends Dico 10 %

Elle implémente Iterable<String> 20%

Constructeur 10 %

30% l'itérateur est le bon (contenu ok fonctionnellement)

30% on teste que c'est le même type que celui rendu par Iterators.concat) et que l'iterator rendu est bien défini dans ce package (pas dans java.util)

Exercice 3 : Arbre Binaire de Recherche (4 points)

Important : toutes les classes de cet exercice seront placées dans le package `abr`

Notions testées : Structures de données récursives (arbres).

Un arbre binaire de recherche (ABR) est une structure de données pour représenter un ensemble totalement ordonné. Un arbre binaire de recherche permet des opérations rapides pour rechercher une clé, insérer ou supprimer une clé.

On considère un arbre binaire de recherche (ABR), où chaque noeud porte un champ *valeur* (typé String dans l'exercice) et a deux fils, qui sont eux-mêmes des noeuds.

Un ABR respecte un invariant fondamental : toutes les valeurs portées par le sous-arbre dont le fils gauche (respectivement fils droit) est la racine sont strictement *inférieures* (respectivement *supérieures*) à la valeur portée par le noeud. Ajouter une nouvelle valeur consiste à ajouter un noeud dans l'arbre (à la bonne position pour respecter l'invariant).

Si la taille des sous arbres gauche et droit est *équilibrée* (à peu près la même taille) on peut ajouter ou trouver si une valeur appartient à l'arbre en $O(log(n))$. Dans cet exercice, on ne se préoccupera pas d'équilibrer l'arbre, bien qu'il existe des algorithmes pour le faire efficacement au fil de l'insertion.

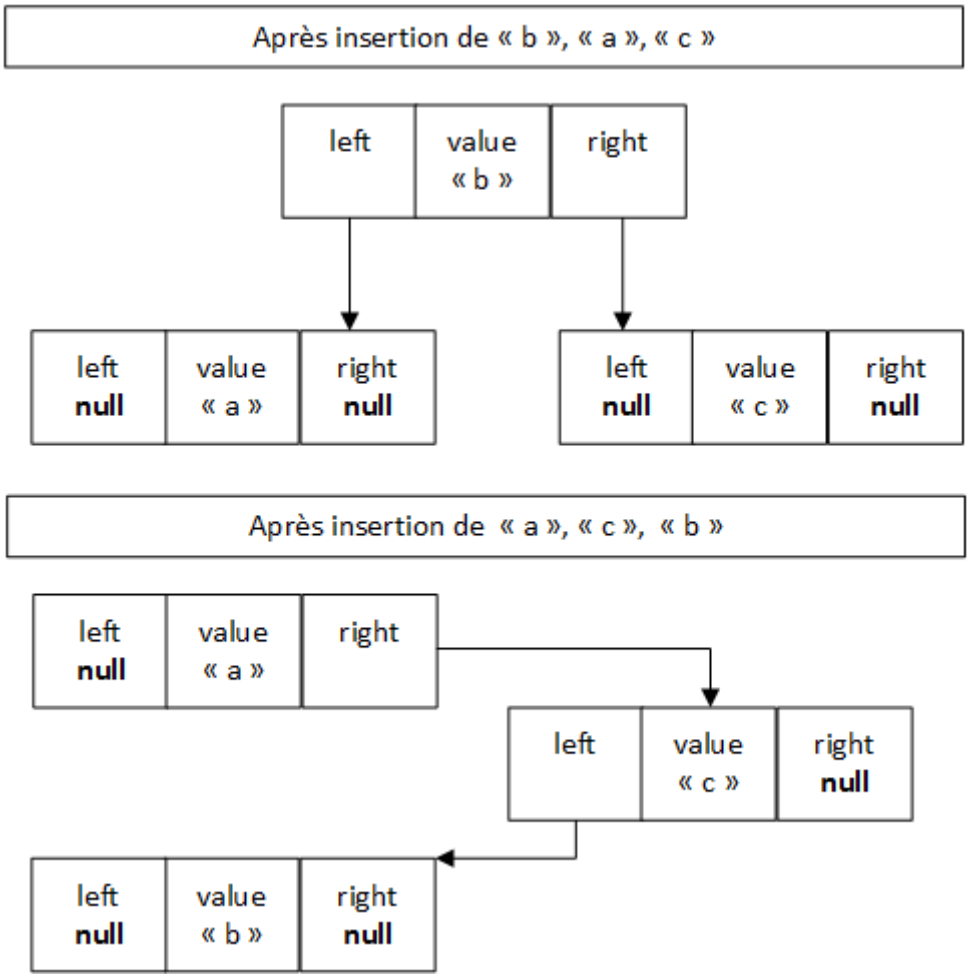

Figure 1: Arbres binaires obtenus selon l'ordre d'insertion des éléments.

a) (2,5 points)

L'interface `INode` qu'implantera la classe `Node` demandée est donnée ici (et fournie dans l'archive):

INode.java

```java
package abr;

public interface INode {
	/**
	 * La valeur (clé) associée à ce node.
	 * @return la string associée au noeud courant
	 */
	String getValue();
	/**
	 * Le fils gauche de ce noeud, dont toutes les clés sont inférieures strictement à la clé.
	 * @return le fils gauche ou null s'il n'y en a pas.
	 */
	INode getLeft();
	/**
	 * Le fils droit de ce noeud, dont toutes les clés sont supérieures strictement à la clé.
	 * @return le fils gauche ou null s'il n'y en a pas.
	 */
	INode getRight();
	/**
	 * Teste l'existence d'une clé particulière dans le sous arbre dont ce noeud est la racine.
	 * L'exploration compareTo la clé "s" à la clé courante, et décide si on peut répondre vrai,
	 * où s'il faut poursuivre l'exploration dans le fils gauche ou le fils droit ( attention aux fils à null).
	 * @param s la clé recherchée
	 * @return vrai si un node porte cette clé, faux sinon.
	 */
	boolean contains(String s);
	/**
	 * Ajoute une chaine à l'arbre.
	 * Crée un nouveau noeud et l'ajout à la bonne position s'il n'y a pas encore de noeud portant cette clé.
	 * @param s la clé à insérer
	 * @return vrai si la valeur a été insérée (avec création d'un Node), faux sinon (la clé existait déjà )
	 */
	boolean add (String s);
}
```

On fournit également un test basique `abr.TestNodes` dans l'archive, correspondant aux scénarios décrit dans la figure 1.

Implantez la classe `pobj.abr.Node` qui réalise `INode`. Vous munirez la classe d'un constructeur `public Node (String value, INode left, INode right)`.

La recherche `contains` dans un arbre binaire d'un nœud ayant une clé particulière est un procédé récursif. On commence par examiner la racine. Si sa clé est la clé recherchée, l'algorithme se termine et renvoie la racine. Si elle est strictement inférieure, alors elle est dans le sous-arbre gauche, sur lequel on effectue alors récursivement la recherche. De même si la clé recherchée est strictement supérieure à la clé de la racine, la recherche continue dans le sous-arbre droit. Si on atteint une feuille dont la clé n'est pas celle recherchée, on sait alors que la clé recherchée n'appartient à aucun nœud, elle ne figure donc pas dans l'arbre de recherche.

L'insertion d'une clé `add` commence par une procédure similaire : on cherche la clé du nœud à insérer ; lorsqu'on arrive à une feuille (sans avoir trouvé la clé), on ajoute un nouveau nœud comme fils de la feuille en comparant sa clé à celle de la feuille : si elle est inférieure, le nouveau nœud sera à gauche ; sinon il sera à droite.

Fournir `src/abr/Node.java`.

b) (1,5 point)

A présent, on souhaite pouvoir afficher toutes les valeurs que contient un arbre, dans l'ordre. Écrivez une méthode qui prenne un `java.io.PrintStream` (e.g. comme `System.out`) et y construit le contenu d'un ABR. On demande de séparer les valeurs par un espace, et on tolère un espace à la fin de la chaîne.

Ceci mène à définir l'interface `INode2` :

INode2.java

```java
package abr;

import java.io.PrintStream;

public interface INode2 extends INode {
	void print(PrintStream pw);
}
```

Modifiez votre classe `pobj.abr.Node` afin qu'elle réalise `INode2` et implantez cette nouvelle méthode.

Corrigé :

Node.java

```java
package abr;

import java.io.PrintStream;

public class Node implements INode2 {
	private final String val;
	private INode2 left;
	private INode2 right;

	public Node(String val, Node left, Node right) {
		this.val = val;
		this.left = left;
		this.right = right;
	}

	public Node(String s) {
		this.val = s;
	}

	@Override
	public String getValue() {
		return val;
	}

	@Override
	public boolean add(String s) {
		int k = val.compareTo(s);
		if (k==0) {
			return false;
		} else if (k < 0) {
			if (right == null) {
				right = new Node(s);
				return true;
			} else {
				return right.add(s);
```

```
                }
            } else {
                if (left == null) {
                    left = new Node(s);
                    return true;
                } else {
                    return left.add(s);
                }
            }
        }

        @Override
        public boolean contains (String s) {
            int k = val.compareTo(s);
            if (k==0) {
                return true;
            } else if (k < 0) {
                if (right == null) {
                    return false;
                } else {
                    return right.contains(s);
                }
            } else {
                if (left == null) {
                    return false;
                } else {
                    return left.contains(s);
                }
            }
        }

        public INode getLeft() {
            return left;
        }

        public INode getRight() {
            return right;
        }

        @Override
        public void print(PrintStream pw) {
            if (left != null)
                left.print(pw);
            pw.append(val).append(" ");
            if (right != null)
                right.print(pw);
        }
}
```

Exercice 4 : Abonnés, Abonnements (6 points)

Notions testées : Collection, Map, Set, manipulation et conversions, exceptions.

Important : toutes les classes de cet exercice seront placées dans le package `abonne`. Lisez toute la partie a) avant de répondre, il y a des indications importantes sur la

notation et les contraintes à respecter. La partie b) est indépendante.

a) (4,5 points)

Dans une application de réseau social type Twitter, on considère la mise en place d'un mécanisme permettant d'enregistrer et de retrouver des abonnements. Une personne peut être abonnée à un nombre arbitraire d'autres personnes, et réciproquement une personne peut être suivie par un nombre arbitraire d'abonnés.

Par abus de langage on parle des abonnés (respectivement abonnements) d'une personne pour désigner *l'ensemble des personnes* qui le suivent (respectivement qu'il suit).

Par exemple (NB: on fournit également un test basique dans l'archive, correspondant à ce scénario), si l'on considère trois personnes a, b, c. On ajoute trois abonnements $(a, b), (a, c), (b, c)$.

- On obtient que les abonnés de a sont $\{b, c\}$, les abonnés de b sont $\{c\}$ et les abonnés de c est l'ensemble vide $\emptyset$.
- On dit que l'*ensemble des suivis* est $\{a, b\}$ (car c n'a pas d'abonné).
- Réciproquement, les abonnements de a sont l'ensemble vide $\emptyset$, les abonnements de b est $\{a\}$ et les abonnements de c sont $\{a, b\}$.
- On dit que l'*ensemble des suiveurs* est $\{b, c\}$ (car a n'a pas d'abonnement).

Si l'on retire alors l'abonnement (b, c),

- Les abonnés de a sont $\{b, c\}$, les abonnés de b et de c sont l'ensemble vide $\emptyset$.
- L'*ensemble des suivis* est $\{a\}$
- Réciproquement, les abonnements de a sont l'ensemble vide $\emptyset$, et les abonnements de b et de c sont $\{a\}$.
- L'*ensemble des suiveurs* est $\{b, c\}$

L'interface `IAbonnements` est donnée ici (et fournie dans l'archive): Dans une classe `abonne.Abonnements`, implantez cette interface.

Attention à bien respecter la spécification (javadoc) de l'interface, en particulier sur les valeurs de retour booléennes et les ensembles vides. On pourra utiliser les collections vides disponibles dans `Collections.emptyXXXX()` pour matérialiser les ensembles vides.

Bien que de nombreuses réalisations soient envisageables, il y a deux principales possibilités.

La solution recommandée est de s'appuyer sur deux `Map<String,Set<String> >`, associant aux noms de personnes respectivement un ensemble d'abonnés ou d'abonnements. Cette solution garantit la complexité $O(1)$ en temps sur toutes les opérations de `IAbonnements`. Attention à ne pas associer des clés à des ensembles vides dans vos Map (à contrôler lors du `remAbonnement`).

On peut aussi implémenter une solution plus naïve qui stocke simplement les paires correspondant aux abonnements, et qui parcourt cet ensemble de paires pour répondre à chaque requête. La complexité est donc $O(n)$ a priori.

Dans tous les cas, comme beaucoup d'opérations sont symétriques l'une de l'autre, il peut être utile de définir des sous-fonctions pour éviter de dupliquer le code.

80% des points de cette question portent sur des tests purement fonctionnels, que les deux solutions peuvent valider. 20 % des points de cette question porteront sur un test de performance cherchant à valider que la solution O(1).

Le barème prévoit d'abord des scénarios qui testent le fonctionnement sans retirer d'abonnements, avant de tester que le comportement reste correct malgré l'ajout et la suppression dans un ordre arbitraire d'abonnements. Donc on pourra valider des points en ne traitant que l'enregistrement.

Fournir `src/abonne/Abonnements.java`.

IAbonnements.java

```
package abonne;

import java.util.Collection;

```

```java
/**
 * Représente un ensemble d'abonnements.
 */
public interface IAbonnements {
	/**
	 * Enregistre un abonnement : suiveur est abonné de suivi ; suivi est ajouté aux abonnements de suiveur.
	 * @param suivi le nom de la personne suivie
	 * @param suiveur le nom de la personne qui s'abonne
	 * @return true si l'abonnement n'existait pas encore, false sinon.
	 */
	public boolean addAbonnement (String suivi, String suiveur);
	/**
	 * Efface un abonnement : suiveur n'est plus abonné de suivi ; suivi est retiré des abonnements de suiveur.
	 * @param suivi le nom de la personne suivie
	 * @param suiveur le nom de la personne qui s'abonne
	 * @return true si l'abonnement existait et a été détruit, false (et aucun effet) sinon.
	 */
	public boolean remAbonnement (String suivi, String suiveur);
	/**
	 * Calcule la liste des abonnés d'une personne.
	 * @param suivi le nom de la personne suivie
	 * @return l'ensemble (sans doublon) des noms des personnes abonnées à suivi
	 */
	public Collection<String> getAbonnes (String suivi);
	/**
	 * Calcule la liste des abonnements d'une personne, i.e. l'ensemble des personnes qu'elle suit.
	 * @param suiveur le nom de la personne qui s'abonne
	 * @return l'ensemble (sans doublon) des noms des personnes auquel suiveur est abonné
	 */
	public Collection<String> getAbonnements (String suiveur);
	/**
	 * Calcule l'ensemble (SANS doublon) des noms de toute personne qui a au moins un abonné.
	 */
	public Collection<String> getAllSuivis();
	/**
	 * Calcule l'ensemble (SANS doublon) des noms de toute personne abonnée à au moins une autre personne.
	 */
	public Collection<String> getAllSuiveurs();
}
```

Corrigé :

Abonnements.java

```java
package abonne;

import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

```

```java
import abonne.IAbonnements;

public class Abonnements implements IAbonnements {

	private Map<String, Set<String>> abonnes = new HashMap<>();
	private Map<String, Set<String>> abonnements = new HashMap<>();

	private boolean addPair (String a, String b, Map<String, Set<String>> map ) {
		Set<String> set = map.get(a);
		if (set == null) {
			set = new HashSet<>();
			map.put(a, set);
		}
		return set.add(b);
	}

	@Override
	public boolean addAbonnement(String suivi, String suiveur) {
		boolean ret = addPair(suivi, suiveur, abonnes);
		if (ret) addPair(suiveur, suivi, abonnements);
		return ret;
	}

	@Override
	public boolean remAbonnement(String suivi, String suiveur) {
		boolean ret = remPair(suivi, suiveur, abonnes);
		if (ret) remPair(suiveur, suivi, abonnements);
		return ret;
	}

	private boolean remPair(String a, String b, Map<String, Set<String>> map) {
		Set<String> set = map.get(a);
		if (set == null) {
			return false;
		}
		boolean ret = set.remove(b);
		if (ret && set.isEmpty()) {
			map.remove(a);
		}
		return ret;
	}

	@Override
	public Collection<String> getAbonnes(String suivi) {
		return getSet(suivi, abonnes);
	}

	private Collection<String> getSet(String a, Map<String, Set<String>> map) {
		Set<String> set = map.get(a);
		if (set == null) {
			return Collections.emptySet();
		}
		return set;
	}

	@Override
	public Collection<String> getAbonnements(String suiveur) {
		return getSet(suiveur,abonnements);
	}
```

```java
	@Override
	public Collection<String> getAllSuivis() {
		return abonnes.keySet();
	}

	@Override
	public Collection<String> getAllSuiveurs() {
		return abonnements.keySet();
	}

}
```

Barème:

10 % implements IAbo

40 % fonctions de base, sans remove, fonctionnellement correct

30 % malgré le remove, fonctionnellement correct (attention ensembles vides...)

20 % implém en complexité $< O(n)$

b) (1,5 points)

On vous fournit le source d'une classe permettant de charger des fichiers contenant des abonnements. Il est prévu que cette procédure puisse produire trois types d'erreurs selon que le fichier n'existe pas, qu'il est vide, ou qu'il est mal formaté (cf. le `main` fourni pour le nomage de ces exceptions).

Le parser est déjà presque fonctionnel, mais il manque la gestion des exceptions. Ajoutez donc les instructions utiles (aux endroits notés *TODO*) et définissez les exceptions manquantes pour que le main puisse fonctionner.

ChargeurFichier.java

```java
package abonne;

import java.io.File;
import java.util.Scanner;

public class ChargeurFichier {

	public static int charge(String nom, IAbonnements abos) throws AbonnementException {
		Scanner sc;
		// erreur si le fichier n'existe pas : lève une IOException.
		// TODO : convertir en BadAbonnementFileName
		sc = new Scanner(new File(nom));

		int done = 0;
		while (sc.hasNextLine()) {
			String line = sc.nextLine();
			String [] words = line.trim().split(":");
			if (words.length != 2) {
				sc.close();
				// erreur : format de fichier incorrect
				// TODO : lever BadAbonnementFormat

			} else {
				abos.addAbonnement(words[0], words[1]);
				done++;
			}
		}
		sc.close();
```

```
            if (done == 0) {
                  // erreur : fichier vide
                  // TODO : lever EmptyAbonnementFile
            }
            return done;
      }

      public static void main(String[] args) {
            try {
                  IAbonnements a = new Abonnements();
                  int taille = charge("abo.txt", a);
                  System.out.println("Lu : "+ taille);
            } catch (BadAbonnementFileName e) {
                  System.err.println("Mauvais fichier !");
            } catch (BadAbonnementFormat e) {
                  System.err.println("Mauvais format !");
            } catch (EmptyAbonnementFile e) {
                  System.err.println("Fichier vide !");
            } catch (AbonnementException e) {
                  e.printStackTrace();
            }
      }
}
```

Fournir dans `src/abonne/` les fichiers java définissant les exceptions utiles.
Modifier `src/abonne/ChargeurFichier.java` pour lever les exceptions appropriées.

ChargeurFichier.java

```
package abonne;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ChargeurFichier {

      public static int charge(String nom, IAbonnements abos) throws AbonnementException
            {
             Scanner sc;
             // erreur si le fichier n'existe pas
             try {
                   sc = new Scanner(new File(nom));
             } catch (FileNotFoundException e) {
                   throw new BadAbonnementFileName();
             }

             int done = 0;
             while (sc.hasNextLine()) {
                   String line = sc.nextLine();
                   String [] words = line.trim().split(":");
                   if (words.length != 2) {
                         sc.close();
                         // erreur : format de fichier incorrect
                         throw new BadAbonnementFormat();
                   } else {
                         abos.addAbonnement(words[0], words[1]);
                         done++;
                   }
```

```
            }
            sc.close();
            if (done == 0) {
                  // erreur : fichier vide
                  throw new EmptyAbonnementFile();
            }
            return done;
      }

      public static void main(String[] args) {
            try {
                  IAbonnements a = new Abonnements();
                  int taille = charge("abo.txt", a);
                  System.out.println("Lu : "+ taille);
            } catch (AbonnementException e) {
                  e.printStackTrace();
            }
      }

}
```