MIF01 - Gestion de projet et génie logiciel

—

Contrôle continu final

17 août 2020

Merci de rendre chaque partie (I, II, III) sur des copies différentes. Rendez impérativement 3 copies (éventuellement blanche, si vous n'avez pas traité une partie).

Documents autorisés : 5 feuilles A4 (recto-verso).

Afin d'obtenir tous les points il vous est demandé de justifier vos résultats. Le barème proposé est susceptible d'être modifié lors de la correction. Il n'est présent que pour vous donner une idée du poids relatif des différentes questions.

I Questions de Cours (6 points)

Q.I.1) - En Scrum, quels sont les objectifs de la réunion de planification de sprint ? Citez 3 éléments concrets produits pendant cette réunion. (2 points)

— Backlog de sprint : liste de stories
— Liste de tâches techniques
— Évaluation des stories et/ou des tâches techniques
— Burndown initialisé avec 1 point.

Q.I.2) - Quelle est la différence entre une « user-story » (au sens Scrum ou XP) et un cas d'utilisation (au sens d'Alistair Cockburn) ? (1 point)

Une user-story est très concise, ne détaille pas les étapes individuelles. Un cas d'utilisation est aussi plus général (par exemple un cas d'utilisation avec extensions correspond en général à plusieurs user-stories).

Q.I.3) - Quelle est la différence entre un test fonctionnel et un test structurel ? Quels sont les intérêts de ces deux types de tests ? Si possible, appuyez vos explications sur des exemples. (2 points)

Test structurel : on s'intéresse à l'implémentation et on cherche à tester les cas particuliers liés à la structure du code. Intérêt : trouver les bugs liés à l'implémentation.

Test fonctionnel : on s'intéresse à la fonctionnalité, donc à la spécification du logiciel, et on teste les cas prévus par la spécification.

Par exemple, si le code à tester est un tableau associatif (« map »), un test fonctionnel va tester l'ajout d'un élément, la recherche d'un élément, l'ajout d'un élément déjà présent. Un test structurel pour une implémentation à base de table de hashage devra également tester les cas de collision de la fonction de hashage (la fonction ne faisant pas partie de la spécification c'est impossible en test fonctionnel).

II Design-Pattern et Testabilité (10 points)

Le but de cet exercice est d'écrire une classe « chronomètre ». L'interface à respecter est la suivante :

```
1  public interface StopWatch {
2      void start();
3      void stop();
4      /**
5       * Measure the time between a call to start() and a call to stop()
6       * (in seconds)
7       */
8      double value();
9  }
```

Un exemple d'utilisation est :

```
1  StopWatch w = ...;
2  w.start();
3  // ...
4  w.stop();
5  double timeBetweenStartAndStop = w.value();
```

Une première implémentation est la suivante (`System.nanoTime()` renvoie le temps écoulé depuis le démarrage de la JVM, en nanosecondes) :

```
1  public class StopWatchHardCoded implements StopWatch {
2      private long startTime;
3      private long stopTime;
4
5      public void start() {
6          startTime = System.nanoTime();
7      }
8
9      public void stop() {
10         stopTime = System.nanoTime();
11
12     }
13
14     public double value() {
15         return (stopTime - startTime) / 1e9;  // nanoseconds to seconds
16     }
17
18 }
```

Pour les contraintes de l'exercice, on se limite à une logique très simple (soustraction de dates et conversion de nanosecondes en secondes) qui peut rendre l'exercice un peu artificiel. L'exercice serait plus parlant, mais plus laborieux sur papier, sur une classe plus évoluée (par exemple un chronomètre avec une fonction « pause »).

On rappelle ici les 5 principes SOLID (`https://fr.wikipedia.org/wiki/SOLID_(informatique)`) :

— Responsabilité unique *(**S**ingle responsibility principle)*
une classe, une fonction ou une méthode doit avoir une et une seule responsabilité
— Ouvert/fermé *(**O**pen/closed principle)*
une classe doit être ouverte à l'extension, mais fermée à la modification
— Substitution de Liskov *(**L**iskov substitution principle)*
une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme
— Ségrégation des interfaces *(**I**nterface segregation principle)*
préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale
— Inversion des dépendances *(**D**ependency inversion principle)*
il faut dépendre des abstractions, pas des implémentations

Q.II.1) - L'un des principes SOLID est violé par `StopWatchHardCoded`, lequel ? Pourquoi ? (1 point)

Le D de SOLID, « Dependency inversion principle », qui demande de dépendre d'interfaces mais pas d'implantation particulière (`System.nanoTime()`). Une autre manière de dire plus ou moins la même chose est le « low coupling » de GRASP que nous ne respectons pas en écrivant en dur une dépendance sur `System.nanoTime()`.

Un test unitaire pour tester cette classe peut être écrit comme :

```
1  @Test
2  public void testHardCoded() throws InterruptedException {
3      StopWatch w = new StopWatchHardCoded();
4      w.start();
5      TimeUnit.SECONDS.sleep(100);
6      w.stop();
7
8      assertEquals(100.0, w.value(), 0.1);
9  }
```

`TimeUnit.SECONDS.sleep(100)` fait une attente de 100 secondes. La dernière ligne (`assertEquals`) vérifie que `w.value()` est comprise entre 99.9 et 100.1.

Q.II.2) - Donnez deux problèmes potentiels avec ce test. (2 points)

— Il met 100 secondes à s'exécuter, c'est long !
— On ne peut pas tester la valeur exacte renvoyée par `w.value()` car cette valeur dépend de trop de choses (OS, JVM, charge du système, ...) et sera toujours « légèrement supérieure à 1.0 ».

Q.II.3) - Citez un test qui aurait été pertinent sur cette classe mais qu'on ne pourra pas réaliser en suivant le même principe que `testHardCoded()`. (1 point)

> Les tests d'attente longue, pour vérifier qu'il n'y a pas de débordement arithmétiques sur les grandes valeurs de retour de `w.value()` : les tests prendraient des années !

Pour améliorer la situation et permettre un vrai test unitaire de la classe, un développeur propose d'utiliser le pattern Indirection pour réaliser une inversion de dépendance entre `StopWatch` et `System.nanoTime()` pour éliminer le couplage entre ces deux classes [1]. Il utilise l'interface suivante :

```
1  public interface TimeProvider {
2      long getNanoSeconds();
3  }
```

Il crée donc une deuxième implémentation de `StopWatch` qu'il appelle `StopWatchTimeProvider`. Il écrit alors le test suivant qui est équivalent à `testHardCoded` (et qui pose donc les mêmes problèmes que l'ancienne version) et passe également :

```
1      @Test
2      public void testTimeProvider() throws InterruptedException {
3          StopWatch w = new StopWatchTimeProvider(
4              new SystemTimeProvider());
5          w.start();
6          TimeUnit.SECONDS.sleep(100);
7          w.stop();
8
9          assertEquals(100.0, w.value(), 0.1);
10     }
```

Q.II.4) - Dessinez le diagramme UML des classes `StopWatchTimeProvider`, `SystemTimeProvider` et des interfaces utilisées par ces classes. Il n'est pas nécessaire de mentionner les champs, mais seulement les noms de classes et de méthodes. (1 point)

Q.II.5) - Écrivez le code Java correspondant (classes et interfaces) (2 points)

```
1   public class SystemTimeProvider implements TimeProvider {
2       public long getNanoSeconds() {
3           return System.nanoTime();
4       }
5   }
6
7   public class StopWatchTimeProvider implements StopWatch {
8       private TimeProvider timeProvider;
9       private long startTime;
10      private long stopTime;
11
12      public StopWatchTimeProvider(TimeProvider tp) {
13          timeProvider = tp;
14      }
15
16      public void start() {
```

1. Une généralisation de cette approche est l'injection de dépendance

```
17              startTime = timeProvider.getNanoSeconds();
18          }
19
20          public void stop() {
21              stopTime = timeProvider.getNanoSeconds();
22          }
23
24          public double value() {
25              return (stopTime - startTime) / 1e9;
26          }
27
28      }
```

Q.II.6) - En quoi l'inversion de dépendance a-t-elle résolu notre problème de test initial ? Proposez un test unitaire de `StopWatchTimeProvider` qui n'utilise pas, même indirectement, `System.nanoTime()`. Une solution est d'écrire une classe `MockTimeProvider` qui implémente `TimeProvider`, mais dont la méthode `getNanoSeconds()` renvoie des valeurs prises dans une liste pré-déterminée (par exemple passée au constructeur de `MockTimeProvider`) au lieu d'appeler `System.nanoTime()`. (2 points)

```
1       public class MockTimeProvider implements TimeProvider {
2           private Queue<Long> values = new LinkedList<Long>();
3
4           public void addValue(long i) {
5               values.add(i);
6
7           }
8
9           public long getNanoSeconds() {
10              return values.poll();
11          }
12
13      }
14
15      // ...
16
17          @Test
18          public void testTimeProviderMocked() {
19              MockTimeProvider p = new MockTimeProvider();
20              p.addValue(1000000000001);
21              p.addValue(2000000000001);
22              StopWatch w = new StopWatchTimeProvider(p);
23              w.start();
24              w.stop();
25
26              assertEquals(100.0, w.value(), 0.0);
27          }
```

(Ou mieux, utiliser un framework de mock)

Q.II.7) - Mis à part la testabilité, notre code a-t-il été rendu plus générique ? Pourquoi ? (1 point)

Oui, il est plus générique. Si par exemple on ne fait pas confiance à l'horloge locale de la machine on peut aller chercher l'heure ailleurs, il suffira d'implémenter l'interface `TimeProvider`.

III Cas d'utilisation (5 points)

Halloween approche, et vous, vampire, détestez cette fête. Plutôt que de devoir supporter le contact avec les enfants, vous décidez de construire une maison hantée automatisée, contrôlée informatiquement.

Votre maison doit satisfaire aux exigences suivantes :

- Les enfants peuvent sonner à la porte, et quand ils le font ils sont accueillis par un automate (robot d'apparence humaine).
- La maison contient une réserve de bonbons, que vous pouvez recharger par l'arrière de la maison.
- Pour garder un effet de surprise, il arrive qu'un monstre sorte de la maison pour effrayer les enfants.
- Comme vous n'êtes pas si méchant que ça, la plupart du temps vous donnez des bonbons aux enfants qui en demandent (« trick or treat ? », ou « des bonbons ou un sort ? »)
- La porte ne doit pas être ouverte quand aucun enfant n'est devant la maison.
- La porte et l'automate sont contrôlés par des moteurs, eux-mêmes contrôlés par informatique.
- Si des enfants reviennent avec trop d'insistance, le système vous alerte par SMS pour que vous veniez régler la situation (et possiblement satisfaire votre soif de sang par effet de bord).

Proposez deux cas d'utilisation (au sens A. Cockburn) :

- L'un de niveau stratégique, avec vous (le vampire) comme acteur principal. Vous pouvez utiliser le format simplifié (peu d'en-têtes) pour ce cas et vous pourrez omettre les extensions.
- L'autre de niveau utilisateur, avec un groupe d'enfants comme acteur principal. Utilisez un format étoffé pour celui-ci.

III.1 Cas 1

Barème : 1 point pour les en-têtes, 1 point pour les étapes.

Nom Une fête d'Halloween

Acteur principal Vampire

Porté Maison

Niveau Stratégique

Sénario Nominal 1. Le vampire rempli la réserve de bonbons

2. Le vampire active le mode automatique sur la maison
3. La maison accueille les groupes d'enfants qui se présentent

III.2 Cas 2

Barème : 1 point pour les en-têtes (pas forcément tous, mais au moins 7 en-têtes), 1 point pour les étapes, 1 point pour les extensions (au moins 2).

Nom Un groupe d'enfants est accueilli par la maison

Acteur Principal Le groupe d'enfants

Porté Maison

Niveau Utilisateur

Intervenants et intérêts Le vampire, uniquement dans les cas où sa présence est requise (cf. extensions)

Pré-condition La porte est fermée

Garanties minimales Les enfants sont accueillis

Garanties en cas de succès Les enfants récupèrent des bonbons

Déclencheur Les enfants sonnent à la porte

Scénario nominal 1. La porte s'ouvre
2. Un automate dit bonjour aux enfants
3. Les enfants crient « trick or treat » (« Des bonbons ou un sort ! »)
4. L'automate distribue des bonbons aux enfants
5. Les enfants repartent
6. La porte se ferme

Extensions

2a Si les enfants sont déjà venus plus de 2 fois, le système prévient le vampire par SMS.

4a Le système fait un tirage aléatoire et avec une probabilité 1/10, fait sortir un monstre de la maison.

4b S'il n'y a plus de bonbons, le système fait sortir un monstre de la maison même si la condition 4a n'est pas satisfaite.