Examen de Conception/Programmation OO SUPAERO 2A

Christophe Garion <garion@supaero.fr> 2 mars 2004

Cet examen est composé de deux parties indépendantes. Nous vous conseillons d'y consacrer la même durée. Tous les documents sont autorisés.

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 ;

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 ;
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 ;
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.

Exercice 2

Remarques importantes : dans cet exercice, vous allez devoir écrire du code Java. Il est évident que les petites erreurs de syntaxe ne seront pas pénalisantes (qui peut se vanter d'écrire directement du code correct à chaque fois ?). De même, vous n'écrirez la documentation Javadoc d'une méthode ou d'une classe que si vous le jugez nécessaire (préciser la signification d'un paramètre ou du retour d'une méthode par exemple).

Enfin, le code à écrire n'est pas long, mais comprend quelques subtilités. Réfléchissez bien avant de vous lancer dans l'écriture.

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 1.

   La classe `Monnaie` possède :
   - un attribut `valeur` qui est un réel ;
   - un attribut `devise` qui est une chaîne de caractères ;
   - un constructeur prenant comme paramètres un double et une chaînes de caractères ;
   - une méthode `getValeur()` qui renvoie la valeur de l'objet ;

[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.

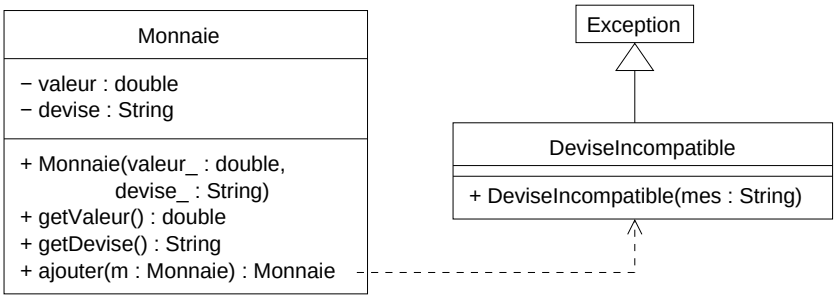

FIG. 1 – La classe `Monnaie`

- 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`). Écrire également le code de la classe `DeviseIncompatible`.

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 ;
   - 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.
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 chaîne 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 2 (les classes et méthodes en italique sont abstraites).

   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

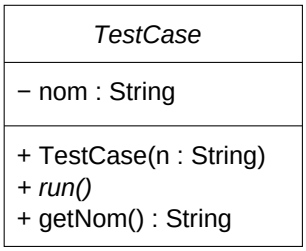

FIG. 2 – Première version de la classe `TestCase`

méthodes de la façon représentée sur la figure 3.

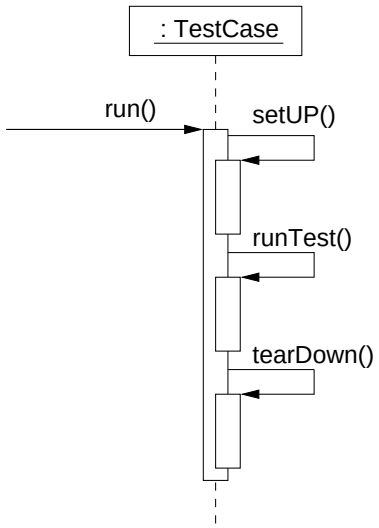

FIG. 3 – Diagramme de séquence représentant le déroulement de la méthode `run()`

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 4.

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

(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`) ?

(c) écrire le code de la classe `TestCase` (vous ne devez pas modifier la structure des classes présentées sur le diagramme 4.

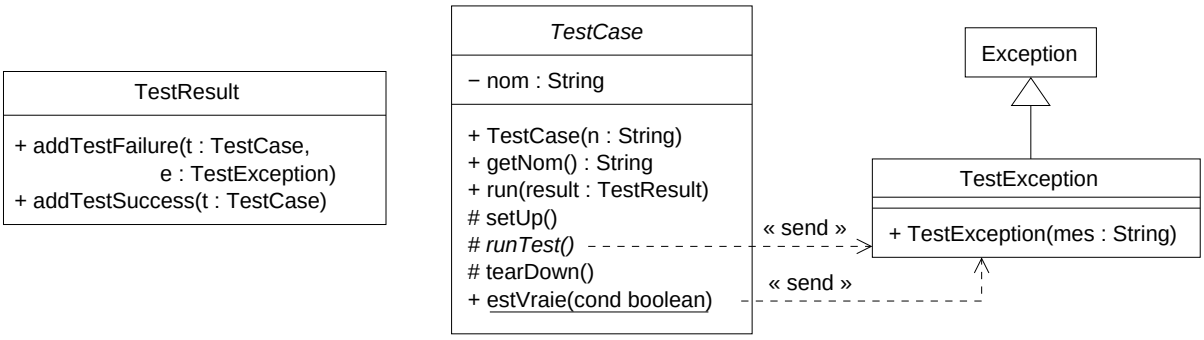

FIG. 4 – Diagramme de classe « final »

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 5.

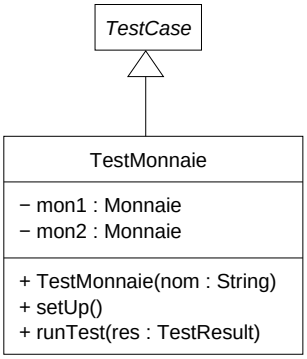

FIG. 5 – Diagramme de classe de `TestMonnaie`

Écrire la classe `TestMonnaie`.