Conception/Programmation OO - Corrigé

SUPAERO 2A

Christophe Garion <garion@supaero.fr>

2 mars 2004

Voici un corrigé possible de l'examen de conception/programmation orientées objet.

Exercice 1

1.1 Présentation du problème

On désire modéliser un système d'authentification d'utilisateurs souhaitant accéder à des documents classifiés selon différents niveaux.

Les utilisateurs du système sont représentés par un nom et un mot de passe qui sont des chaînes de caractères. Le mot de passe est une chaîne de caractères particulière qui peut être cryptée ou décryptée grâce à une clé (une chaîne de caractères).

Un utilisateur peut être habilité « Confidentiel », « Secret » ou ne pas être habilité. De plus, un utilisateur habilité « Secret » est également habilité « Confidentiel ».

Le système informatique possède trois serveurs caractérisés par le type de la machine et son système d'exploitation. Chaque machine possède un certain nombre de disques durs permettant de stocker des données. Ces disques peuvent être montés ou non (i.e. accessibles sur le système de fichier de l'ordinateur ou pas).

Le premier serveur est un serveur d'authentification et contient en particulier un fichier contenant les noms et mots de passe des utilisateurs sous forme d'une même entité appelée « login ». Le second serveur contient un système de fichiers représentant les répertoires « home » de chaque utilisateur du système et un système de fichiers contenant les applications du système. Le troisième serveur contient les documents classifiés, qui sont des fichiers particuliers possédant un degré de confidentialité.

Chaque fichier est stocké sur un disque dur particulier.

Un scénario particulier de récupération d'un document classifié « Confidentiel »par un utilisateur extérieur est le suivant :

- l'utilisateur se connecte au serveur d'authentification. Celui-ci lui demande alors son login et mot de passe ;
- l'utilisateur envoie son login et son mot de passe crypté au serveur ;
- celui-ci vérifie alors que le login et le mot de passe sont corrects ;
- le serveur d'authentification indique au système que l'utilisateur est bien « valide » en ouvrant une session sur le système pour l'utilisateur ;
- le système envoie un signal à l'utilisateur pour lui signaler qu'il est en attente d'une commande de sa part ;
- l'utilisateur demande au système l'accès et la sauvegarde d'un document classifié sur son compte ;
- le système demande la classification du document au serveur les contenant, puis vérifie que l'utilisateur est habilité à récupérer ce type de document ;
- l'utilisateur demande alors au serveur de documents de transférer le document sur son compte et celui-ci effectue l'opération.

1.2 Questions

1. représenter sous forme d'un diagramme de séquence le scénario présenté en section 1.1. Vous supposerez que vous disposez dans chaque classe d'opérations au nom explicite ;

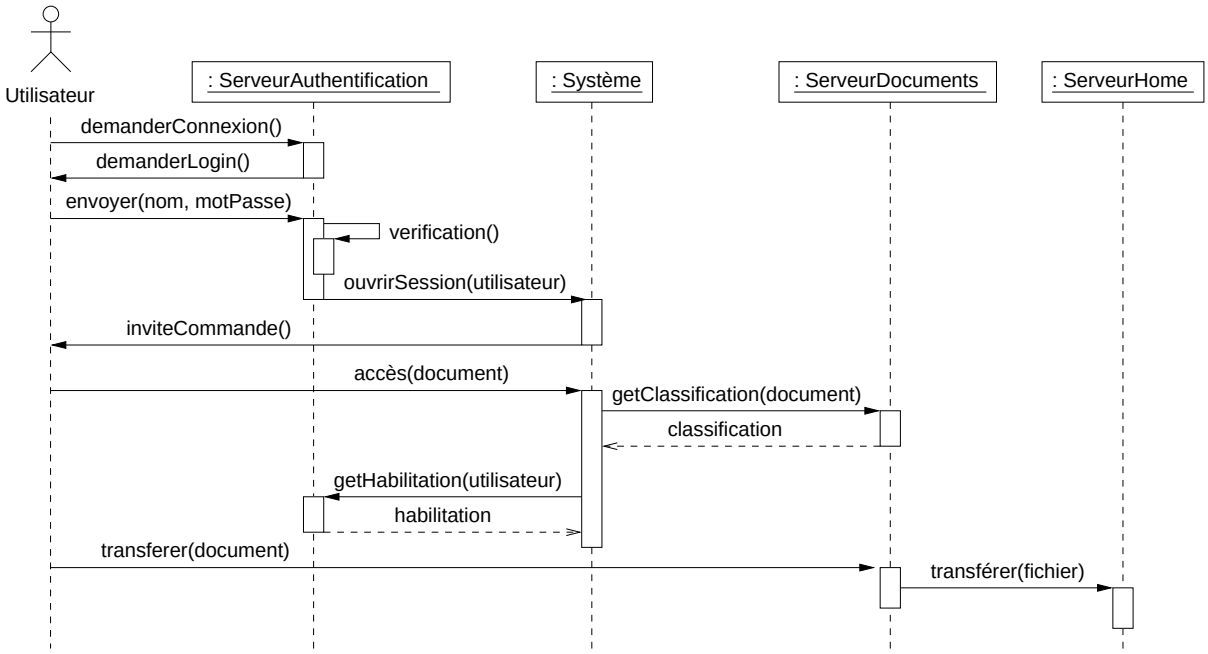

FIG. 1 – Diagramme de séquence

Un diagramme de séquence possible est représenté sur la figure 1.

Quelques remarques :
- j'ai choisi de représenter un objet `Système` qui représentait le système (au sens informatique). On aurait très bien pu considérer que le système se situait sur le serveur d'applications (représenté ici par `ServeurHome`) ;
- les flèches en pointillés correspondent au retour des appels de méthodes ;
- j'ai considéré que l'habilitation des utilisateurs était stockée sur le serveur d'authentification ;
- on aurait pu également faire intervenir un objet `Utilisateur` (voire `Login` également) pour vraiment raffiner le modèle.

2. proposer un diagramme *UML* de conception préliminaire (analyse, donc sans attributs ni méthodes) de l'ensemble des éléments décrits dans l'énoncé présentant les classes, les relations entre les classes, les éventuels rôles et multiplicités (ou cardinalités).
   Vous pourrez justifier par écrit les relations utilisées et modifier de façon mineure l'énoncé si celui-ci vous paraît ambigu ;

   Un diagramme de classe possible est représenté sur la figure 2.

   Quelques remarques :
   - **ce n'est évidemment pas LA solution** ;
   - les relations d'héritage entre les différents types d'utilisateurs se déduisaient assez naturellement de l'énoncé ;
   - j'ai choisi d'utiliser une classe `Login` pour faciliter la représentation (en particulier le lien avec le fichier de login) ;
   - les liens entre `Serveur`, `Machine`, `DisqueDur` étaient faciles à trouver. Je considère ici (agrégation) que si une machine est détruite, on peut toujours récupérer ses disques durs. Un système d'exploi-

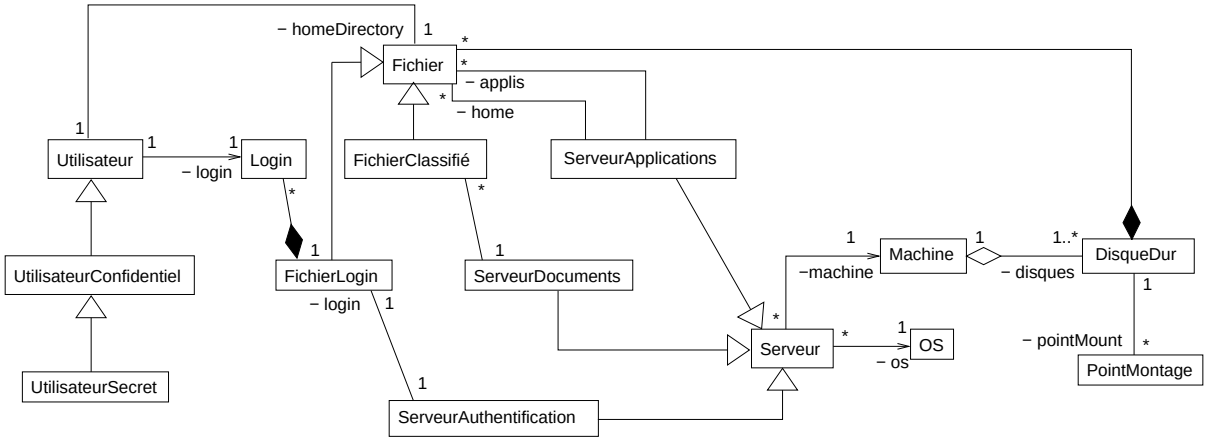

FIG. 2 – Diagramme de classe représentant le système

tation peut être installé sur plusieurs machines et une machine peut abriter plusieurs serveurs. Le seul point difficile était le point de montage du disque, que j'ai représenté par une classe ;
– les trois serveurs spécialisés étaient faciles à trouver. Les liens avec les différents fichiers également. Les fichiers sont rattachés à un disque dur et si le disque est détruit, les fichiers le sont également.

3. proposer un diagramme de conception détaillée (attributs et opérations typés) de la classe `Utilisateur`. Ce diagramme devra faire apparaître l'implantation des relations existant avec les autres classes. Vous vous limiterez à la construction d'un petit nombre d'opérations sur cette classe ;

Un diagramme possible est représenté sur la figure 3.

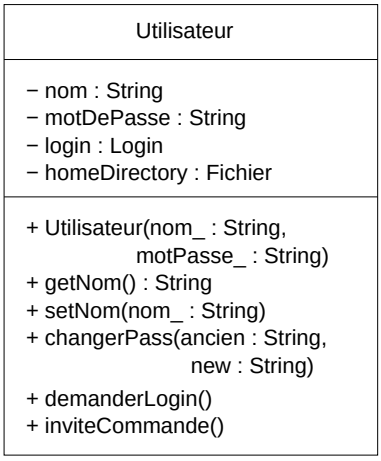

FIG. 3 – Diagramme de la classe `Utilisateur`

Rien de bien particulier ici. Les attributs correspondent au nom et au mot de passe de l'utilisateur, à l'objet de type `Login` et à celui de type `Fichier` liés par les associations représentées sur la figure 2. Le constructeur n'initialise que le nom et le mot de passe, le login et le répertoire de base étant construits à la volée.

Les méthodes « classiques » que sont les accesseurs et modifieurs n'avaient de sens que pour le nom.

On ne doit pas pouvoir accéder au mot de passe directement et on doit faire appel à une méthode particulière pour le changer (en particulier, celle ci doit vérifier l'ancien mot de passe et en demander un nouveau).

Les deux dernières méthodes se déduisaient du diagramme de séquence construit dans la première question.

4. le système informatique possède un objet `Dispatcher` qui s'occupe de répartir les requêtes. Pour éviter des problèmes d'encombrements et de synchronisation, on souhaiterait qu'il ne soit possible de créer qu'un seul `Dispatcher` pour le système. Proposer une solution simple.

   C'était évidemment la question la plus difficile de cette partie. On peut (va) utiliser un attribut statique pour vérifier si l'objet a déjà été créé ou pas (un **`boolean`** par exemple). Comme l'attribut est statique, il est indépendant de toute instance et on « conserve » la valeur de cet attribut, car elle est associée à la classe.

   Le gros problème qui se pose est l'utilisation du constructeur. Si l'on utilise le constructeur de la classe pour créer un objet de type `Dispatcher`, on créera de toute façon l'objet (il n'y a aucun moyen d'éviter cela, à part arrêter le programme). Il faut donc rendre le constructeur privé pour éviter une utilisation sans contrôle.

   Reste le problème de la création d'une instance de `Dispatcher`. On va utiliser une méthode statique (car elle ne manipule pas d'instance de la classe) qui va vérifier que le booléen est bien faux (aucun dispatcher de créé pour l'instant) et qui dans ce cas va faire appel au constructeur de la classe (possible même s'il est privé) et va renvoyer l'objet.

   Tout cela est présenté sur la figure 4.

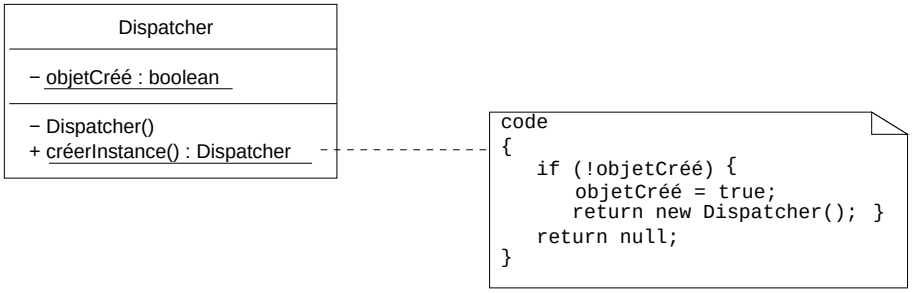

FIG. 4 – Diagramme de classe de `Dispatcher`

Pour créer un objet de type `Dispatcher`, on va donc faire un appel à `créerInstance`, par exemple : `Dispatcher d = Dispatcher.créerInstance();`. Si un dispatcher a déjà été créé, on récupère une référence **`null`** inutilisable. Cette solution est à rapprocher du *pattern* Singleton (cf. [2]).

Exercice 2

2.1 Présentation du problème

Lorsque l'on développe un logiciel, on devrait toujours tester son code. En particulier, chaque méthode d'une classe doit être testée de façon exhaustive, de façon à prouver que son comportement est bien celui précisé dans les spécifications de la classe (test unitaire). Malheureusement, un programmeur manque souvent de temps (ou d'envie...) pour effectuer ces vérifications qui sont nécessaires à l'obtention d'un code stable.

On va donc développer un *framework* de test adapté à un développement JAVA qui va nous permettre d'écrire des classes de test au fur et à mesure que l'on écrit les méthodes d'une classe[1]. Ce framework doit permettre aux développeurs d'écrire rapidement leurs tests.

De plus, un tel cadre de test doit être « persistant ». On doit pouvoir reproduire facilement un test 5 ans après que le programme ait été écrit. Par ailleurs, on doit pouvoir réutiliser des tests existants de façon simple. Enfin, ces tests devront être exécutés automatiquement.

2.2 Questions

1. on considère que l'on développe une classe `Monnaie` qui représente un certain montant d'argent dans une devise particulière.

   Le diagramme UML de la classe `Monnaie` est présenté sur la figure 5.

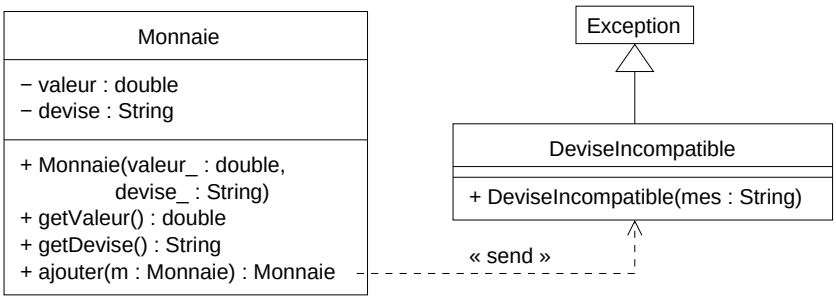

FIG. 5 – La classe `Monnaie`

   La classe `Monnaie` possède :
   - un attribut `valeur` qui est un réel ;
   - un attribut `devise` qui est une chaine de caractères ;
   - un constructeur prenant comme paramètres un double et une chaines de caractères ;
   - une méthode `getValeur()` qui renvoie la valeur de l'objet ;
   - une méthode `getDevise()` qui renvoie la devise de l'objet ;
   - une méthode `ajouter(Monnaie m)` qui permet d'« ajouter » un autre objet de type `Monnaie` à l'objet courant. Cette méthode renvoie une objet de type `Monnaie` correspondant à l'addition. Cette méthode peut lever une exception de type `DeviseIncompatible` si la devise de l'objet passé en paramètre n'est pas la même que celle de l'objet courant.

   Écrire le code de la classe `Monnaie` (ne pas oublier que les chaînes de caractères sont des instances de la classe `String`).

   Voici le code de la classe `Monnaie` :

```
/**
 * <code>Monnaie</code> représente une certaine valeur d'argent pour une
 * devise particulière.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class Monnaie {
```

[1] Une pratique encore meilleure serait d'écrire tout d'abord les tests (ce qui permet de bien comprendre comment fonctionne la méthode), puis d'écrire le code de la méthode correspondante avant de la tester effectivement.

```
    private double valeur;

    private String devise;

    /**
     * Créer une instance de <code>Monnaie</code>.
     *
     * @param valeur_ le montant
     * @param devise_ la devise
     */
    public Monnaie(double valeur_, String devise_) {
        this.valeur = valeur_;
        this.devise = devise_;
    }

    /**
     * Renvoyer la valeur de l'instance.
     *
     * @return la valeur
     */
    public double getValeur() {
        return this.valeur;
    }

    /**
     * Renvoyer la devise de l'instance.
     *
     * @return la devise (qui est une instance de <code>String</code>
     */
    public String getDevise() {
        return this.devise;
    }

    /**
     * <code>ajouter</code> permet d'ajouter deux instances de
     * <code>Monnaie</code>.
     *
     * @param m l'instance de <code>Monnaie</code> à ajouter
     * @return l'instance de <code>Monnaie</code> résultat
     * @exception DeviseIncompatible si les deux devises sont différentes
     */
    public Monnaie ajouter(Monnaie m) throws DeviseIncompatible {
        if (getDevise().compareTo(m.getDevise()) != 0) {
            throw new DeviseIncompatible("Problème de devise");
        } // end of if (!getDevise().equals(m.getDevise()))
        return new Monnaie(getValeur() + m.getValeur(), getDevise());
    }
}
```

Il n'y avait pas de problème particulier pour écrire la classe. Les deux seules petites difficultés que l'on pouvait trouver étaient les suivantes :

- la devise étant représentée par un objet de type `String`, on ne pouvait pas utiliser directement `==` pour comparer les devises. Il fallait utiliser la méthode `compareTo` disponible pour les instances de `String` ;
- ne pas oublier de déclarer l'exception dans la signature de la méthode `ajouter`.

Écrire également le code de la classe `DeviseIncompatible`.

Voici le code de la classe `DeviseIncompatible` :

```
/**
 * <code>DeviseIncompatible</code> est une exception lancée si on ne peut
 * pas ajouter deux instances de monnaie car leurs devises sont incompatibles.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class DeviseIncompatible extends Exception {
    public DeviseIncompatible(String mes) {
        super(mes);
    }
}
```

Aucune difficulté, il fallait juste penser au **super** dans le constructeur.

2. écrire deux classes de tests :
   - une première classe, `TestBasique`, qui va :
     - créer une instance de `Monnaie` qui correspond à 5 dollars ;
     - vérifie « à la main » que la valeur de cette instance est 5 ;
     - vérifie « à la main » que la devise de cette instance est dollars ;

   Voici le code source de la classe `TestBasique` :

```
/**
 * <code>TestBasique</code> teste de façon rudimentaire la classe
 * <code>Monnaie</code>.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TestBasique {
    public static void main(String[] args) {
        Monnaie m = new Monnaie(5, "dollars");
        if (!(m.getValeur() == 5)) {
            System.out.println("Erreur : la valeur n'est pas bonne");
        } // end of if (!(m.getValeur() == 5))
        if (((m.getDevise()).compareTo("dollars")) != 0) {
            System.out.println("Erreur : la devise n'est pas bonne");
        } // end of if (((m.getDevise()).equals("dollars")))
    } // end of main()
}
```

– une seconde classe, `TestAdd`, qui va :
  – créer une instance de `Monnaie` qui correspond à 10 euros ;
  – créer une instance de `Monnaie` qui correspond à 3 euros ;
  – obtenir une instance de `Monnaie` qui correspond à l'addition des deux premiers objets créés ;
  – vérifier « à la main » que l'objet ainsi obtenu correspond bien à ce que l'on attend.

Voici le code source de la classe `TestAdd` :

```
/**
 * <code>TestAdd</code> teste de façon rudimentaire l'ajout de deux
 * instances de <code>Monnaie</code>.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TestAdd {
    public static void main(String[] args) {
        Monnaie m1 = new Monnaie(10, "euros");
        Monnaie m2 = new Monnaie(3, "euros");

        try {
            Monnaie m = m1.ajouter(m2);

            if (!(m.getValeur() == 13)) {
                System.out.println("Erreur : la valeur n'est pas bonne");
            } // end of if (!(m.getValeur() == 13))
            if (((m.getDevise()).compareTo("euros")) != 0) {
                System.out.println("Erreur : la devise n'est pas bonne");
            } // end of if (((m.getDevise()).equals("euros")))
        } catch (DeviseIncompatible e) {
            System.out.println("Problème avec les devises : il ne devrait " +
                               " pas arriver !");
            e.printStackTrace();
        } // end of try-catch
    } // end of main()
}
```

Il ne fallait pas oublier que `ajouter` pouvait lever une exception de type `DeviseIncompatible`. Le traitement de cette exception est minimale ici : on affiche juste un message, car cette exception ne devrait pas être levée dans le programme de test (les deux instances de `Monnaie` ont des devises indentiques). Cela permettait également de vérifier que le test sur les devises est correct.

3. écrire des tests de la façon présentée précédemment est assez fastidieux et ne répond pas aux exigences que nous avions émises en section 2.1. Pour pouvoir disposer d'un cadre commun à tous les tests, on va donc utiliser des objets représentant des tests particuliers et créer une classe regroupant les caractéristiques communes à ces tests. On appelera cette classe `TestCase`.

   L'état d'un objet de type `TestCase` va être caractérisé par une chaine de caractères représentant le nom du test. Pour l'instant, cela nous suffit.

   Un test est avant tout une opération. La classe `TestCase` va donc contenir une méthode `run` qui va effectuer le test en lui-même. Normalement, cette méthode devrait être abstraite, car on devra spécialiser la classe pour décrire le test qui nous intéresse. Il n'y a donc aucun intérêt à créer un

objet de type `TestCase`. Un premier diagramme UML représentant la classe est présenté sur la figure 6 (les classes et méthodes en italique sont abstraites).

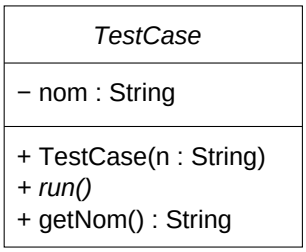

FIG. 6 – Première version de la classe `TestCase`

Pour respecter les exigences que nous avons imposées sur un *framework* de test (cf. section 2.1), il nous faut donner à l'utilisateur de `TestCase` un moyen d'écrire son code de test facilement et proprement. En particulier, on peut décomposer le test en trois phases distinctes :
– une phase de préparation du test (construction des objets nécessaires etc.) ;
– la phase de test à proprement parler ;
– une phase de « nettoyage ».
Ces trois phases peuvent être représentées par trois méthodes de la classe `TestCase` qu'on appellera respectivement `setUp()`, `runTest()` et `tearDown()`. La méthode `run` va donc utiliser ces trois méthodes de la façon représentée sur le diagramme 7.

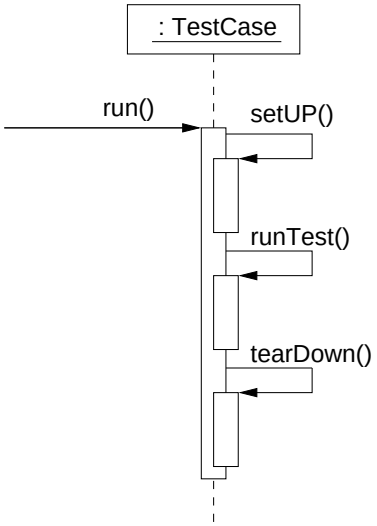

FIG. 7 – Diagramme de séquence représentant le déroulement de la méthode `run()`

Il faut maintenant pouvoir collecter de façon efficace les résultats du test. En particulier, il faut connaître quels sont les tests qui ont échoué et quels sont ceux qui ont réussi. Pour cela, on va supposer que l'on dispose :
– d'une classe `TestException` qui représente une exception qui est levée lorsqu'un test échoue ;
– d'une méthode de classe `estVraie(boolean condition)` dans `TestCase` qui lève une exception de type `TestException` si la condition passée en paramètre n'est pas vraie ;
– d'une classe `TestResult` qui possède une méthode `addTestFailure(TestCase t, TestException test)`

qui stocke le fait qu'un test se soit mal déroulé et `addTestSuccess(TestCase)` qui stocke le fait qu'un test se soit déroulé correctement.

Le diagramme de classe final est présenté sur la figure 8.

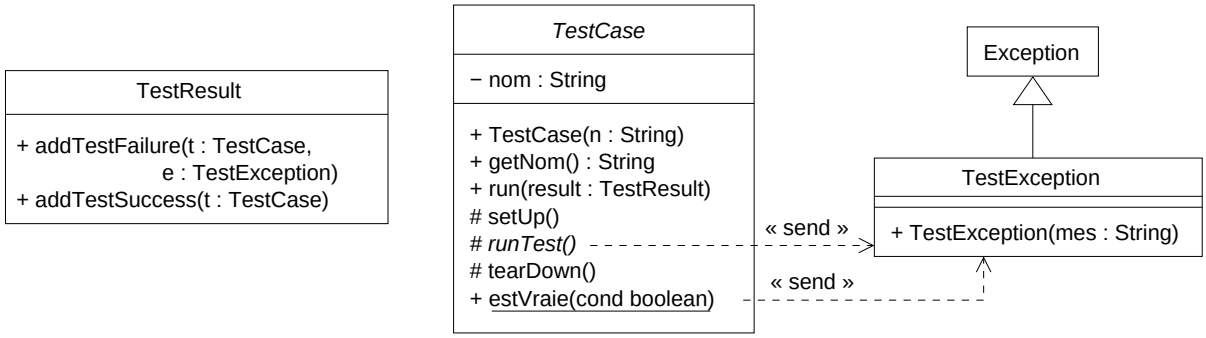

FIG. 8 – Diagramme de classe « final »

(a) pourquoi les méthodes `setUp`, `runTest` et `tearDown` ont-elles un droit d'accès « protégé » ?

Ces trois méthodes décomposent en fait le déroulement de la méthode `run`, qui représente le déroulement du test. Elles devraient donc être a priori privées, car elles ne sont pas autosuffisantes (on ne doit pas pouvoir les exécuter indépendamment les unes des autres).

Ici, on va spécialiser la classe `TestCase` pour réaliser des tests particuliers. Ces tests seront définis par les trois méthodes précédentes. On va donc positionner le droit d'accès à ces méthodes à « protégé » pour autoriser les sous-classes à redéfinir les méthodes `setUp`, `runTest` et `tearDown`. On peut remarquer qu'avec JAVA, les méthodes pourront également être appelées dans une classe du paquetage (ce qui peut être un peu génant).

(b) pourquoi à votre avis seule la méthode `runTest` est abstraite et pas `setUp` et `tearDown` (on supposera que ces méthodes ne font rien dans la classe `TestCase`) ?

Si les méthodes `setUp` et `tearDown` étaient abstraites, on devrait les redéfinir à chaque fois que l'on spécialise la classe `TestCase` pour écrire une classe de test concrète. Or ces deux méthodes servent à initialiser le test et à le nettoyer une fois effectué. Dans la plupart des cas, on ne s'en servira pas. On autorise donc l'utilisateur à ne pas redéfinir ces méthodes et à utiliser les méthodes de `TestCase` qui ne font rien. Par contre, `runTest` est le cœur du test et doit être obligatoirement redéfinie à chaque fois, donc on la définit comme abstraite.

(c) écrire le code de la classe `TestCase` (vous ne devez pas modifier la structure des classes présentées sur le diagramme 8).

Voici le code de la classe `TestCase` :

```
/**
 * <code>TestCase</code> est une classe abstraite représentant un test
 * générique.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public abstract class TestCase {
```

```
private String nom;

/**
 * Créer une instance de <code>TestCase</code>.
 *
 * @param n le nom du test
 */
public TestCase(String n) {
    this.nom = n;
}

/**
 * Le nom du test.
 *
 * @return une instance de <code>String</code> qui est le nom du test
 */
public String getNom() {
    return this.nom;
}

/**
 * <code>run</code> décrit le déroulement du test : setUp, runTest,
 * tearDown et enregistrement.
 *
 * @param res une instance de <code>TestResult</code> qui sert
 *            d'enregistrement
 */
public void run(TestResult res) {
    this.setUp();
    try {
        this.runTest();
        res.addTestSuccess(this);
    } catch (TestException e) {
        res.addTestFailure(this, e);
    } finally {
        this.tearDown();
    }// end of try-finally
}

/**
 * <code>setUp</code> décrit la préparation du test. Par défaut, elle ne
 * fait rien.
 *
 */
protected void setUp() {
}

/**
 * <code>runTest</code> décrit le déroulement du test en lui-même.
 *
```

```
     * @exception TestException si une erreur arrive lors d'un test
     */
    protected abstract void runTest() throws TestException;

    /**
     * <code>tearDown</code> décrit le nettoyage du test. Par défaut, elle ne
     * fait rien.
     *
     */
    protected void tearDown() {
    }

    /**
     * <code>estVraie</code> renvoie une exception si la condition est fausse.
     *
     * @param cond la condition à tester
     * @exception TestException l'exception levée si la condition est fausse
     */
    public static void estVraie(boolean cond) throws TestException {
        if (!cond) {
            throw new TestException("Problème dans le test");
        } // end of if (!cond)
    }
}
```

Rien de bien particulier ici. Il fallait bien comprendre que le fait qu'une exception soit levée par `runTest` nous permettait d'arrêter le test et d'inscrire l'échec dans l'objet de type `TestResult` passé en paramètre de `run`. Dans le cas contraire, on inscrit le succès. J'ai choisi d'exécuter `tearDown` dans le bloc **`finally`** pour être sûr de nettoyer le test même en cas d'échec.

La petite subtilité venait de `setUp` et de `tearDown` qui ne font rien (leur corps est vide), mais qui ne sont pas abstraites (cf. question précédente).

4. on va maintenant utiliser la classe développée précédemment pour tester la classe `Monnaie`. On va donc spécialiser la classe `TestCase` en une classe `TestMonnaie` qui va tester l'addition de deux objets de type monnaie ayant la même devise. On stockera ces objets en attributs de la classe et on les initialisera avec `setUp`. On ne redéfinira pas `tearDown`. Le diagramme de classe est présenté sur la figure 9.

   Écrire la classe `TestMonnaie`.

   Le code de la classe `TestMonnaie` est présenté ci-dessous.

   Attention, dans l'énoncé distribué, les méthodes `setUp` et `runTest` sont publiques au lieu d'être protégées. Il n'en sera bien entendu pas tenu compte dans la notation.

   Il fallait ne pas oublier de faire appel à **`super`** dans le constructeur de `TestMonnaie`. Le corps de `setUp` devait initialiser les deux attributs de la classe et `runTest` devait simplement utiliser `estVraie` pour vérifier que le test se déroulait bien.

   En conclusion de cet exercice, je vous renvoie à un *framework* de test complet qui a servi à concevoir ce problème : JUnit [1]. À noter que le modèle de conception s'appuie sur de nombreux *design patterns* que vous pouvez essayer de retrouver dans [2].

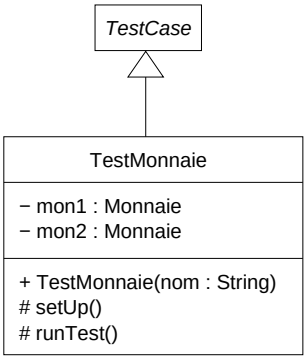

FIG. 9 – Diagramme de classe de `TestMonnaie`

```
/**
 * <code>TestMonnaie</code> permet de tester l'addition de deux objets de type
 * <code>Monnaie</code>.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TestMonnaie extends TestCase {

    private Monnaie mon1;

    private Monnaie mon2;

    /**
     * Créer une instance de <code>TestMonnaie</code>.
     *
     * @param nom le nom du test
     */
    public TestMonnaie(String nom) {
        super(nom);
    }

    /**
     * <code>setUp</code> initialise les deux instances de <code>Monnaie</code>
     * nécessaires au test.
     *
     */
    protected void setUp() {
        mon1 = new Monnaie(10, "euros");
        mon2 = new Monnaie(3, "euros");
    }

    /**
     * <code>runTest</code> teste effectivement si l'addition est correcte.
```

```
     *
     */
    protected void runTest() throws TestException {
        try {
            Monnaie m = mon1.ajouter(mon2);
            TestCase.estVraie(m.getValeur() == 13);
            TestCase.estVraie((m.getDevise()).compareTo("euros") == 0);
        } catch (DeviseIncompatible e) {
            e.printStackTrace();
        } // end of try-catch
    }
}
```

Références

[1] http://www.junit.org.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1994.