Un cannevas pour les tests
Corrigé

Lorsque l'on développe une application (ou même une seule classe, voire un seul sous-programme), il est essentiel de définir des programmes de test. Leur objectif est d'essayer de trouver des erreurs dans l'application. Si les programmes de test sont suffisamment complets et qu'ils réussissent tous, alors on pourra avoir confiance dans l'application. Si certains échouent, il faut alors corriger l'application (ou les programmes de test si ce sont eux qui contiennent l'erreur) et rejouer l'ensemble des programmes de test (tests dits de non régression) pour vérifier que les modifications apportées n'ont pas introduit de nouvelles erreurs. Les tests sont également rejoués quand l'application évolue.

Malheureusement, il est souvent fastidieux d'écrire les programmes de test. L'objectif des exercices qui suivent est d'étudier comment est conçu et fonctionne le « framework » de test JUnit (http ://junit.org). Un « framework » est un ensemble de classes éventuellement incomplètes qui peuvent être réutilisées et adaptées.

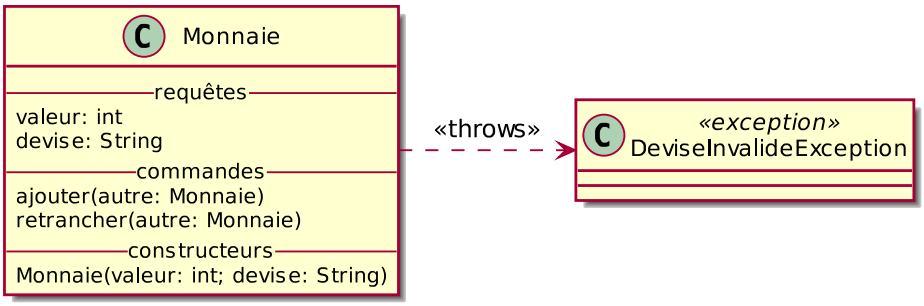
Dans un premier temps, nous définissons notre application qui se résume ici à une classe Monnaie (exercice 1). Nous écrivons ensuite quelques programmes de test « traditionnels » pour tester cette classe (exercice 2). Ceci nous permet de voir les limites de l'approche « traditionnelle » et d'introduire l'approche JUnit (exercice 3). JUnit permet de créer des tests élémentaires (exercice 4) qui peuvent être organisés en suites de tests (exercice 5).

Exercice 1 : La classe Monnaie

Une monnaie représente une certaine valeur d'argent dans une devise particulière. On considèrera que la valeur est représentée par un entier et la devise par une chaîne de caractères. Il est possible d'ajouter à la valeur d'une monnaie la valeur d'une autre monnaie. Il est également possible de retrancher à une monnaie la valeur d'une autre monnaie. Ces deux opérations n'ont de sens que si les deux monnaies ont même devise. Dans le cas contraire, elles lèvent une exception `DeviseInvalideException`.

1.1. Dessiner le diagramme de classe qui fait apparaître la classe Monnaie.

Solution :



Quelques remarques :

1. Ici, on indique une dépendance entre la classe Monnaie et DeviseInvalideException car certaines de ces méthodes peuvent lever cette exception. En toute rigueur, il faudrait dire quelles méthodes peuvent lever l'exception, ici ajouter et retrancher.
2. Dans un diagramme de classe, on ne fait généralement pas apparaître les exception. C'est un niveau de détail trop fin.
3. On a utiliser un stéréotype («exception») pour dire que DeviseInvalideException est une exception.
4. Nous avons commencé par un diagramme d'analyse. Ici, il sera naturel de choisir comme attributs valeur et devise.

1.2. Écrire en Java l'exception `DeviseInvalideException` vérifiée par le compilateur.
Solution : `DeviseInvalideException` doit être sous-type ni de `Error`, ni de `RuntimeException`. Ici, on la fait donc hériter de `Exception`.

```java
public class DeviseInvalideException extends Exception {

    public DeviseInvalideException(String message) {
        super(message);
    }

}
```

1.3. Écrire en Java la classe Monnaie.
Solution :

```java
public class Monnaie {
    private int valeur;
    private String devise;

    public Monnaie(int valeur, String devise) {
        this.valeur = valeur;
        this.devise = devise;
    }

    public int getValeur() {
        return this.valeur;
    }

    public String getDevise() {
        return this.devise;
    }

    public void ajouter(Monnaie autre) throws DeviseInvalideException {
        verifierMemesDevises(autre);
        this.valeur += autre.valeur;
    }

    public void retrancher(Monnaie autre) throws DeviseInvalideException {
```

```
24          verifierMemesDevises(autre);
25          this.valeur -= autre.valeur;
26      }
27
28      private void verifierMemesDevises(Monnaie autre) throws DeviseInvalideException {
29          if (! this.devise.equals(autre.devise)) {
30              throw new DeviseInvalideException("Devises incompatibles : "
31                      + this.devise + " et " + autre.devise);
32          }
33      }
34
35  }
```

Principaux points d'attention :

1. Les méthodes `ajouter` et `retrancher` signalent par l'exception `DeviseInvalideException` que les monnaies sont incompatibles. Comme cette exception est vérifiée par le compilateur, il faut la faire apparaître dans la signature de ces méthodes grâce à la clause **`throws`**.
2. Pour comparer les devises, il ne faut pas utliser `==` entre les chaînes de caractères (égalité physique) mais bien `equals` (égalité logique).
3. On remarque que la même vérification doit être faite dans les deux méthodes. On a donc intérêt à la factoriser dans une méthode privée à la classe (éventuellement **`protected`**) : `verifierMemesDevises`.

Exercice 2 : Test de la classe Monnaie

Il s'agit maintenant d'écrire les programmes de test de la classe Monnaie. Notons que ces programmes devraient être écrits en même temps que la classe, voire avant ! Ici nous nous limiterons à deux programmes de test.

Ces deux programmes construisent deux monnaies, m_1 qui correspond à 5 euros et m_2 à 7 euros. Le premier programme, `TestMonnaie1`, ajoute m_2 à m_1 et affiche la valeur de m_1. L'utilisateur pourra contrôler qu'elle vaut 12. Le deuxième programme, `TestMonnaie2`, retranche m_2 à m_1 et affiche la valeur de m_1 qui devrait être -2.

Écrire seulement le programme `TestMonnaie1` car `TestMonnaie2` en est très proche.

Solution :

```
1   public class TestMonnaie1 {
2
3       public static void main(String[] args) throws DeviseInvalideException {
4           Monnaie m1 = new Monnaie(5, "euro");
5           Monnaie m2 = new Monnaie(7, "euro");
6
7           m1.ajouter(m2);
8           System.out.println("valeur de m1 = " + m1.getValeur());
9       }
10
11  }
```

Nous avons choisi de laisser se propager l'exception `DeviseInvalideException` (elle est donc spécifiée dans la signature de la méthode principale) car il s'agit d'un programme de test.

Si l'exception se produit (ce qui ne devrait pas être le cas ici), le programme s'arrêtera sur une exception non récupérée et l'utilisateur constatera donc qu'il y a un problème.

Exercice 3 : Conception du framework de test

Les tests de l'exercice 2 reposent sur l'utilisateur du programme de test qui doit vérifier que le programme affiche bien les résultats attendus. Malheureusement, l'utilisateur n'est pas fiable pour détecter si un test a réussi ou non. L'idée est donc d'automatiser les tests mais aussi leur évaluation pour savoir s'ils ont réussi ou échoué.

Voici une description des aspects à prendre en compte pour automatiser les tests.

1. Un test élémentaire décrit un seul test.
   Solution : Une classe TestÉlémentaire.
2. Un test élémentaire peut être lancé.
   Solution : Une méthode `lancer` sur la classe TestÉlémentaire.
3. Ce qui se passe lorsqu'un test est lancé dépend du système à tester et du test envisagé.
   Solution : Ce paragraphe concerne la méthode `lancer` que nous avons identifiée.
   On ne peut donc pas écrire le code de la méthode `lancer` car il dépend du test considéré. Elle est donc retardée (abstraite) et la classe est abstraite.
   On peut également décider de faire une interface de `TestÉlémentaire` au lieu d'en faire une classe. C'est ce qui serait le plus naturel (car imposant le moins de contraintes). Il est toutefois trop tôt pour faire ce choix. Il faut attendre d'avoir bien compris `TestÉlémentaire`.
4. Faire un seul test n'est pas suffisant. Il faut généralement plusieurs tests pour tester une méthode. Et bien sûr, il faut tester toutes les méthodes de toutes les classes de l'application. En conséquence, ces tests élémentaires sont regroupés sous la forme de suites de tests.
   Solution : On ajoute donc une classe SuiteTest qui est constituée de plusieurs `TestÉlémentaire`. On peut utiliser une relation de composition ou d'agrégation. Ici, nous pouvons choisir agrégation en considérant qu'une même test pourrait faire partie de plusieurs suites.
5. Une suite de tests peut être lancée.
   Solution : SuiteTest possède une méthode lancer.
6. Lancer une suite de tests consiste à lancer chaque test de la suite.
   Solution : La méthode lancer de SuiteTest appelle la méthode lancer de chacun des TestÉlémentaire qui la compose. On sait donc écrire le code de cette méthode (qui n'est donc pas abstraite).

```
for (TestElementaire t : tests) {
   t.lancer();
}
```

7. Toujours dans un souci de structuration, une suite de tests peut contenir des suites de tests.

   Solution : Il faut donc pouvoir ajouter une SuiteTest dans une SuiteTest. Plusieurs solutions sont envisageables :

(a) On ajoute une relation d'agrégation réflexive sur SuiteTest. L'inconvénient, c'est que lancer une suite consistera à lancer les tests unitaires d'une part et les suites de test d'autre part. Le code de `lancer` devient plus compliqué.

(b) Pour éviter ce défaut, il faudrait pouvoir mettre la suite dans l'ensemble des tests. Il faudrait alors que SuiteTest soit un sous-type de TestElementaire. Même si techniquement (au niveau Java), ceci pourrait fonctionner, c'est conceptuellement faux car la définition de TestElementaire dit qu'il s'agit d'un *seul* test.

(c) On peut donc généraliser TestÉlémentaire et SuiteTest en Test (une interface) qui contient une unique méthode `lancer`. Une SuiteTest est alors un ensemble de Test, qui peut donc contenir soit des SuiteTest, soit des TestElementaire.

C'est une application du patron de conception **Composite** utilisé pour modéliser les structures arborescentes : les suites de tests sont les n,oeuds, les tests élémentaires les feuilles.

Dans tous les cas, il faut faire attention à ne pas créer de cycle : il ne faut pas ajouter à une suite cette suite, on l'une des suite la contenant.

8. Il est intéressant d'avoir le résultat des tests, c'est-à-dire des informations qui indiquent :
   — le nombre total de tests lancés ;
   — le nombre de tests qui ont échoué ;
   — les tests qui ont échoué ;

   Solution : Nous ajoutons une classe RésultatTest. Elle possède trois informations : le nombre de test lancés, le nombre de tests en échec et l'ensemble des tests échoués. Cette dernière information est en fait une relation vers TestÉlémentaire.

   Notons que l'information « nombre de tests en échec » de être déduite de l'ensemble des tests échoués : c'est sa taille. En UML, on le fait précéder donc d'une barre oblique.

   Puisqu'il y a un ensemble de tests échoués, il faut pouvoir ajouter un nouveau test quand il a échoué. Il y a donc une méthode `enregistrerÉchec` qui prend en paramètre un TestÉlémentaire car ce sont les tests élémentaires qui peuvent échouer, pas les suites de tests.

   Il faut pouvoir enregistrer qu'un nouveau test a été lancé (`incrémenterNbTestsLancés`).

9. Chaque test élémentaire lancé doit mettre à jour ces statistiques en fonction de son résultat.

   Solution : La méthode `lancer` a un paramètre de type `RésultatTest`. Il pourra ainsi le mettre à jour en fonction de son résultat.

   Il s'agit du patron de conception **Collecting parameter**.

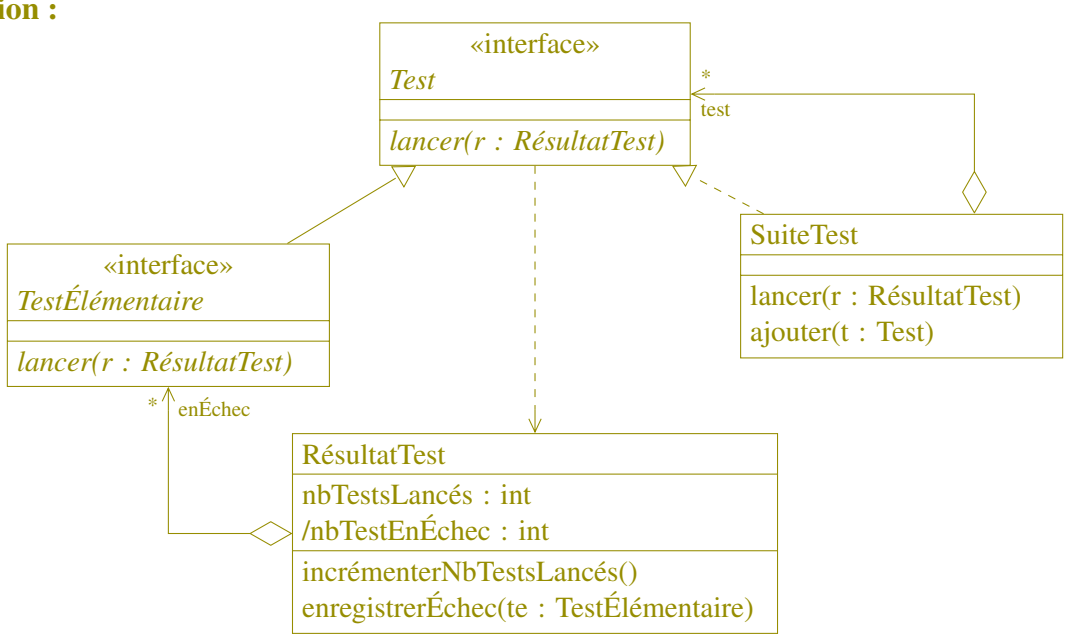
   L'initialiser au travers d'un constructeur et en faire donc un attribut ne serait pas judicieux car le même test ne pourrait pas faire partie de plusieurs suites de test. Ce serait de plus lourd d'initialiser ou de modifier tous ces objets alors que la transmission via le paramètre se fera naturellement lors de l'appel de la méthode `lancer` des tests.

3.1. Pour chaque paragraphe numéroté de la description du framework de test ci-dessus, indiquer en français, en utilisant le vocabulaire UML, ce qui peut en être déduit concernant le diagramme de classe.

Solution : Voir les annotations dans le texte ci-dessus.

3.2. Proposer un diagramme de classe détaillé qui décrit un tel framework de test.

Solution :



Exercice 4 : Le test élémentaire

Intéressons nous maintenant plus en détail à un test élémentaire. La description UML de la classe TestÉlémentaire est donnée figure 1.

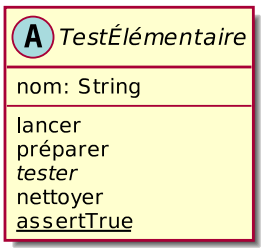


FIGURE 1 – La classe TestÉlémentaire

C'est le testeur (le programmeur qui écrit des programmes de test) qui explique ce qui doit être fait lorsque le test est lancé.

En général, le testeur voudra comparer le résultat d'un calcul à un résultat attendu. Par exemple, vérifier que le pgcd de 10 et 6 est bien 2. Pour se faire une méthode `assertTrue` est définie. Sa signature est :

```
void assertTrue(boolean expression)
```

Cette méthode vérifie que l'expression est effectivement vraie. Dans notre exemple, le testeur pourrait écrire `assertTrue(2 == pgcd(10, 6))`.

Le programmeur peut souhaiter réaliser plusieurs tests sur les mêmes données, par exemple $10 + 6 = 16$, $10 * 6 = 60$... Comme un test pourrait modifier les données, il est nécessaire de les

initialiser avant chaque nouveau test. De manière symétrique, lorsque le test est terminé, il peut être nécessaire de libérer les ressources allouées lors de l'initialisation des données. Ainsi lancer un test (`lancer`) consiste à initialiser les données (`préparer`), lancer le test effectif (`tester`) et, enfin, libérer les ressources (`nettoyer`).

Un test peut échouer pour deux raisons :

— soit parce qu'une vérification échoue (`assertTrue` sur une expression booléenne fausse). Cette erreur est qualifiée d'erreur fonctionnelle ;

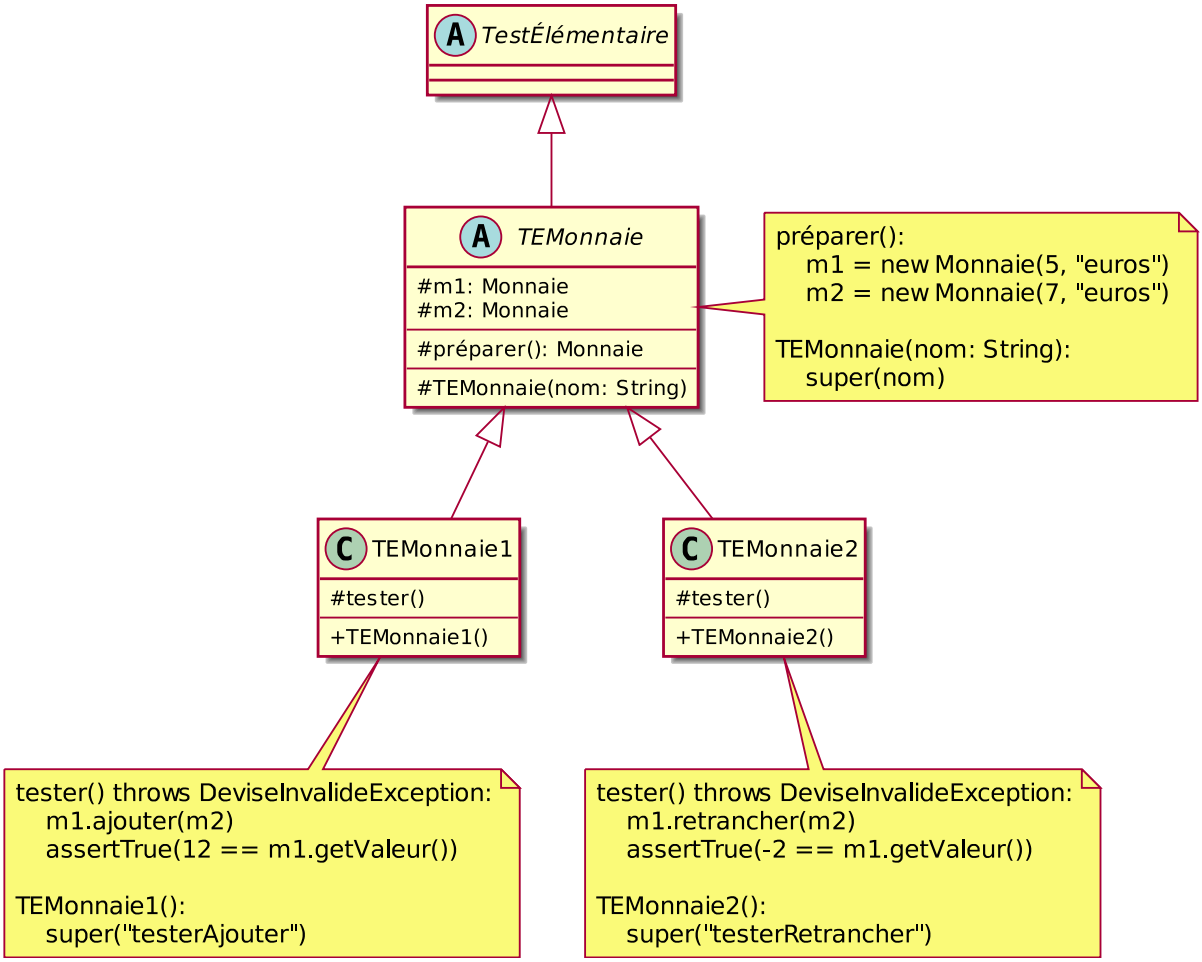
— soit parce qu'une erreur de programmation s'est produite (indice non valide pour un tableau, méthode appliquée sur une poignée nulle, division par zéro...).

Aussi, les résultats de tests doivent comptabiliser et conserver d'une part les tests en échec (erreur fonctionnelle) et les tests en erreur (erreur de programmation).

4.1. *Utilisation de la classe TestÉlémentaire.* Maintenant que la classe TestÉlémentaire existe, nous souhaitons réorganiser les deux programmes de test de la classe Monnaie (exercice 2). L'objectif est de minimiser le nombre d'instructions à écrire.

Dessiner le diagramme de classe correspondant à ces deux programmes de test et donner le code Java des méthodes dans des annotations UML.

Solution :

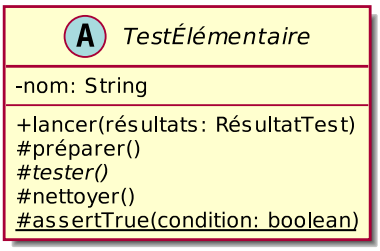


Le code à factoriser, les deux attributs et leur initialisation via la méthode `préparer`, est factorisée dans une classe `TestMonnaie`. Les classes `TestMonnaie1` et `TestMonnaie2` définissent la méthode `tester` pour réaliser le test.

4.2. *Compléter et comprendre la description UML.* Dans un premier temps, nous allons compléter la description UML de la classe `TestÉlémentaire` et comprendre les choix faits.

4.2.1. Indiquer, en le justifiant, le droit d'accès à mettre sur chaque élément de la classe.

Solution :



L'attribut doit être **`private`** : principe de l'accès uniforme et protection en écriture des attributs (pouvoir garantir la cohérence des objets de la classe).

La méthode lancer doit pouvoir être utilisée de l'extérieur de la classe, y compris depuis d'autres paquetages (par exemple elle est définie dans un paquetage d'une application particulière et utilisée par notre framework de test). Elle doit donc être **`public`**.

Les méthodes tester, préparer et nettoyer ne doivent pas être utilisées directement de l'extérieur car elles risqueraient d'être utilisées dans le mauvais ordre. Elle ne doivent donc pas être **`public`**. En revanche, elles doivent pouvoir être redéfinies dans une sous-classe. Elles doivent donc être au moins **`protected`**. En conclusion, elles sont **`protected`**.

La méthode `assertTrue` doit être au moins **`protected`** pour pouvoir être utilisée dans les sous-classes, dans le code de la méthode `tester`.

4.2.2. Seule la méthode `tester` est déclarée abstraite. Les méthodes `lancer`, `préparer` et `nettoyer` ne le sont pas. Expliquer ce qui justifie ces choix.

Solution : On sait écrire le code de la méthode `lancer` (elle appelle les trois autres) donc elle n'est pas abstraite. La méthode `tester` dépend du test à réaliser. Elle doit donc être impérativement définie par le testeur dans la sous-classe. Les deux autres devraient être définies par le testeur mais comme il n'a pas toujours la nécessité de les définir (peut-être n'a-t-il aucune données partagées), il serait fastidieux de lui imposer de définir ces deux méthodes pour chaque test. Aussi elles sont définies dans la classe `TestÉlémentaire`... avec un code vide.

On commencera par expliquer l'intérêt de définir une méthode abstraite.

Solution : Définir une méthode abstraite, c'est permettre au compilateur de vérifier que le programmeur la définira bien dans la sous-classe (ou qu'il déclarera cette classe comme abstraite). Définir une méthode abstraite, c'est donc obligé la sous-classe à la définir (ou à être elle-même abstraite).

4.2.3. Indiquer s'il est possible de créer une instance de la classe TestÉlémentaire. La réponse doit être justifiée.

Solution : La classe contient une méthode abstraite, `tester`, par conséquent la classe est abstraite. Elle ne peut donc pas être instanciée.

Seule une sous-classe concrète (et donc donnant un code à `tester`) pourra être instanciée.

4.2.4. Peut-on définir un constructeur sur cette classe ? La réponse doit être justifiée et, le cas échéant, la signature du constructeur donnée.
Solution : On peut définir un constructeur sur une classe abstraite. Ici la classe a un attribut nom qui doit donc être initialisé. On peut donc définir un constructeur qui prend en paramètre le nom du test. Il sera appelé par les constructeur des sous-classes et garantira que le nom du test sera bien initialisé.
4.3. *La méthode `assertTrue(boolean)`*. La méthode `assertTrue(boolean)` est utilisée par le testeur pour vérifier une expression booléenne. Si cette expression est fausse, la vérification a échoué et une exception `Échec` est levée. Le test sera comptabilisé en échec.
4.3.1. Expliquer pourquoi il est préférable d'utiliser une exception pour signaler l'échec plutôt qu'un code de retour de la méthode `assertTrue` ?
Solution : Si plusieurs `assertTrue` doivent être faits dans un même test élémentaire, par exemple pour vérifier que la valeur et la devise d'une monnaie ont correctement étaient initialisées, il faut faire deux `assertTrue`. Dans le cas d'un code d'erreur, ce serait au programmeur du test de gérer ces codes de retour.

Dans le cas d'un exception, l'erreur n'a pas à être traitée par le testeur et elle se propagera. Le premier `assertTrue` qui échoue invalidera le test.
4.3.2. Expliquer pourquoi la méthode `assertTrue(boolean)` est une méthode de classe.
Solution : C'est une méthode de classe car elle ne manipule pas l'objet de type TestÉlémentaire qui serait sont paramètre implicite sinon (`this`). Elle n'a donc pas à avoir de paramètre `this`.
4.3.3. Étant donné l'objectif de l'exception Échec, indiquer comment la définir en Java et écrire le code correspondant.
Solution : Échec est une exception qui a pour but de signaler que le test a échoué. Il ne faut surtout pas la définir comme vérifiée car alors dès que le programmeur d'un test utiliserait `assertTrue`, le compilateur lui demanderait de mettre un `catch` ou un `throws` alors qu'elle ne doit pas être connue du programmeur du test et doit se propager hors de la méthode `tester`.

L'exception `Échec` doit être un sous-type de `RuntimeException` ou `Error`. Ici, il ne s'agit pas d'une erreur de programmation mais bien d'une erreur non récupérable par le programmeur du test d'où le choix de la faire héritée de Error. On pourrait donc l'appeler ÉchecError (suivant les conventions Java).

```
1  public class Echec extends Error {
2      public Echec() {
3          super("condition non vérifiée");
4      }
5      public Echec(String message) {
6          super(message);
7      }
8  }
```

4.3.4. Écrire le code de la méthode `assertTrue(boolean)`.
Solution :

```
1      static public void assertTrue(boolean condition) {
2          if (! condition) {
3              throw new Echec();
```

```
4           }
5       }
```

4.4. *Programmation de la classe.* Intéressons nous d'abord à la méthode `lancer` qui est chargée de lancer le test et de le comptabiliser.

4.4.1. Comment une erreur fonctionnelle est-elle détectée ?

Solution : Une exception Échec est levée pendant l'exécution de `tester`.

Et une erreur de programmation ?

Solution : Une exception autre que Échec est levée pendant l'exécution de `tester`.

4.4.2. Écrire la méthode `lancer`.

Solution : Nous avons déjà vu qu'il fallait faire dans l'ordre des appels aux méthode préparer, tester puis nettoyer. Comme il s'agit de récupérer les exceptions Échec ou une autre exception, on les entoure d'un `try ... catch`.

Mais il faut être sûr que la méthode `nettoyer` sera appelée même si le test échoue (i.e. si une exception se produit). Comment faire ?

Si on met l'appel à `nettoyer` dans un `finally` après les `catch`, on garantit qu'elle s'exécute. Cependant, on ne maîtrise pas le code de cette méthode puisqu'elle pourra être redéfinie dans une sous-classe. Il se peut qu'elle contiennent une erreur de programmation et lève une exception. Il ne faut pas que le programme s'arrête et il faudra comptabiliser le test en erreur. Il faut donc récupérer ces exceptions ce qui ne sera pas le cas avec seul `try ... catch`. On met donc deux `try ... catch`. Et comme c'est lourd, on définit une nouvelle méthode privée qui lance le test en garantissant que nettoyer sera bien appelée.

Enfin, le code que l'on a donné pour `lancer` est le *bon* code. Il ne faut pas qu'une sous-classe redéfinissent cette méthode. On la déclare donc comme `final`.

Les sous-classes pourront adapter son comportement en redéfinissant les méthodes préparer, tester et nettoyer. Ceci correspond au patron de conception *Patron de méthode* (*Template method*)... que l'on pourrait aussi appeler « algorithme » ou « raffinage ».

```
1       /** Lancer les trois méthodes : preparer, tester puis nettoyer.  Le seul
2        * intérêt est de garantir l'exécution de nettoyer, même si une
3        * exception se produit (finally).  Les exceptions ne sont pas récupérées
4        * ici mais propagées.
5        */
6       private void lancerSansControle() throws Throwable {
7           try {
8               this.preparer();
9               this.tester();
10          } finally {
11              this.nettoyer();
12          }
13      }
14
15      final public void lancer(ResultatTest resultats) {
16          throw new RuntimeException("Implantation de lancer(ResultatTest) "
17                  + " non fournie !");
18          try {
19              resultats.incrementerTest();
```

```
            this.lancerSansControle();
        } catch (Echec e) {
            resultats.ajouterEchec(this, e);
        } catch (Throwable e) {
            resultats.ajouterErreur(this, e);
        }
    }
```

4.4.3. Écrire la classe TestÉlémentaire. On se contentera de mettre des ellipses (...) pour le code des méthodes qui ont déjà été définies.
Solution :

```
abstract public class TestElementaire implements Test {

    private String nom;

    public TestElementaire(String sonNom) {
        this.nom = sonNom;
    }

    final public void lancer() {
        ResultatTest resultats = new ResultatTest();
        this.lancer(resultats);
        System.out.println(resultats);
    }

    /** Lancer les trois méthodes : preparer, tester puis nettoyer.  Le seul
     * intérêt est de garantir l'exécution de nettoyer, même si une
     * exception se produit (finally).  Les exceptions ne sont pas récupérées
     * ici mais propagées.
     */
    private void lancerSansControle() throws Throwable {
        try {
            this.preparer();
            this.tester();
        } finally {
            this.nettoyer();
        }
    }

    final public void lancer(ResultatTest resultats) {
        try {
            resultats.incrementerTest();
            this.lancerSansControle();
        } catch (Echec e) {
            resultats.ajouterEchec(this, e);
        } catch (Throwable e) {
            resultats.ajouterErreur(this, e);
        }
    }

    protected void preparer() throws Throwable {
    }
```

```
42
43      abstract protected void tester() throws Throwable;
44
45      protected void nettoyer() throws Throwable {
46      }
47
48      static public void assertTrue(boolean condition) {
49          if (! condition) {
50              throw new Echec();
51          }
52      }
53
54  }
```

Quelques remarques supplémentaires sur cette classe et ses méthodes.

1. Les méthodes préparer, tester et nettoyer sont spécifiées comme pouvant lever n'importe quelle exception (`throws Throwable`). Ceci est nécessaire car le code de ces méthodes peut lever (directement ou indirectement) des exceptions qui sont vérifiées par le compilateur (par exemple quand on utilise la méthode `ajouter` de Monnaie). Il ne faudrait pas que ce soit interdit par le compilateur car la méthode redéfinie ne l'aurait pas prévu en spécifiant que l'exception pouvait se propager.
2. On a définit une méthode `lancer` sans paramètre : elle crée un objet `RésultatTest` et appelle l'autre méthode `lancer`.
3. Puisqu'on distingue échec d'un test (erreur fonctionnelle) et erreur d'un test (erreur de programmation), il faut comptabiliser ces deux types de résultats. On double donc la relation entre RésultatTest et TestÉlémentaire et on définit deux méthodes spécifiques : enregistrerÉchec et enregistrerErreur.
4. Dans ces deux méthodes, on enregistre aussi pourquoi le test est en échec ou erreur. Ceci est donné par l'exception qui s'est produite. On aura ainsi l'exception, son message et la trace des appels.

Exercice 5 : Suite de tests

Intéressons nous maintenant à la notion de suite de tests.

5.1. Indiquer si la conception détaillée de la classe `TestÉlémentaire` (exercice 4) nécessite de modifier le diagramme de classe du framework de test (exercice 3). Donner les éventuelles modifications.

Solution : L'exercice précédent a montré que TestÉlémentaire était une classe abstraite et non une interface. C'est la seule modification à apporter au diagramme précédent.

Pour la classe RésultatTest, il faut distinguer les deux raisons qui font qu'un test échoue : échec (erreur fonctionnelle) et erreur (erreur de programmation). Il faut conserver deux ensembles de tests. La relation entre RésultatTest et TestÉlémentaire est donc doublée.

5.2. Une suite de tests est un ensemble de tests. Expliquer les différentes possibilités en Java de représenter cette relation avec les avantages et inconvénients associés.

Solution : Trois solutions principales :

- prendre un tableau de base `Test [] tests`. L'inconvénient est que le tableau ne peut pas être redimmensionné et cette opération est donc à la charge du programmeur. L'avantage est que le compilateur peut contrôler les opérations faites sur ce tableau (contrôle de type). On ne peut y mettre que des Test.
- prendre une liste (`List`), par exemple `ArrayList`, ou une autre structure de données (Collection). En Java 1.4, les collections sont des collections d'`Object`. `List` est redimmensionnable (avantage) mais le contrôle de type est perdu.
- depuis Java 1.5, la généricité permet de préciser le type des éléments d'une `List`. On a donc un tableau redimmensionnable et le contrôle de type. **C'est donc la solution à privilégier !**

Conclusion : Il est important de construire des jeux de test pour vos classes. JUnit peut vous aider en apportant encore plus de facilités que ce qui a été présenté dans ce sujet...

La devise d'un développeur pourrait alors être : « Si une fonctionalité d'un programme n'a pas de tests automatisés, nous considérons qu'elle ne fonctionne pas ».