Développement dirigé par les modèles

Travaux Pratiques
Tests unitaires, Observateur, Décorateur,

Année Académique 2024-2025

Enseignant: Stéphane **Dupont***
Assistant: Guillaume **Cardoen**†

Département d'Informatique
Faculté des Sciences
Université de Mons

27 novembre 2024

*Stephane.DUPONT@umons.ac.be
†Guillaume.CARDOEN@umons.ac.be

Table des matières

1 Introduction **3**
Rappel théorique : Introduction . . . 3

2 Les bases **5**
2.1 Tests unitaires . . . 5
Exercice 2.1 : Le grand maître des tests (∗) . . . 5
Rappel théorique : Les tests unitaires . . . 5
Exercice 2.2 : Un tirage au sort contrôllé (∗) . . . 7
Rappel théorique : Le Mocking . . . 8
Exercice 2.3 : Des chiots anormaux (∗) . . . 9
Rappel théorique : Pairwise testing . . . 9
Exercice 2.4 : Fibonacci . . . 11
Exercice 2.5 : Les équations du second degré . . . 11
2.2 Décorateur . . . 11
Exercice 2.6 : Le grand maître de la décoration (∗) . . . 11
Rappel théorique : Le design pattern décorateur . . . 12
Exercice 2.7 : Un raspberry PI gaming ? . . . 13
2.3 Observateur . . . 13
Exercice 2.8 : Le grand maître de l'observation (∗) . . . 13
Rappel théorique : Le design pattern observateur . . . 14
Exercice 2.9 : Un progrès observé (∗) . . . 15
Exercice 2.10 : Ils sont toujours aussi mignons ! . . . 15
Exercice 2.11 : Un joueur qui se déplace . . . 16

3 Gradle **17**
3.1 Introduction à Gradle . . . 17
3.2 Installer Gradle . . . 17
3.3 Création d'un projet Gradle . . . 17
3.4 Utiliser Gradle . . . 18
3.4.1 En ligne de commande . . . 18
3.5 Rapport de testing . . . 18
3.6 Ajouter Mockito à Gradle . . . 18
3.7 Le wrapper : Gradle sans installation . . . 18
3.7.1 Générer le wrapper . . . 18
3.7.2 Utiliser le wrapper . . . 19
3.7.3 Erreur : permission non-accordée . . . 19

1 Introduction

Bienvenue dans les TPs de DDPM. Ces TPs ont été intégralement refait et repensé pour l'année académique 2024-2025 suite aux commentaires des étudiants de l'année 2023-2024 [1]. Bien que les premiers énoncés se passent souvent dans un monde fictif, la matière de ces TPs restent tout ce qu'il y a de plus réelle et sérieux. Celle-ci est fortement utilisé en entreprise et vous permet de gagner beaucoup de temps lors d'une implémentation ou de la conception d'un programme (et encore plus lors de la maintenance de celui-ci). De plus, les design pattern sont généralement beaucoup utilisé par les informaticiens. Vous en verrez souvent dans du code ou des projets réels (même s'ils sont parfois cachés). Il est donc important de les comprendre et de les maîtriser.

Les TPs sont divisés en deux grandes parties :

1. D'abord, des exercices basiques sur chaque design pattern vous sont fournis. Le but de ces exercices est de vous entraîner à implémenter un design pattern en vous concentrant sur celui-ci. Bien que, généralement, ces exercices ne représentent pas un challenge au niveau algorithmique, ils sont un bon point d'entrée dans le monde des design patterns. Ces exercices sont généralement écrit de telle sorte à vous donner une analogie et une intuition sur le design pattern. Ils ne représentent que rarement une situation réaliste, bien que certains ont le but de vous montrer l'utilisation d'un design pattern en pratique.
2. Ensuite, nous vous proposons des exercices combinant plusieurs design patterns, ou qui sont tirés d'examens précédents. Ces exercices vous donnent l'occasion d'approfondir vos connaissances tout en vous entraînant. Ils sont algorithmiquement plus complexes et la compréhension de leur résolution vous oblige à comprendre les design patterns concernés. Ne commencez donc pas par ces exercices si les design patterns ne sont pas encore clairs pour vous. Au contraire des exercices basiques, ces exercices représentent des situations réalistes auxquelles vous serez peut-être confronter dans le futur.

Plusieurs symboles sont utilisés dans les énoncés afin de simplifier la navigation et le choix des exercices :

- 🟢 : simple ;
- 🟠 : légèrement plus complexe ;
- 🔴 : complexe, mais faisable si vous avez compris ;
- ⚫ : bonne chance, amusez-vous bien ! ;
- 🟣 : l'exercice est reprit d'un examen ou d'un devoir précédent ;
- (∗) : l'exercice est supposé être fait en TP ;

Rappel théorique : Introduction

Cette boîte vous donne des « bref »rappels théoriques. Les rappels théoriques sont souvent présents dans les exercices basiques afin de vous rappeler comment fonctionne le design pattern.

Le saviez-vous?

Cette boîte représente un « Le saviez-vous ? ». Elle vous donne des informations complémentaires, parfois qui n'ont rien à voir avec l'exercice, mais qui sont intéressantes. Rassurez-vous, ce sont des informations que vous ne devez ni étudier ni connaître pour

1. Oui oui, on vous écoute.

l'examen. Le but est soit de compléter votre culture, soit de vous aider dans l'implémentation de l'exercice via des astuces pas forcément connues de tout le monde.
Par exemple, les énoncés se passent dans un monde fantastique dans le but de vous donner des intuitions. Certains exercices ont l'unique but de vous donner une analogie, tandis que d'autres vous donnent une idée de comment ces designs patterns peuvent être utilisés en pratique.

Si un exercice n'est pas clair pour vous, n'hésitez pas à essayer un autre exercice concernant le même design pattern d'une difficulté semblable et d'y revenir plus tard. Rappelez-vous que vous êtes libre de faire les exercices dans l'ordre que vous le souhaitez. Pour des raisons évidentes, nous vous conseillons fortement de suivre le même ordre que les sessions théoriques afin d'avoir une première idée des design pattern concernés.

Les énoncés sont volontairement longs avec beaucoup de texte et de détails superflus. Un but auxiliaire de ces TPs est de vous entraîner à repérer les points importants dans des énoncés et à vous concentrer sur ceux-ci. De plus, pour les exercices basiques, ce superflu est souvent construit de tel sorte à vous donner une intuition ou une analogie sur le fonctionnement du design pattern. N'hésitez donc pas à rechercher le design pattern dans l'énoncé avant de foncer tête baissée dans l'implémentation.

De manière similaire, les énoncés manquent souvent de détails techniques (i.e., nous ne vous donnons pas les classes à implémenter la plupart du temps). Vous êtes libre d'implémenter les choses comme vous le souhaitez. Le but derrière ceci est de non seulement laisser place à votre créativité, mais aussi de vous entraîner à développer un logiciel en vous basant sur des exigences imprécises. Cependant, vous êtes libre de demander à l'assistant du cours des précisions si un exercice vous semble trop imprécis pour être fait. Pour rappel, ces énoncés sont nouveaux pour la plupart. Il seront adaptés selon vos commentaires et complétés selon vos incompréhensions.

2 Les bases

Ce chapitre vous donne des exercices basiques sur les design patterns afin de vous familiariser avec ceux-ci.

2.1 Tests unitaires

Pour chacun des exercices ci-dessous, on attend de vous que vous créez un projet Gradle. Vous devez aussi y ajouter les dépendances nécessaires (e.g., JUnit et Mockito si Gradle ne l'a pas déjà fait). Vous pouvez vous référer à la Section 3 expliquant Gradle dans ce but. Si, pour une raison ou une autre, vous avez des problèmes avec Gradle, un projet vide Gradle est sur Moodle. Comme celui-ci utilise le wrapper Gradle (cf. le document expliquant Gradle), il devrait fonctionner quelque soit l'ordinateur, si une version récente de Java est installée.

Le saviez-vous?

Gradle est ici utilisé dans le but de gérer vos dépendances pour vous. La gestion des dépendances Java est une source commune d'erreur difficile à comprendre. Gradle simplifie un peu votre vie en gérant ça pour vous. Il sera utilisé plus tard dans votre cursus. C'est donc aussi une bonne occasion de vous familiariser un peu avec lui.
Bien que ce n'est pas obligatoire, vous pouvez continuer de l'utiliser dans les exercices ne nécessitant pas de dépendances (i.e., tous sauf les tests unitaires), mais nous ne vous obligeons à rien pour ces exercices là.

Exercice 2.1 : Le grand maître des tests (∗)

Sur le chemin menant vers la domination de la programmation en Java, vous vous trouvez confronté à une question : comment est-ce qu'on peut s'assurer que nous n'introduisons pas de bug en faisant un changement ? Un drôle de personnage répond à votre pensée : « Facile, les tests unitaires avec JUnit. »D'où vient-il et comment a-t-il su la question à laquelle vous pensiez ? Cela reste encore un mystère à ce jour. Curieux de l'utilisation et l'utilité des tests unitaires, vous lui demandez de vous apprendre.

Le mystérieux personnage se présente comme « Le grand maître des tests unitaires ». « Il doit sûrement rire en disant ça, pas vrai ? »pensez-vous. Il accepte de vous expliquer. Cependant, vous devez lui prouver votre intelligence et votre motivation avant. Le grand maître vous propose d'implémenter en Java une calculatrice comprenant six expressions arithmétiques basiques (addition, soustraction, multiplication, division, modulo et exponentiation). Il précise que vous pouvez en rajouter d'autres si vous le souhaitez. « Une calculatrice ? J'ai une impression de déjà-vu »continuez-vous de penser.

Vous ne comprenez pas en quoi une calculatrice lui prouve quoi que ce soit, mais comme il a l'air convaincu de sa proposition, vous décidez de le faire quand même.

(a) Implémentez une classe Calculator qui implémente les expressions arithmétiques basiques.

Le grand maître, maintenant convaincu de votre motivation et de vos capacités, décide de vous donner un peu de sa sagesse.

Rappel théorique : Les tests unitaires

Pour rappel, le principal but des tests unitaires est de pouvoir tester automatiquement que chaque partie, chaque algorithme, chaque « unité »de votre programme a le comportement attendu. Pour cela, les tests unitaires se basent souvent sur des assertions (i.e., des affirmations qui doivent être vraies à ce moment-là). Le test réussi seulement si toutes les assertions du test sont vraies. Au contraire, si une assertion est fausse, nous parlons alors d'un test raté (représentant alors un bug dans votre programme). Par contre, si une exception innattendue se produit durant le test, c'est alors un test en erreur (i.e., il n'a pas réussi, mais à cause d'une exception innatendue et non d'une assertion fausse).
Afin de tester une « unité », il faut d'abord se demander quel est son but et ce que vous devez tester. Une fois que vous êtes sûr des buts, retours attendus et erreurs possibles de votre « unité », faites en sorte de tester un maximum de cas. Généralement, une même « unité »peut-être testé via plusieurs tests, chacun représentant un contexte différent. Il est important qu'en plus des cas normaux, vous testiez aussi les cas d'erreur. Souvenez-vous des principes de la programmation défensive : l'utilisateur fera tout ce qu'il peut pour détruire votre programme. Cela fait partie de votre travail de l'en empêcher. Les tests vous permettent de tester automatiquement que vos algorithmes fonctionnent comme vous vous y attendez, mais aussi que vos défenses contre les « méchants utilisateurs »sont bien en place et fonctionnelles.
Plusieurs techniques existent afin de découvrir les tests à faire. Si vous séparez les différentes entrées possibles de votre unité en classes d'équivalences (i.e., pour toutes les valeurs d'une même classe, l'algorithme est supposé agir de la même manière), vous faites généralement un test pour chaque classe avec des valeurs que vous choisissez. Si vous avez plusieurs arguments, il est préférable de tester toutes les combinaisons de ces arguments avec des classes d'équivalence. Dans le cas où vous avez une explosion combinatoire (i.e., nettement trop de tests à faire si on considère toutes les combinaisons), utilisez votre bon sens afin de sélectionner ceux intéressants. Généralement, nous sélectionnons les tests pour chaque entrée individuellement (chaque classe d'équivalence) et les différentes combinaisons qui sont susceptibles de changer le comportement de l'algorithme. Dans certains cas, un choix aléatoire de quelques combinaisons peut aussi s'avérer pratique.
JUnit est un des frameworks de test les plus connus et les plus utilisés (si pas le plus connu et le plus utilisé) pour faire des tests unitaires en Java. Le framework de test vous propose plusieurs assertions [a] telles que :

- assertEquals(Object expected, Object actual) affirmant que actual doit être égal à expected. En cas de nombre flottants, un troisième argument delta peut-être donné afin de contrer l'erreur d'approximation des nombres flottants.
- assertNotEquals(Object expected, Object actual) affirme que actual doit être différent de expected.
- assertArrayEquals(Object[] expected, Object[] actual) affirmant que le tableau actual doit être égal au tableau expected.
- assertIterableEquals(Iterable<?> expected, Iterable<?> actual) affirmant que les valeurs de l'itérable actual doit être égal aux valeurs de l'itérable expected (l'ordre doit être respecté aussi).
- assertTrue(boolean condition) et assertFalse(boolean condition) affirment que le booléen condition est respectivement vrai ou faux.
- assertThrows(Class<T> expectedType, Executable executable) affirme que, lors de l'exécution de executable, l'exception de type expectedType est lancée.

— assertSame(Object expected, Object actual) affirme expected et actual font référence au même objet (i.e., les deux objets ont la même référence mémoire).

— assertTimeoutPreemptively(Duration timeout, Executable executable) affirme que l'exécution de executable ne doit pas prendre plus de temps que la durée timeout. Si le temps est dépassé, l'exécution s'arrête.

Faites attention à ne pas inverser les arguments actual et expected. actual est la valeur que votre algorithme vous donne, expected est la valeur que l'algorithme est supposé vous donner (i.e., la valeur à laquelle vous vous attendez). L'inversion de ces valeurs introduit des erreurs dans le rapport de test, introduisant donc de la confusion quant à pourquoi le test rate.

Ces différents tests se placent dans des méthodes annotées de @Test. De manière générale, ces méthodes se trouvent dans des classes situées en dehors du code source principal de votre application (dans un dossier test par exemple). Un exemple d'un tel test vous est donné au Listing 1.

Listing 1 – Exemple d'un test unitaire

```
1  public class SqrtTest {
2
3      @Test
4      public void testSqrt() {
5          assertEquals(1.0, Math.sqrt(1.0), 1e-8);
6          assertEquals(4.0, Math.sqrt(2.0), 1e-8);
7          assertEquals(9.0, Math.sqrt(3.0), 1e-8);
8      }
9
10 }
```

a. Voir `https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html` pour la liste complète des assertions proposées

« Te voilà maintenant prêt ! »dit le maître.

(b) Commencez par énumérer (en français) les tests unitaires utiles que vous voyez pour votre calculatrice. Par exemple, l'exponentiation avec un exposant 0 doit toujours valoir 1 est un cas de test valide. N'oubliez pas d'indiquer aussi les tests d'erreurs (e.g., division par 0). Si vous pensez avoir déjà bien compris et souhaitez gagner du temps, vous pouvez aussi directement lire la question suivante et sauter cette étape.

(c) Implémentez les différents tests que vous avez trouvé au point précédent.

Exercice 2.2 : Un tirage au sort contrôllé (∗) 

Récemment, vous avez joué au lotto. Sans aucun étonnement de votre part, vous avez perdu et étiez un peu déçu. Vous avez déjà ressenti cette déception de nombreuse fois et vous demandez ce que vous ressentiriez si vous gagniez un jour au lotto.

Une petite voix dans votre tête vous annonce alors que vous pouvez imaginer avoir gagné au lotto. Vous aimez l'idée, mais ça ne vous fera pas le même effet que si on vous annonce avoir trouvé la bonne combinaison.

Une autre petite voix dans votre tête vous propose de simuler le lotto jusqu'au moment

où vous gagnez. Vous devez quand même deviner la combinaison et toutes les essayer jusqu'au moment où vous gagnez, ce qui va vous prendre beaucoup trop de temps.

Une troisième petite voix dans votre tête vous propose donc d'enlever la partie aléatoire du lotto. Vous ne comprenez pas tout de suite comment faire. Par définition, il y a de l'aléatoire dans un tirage au sort, c'est tout le but. La petite voie fait résonner « Mocking »dans votre tête.

Désespéré de vouloir gagner au lotto et voulant apprendre quelque chose, vous vous renseignez sur ce qu'est le mocking.

Rappel théorique : Le Mocking

Quand vous voulez faire des tests, certaines parties de vos algorithmes sont hors de votre contrôle. Un appel à un serveur externe, le tirage d'une valeur aléatoire, etc. peut faire rater vos tests ou vous empêcher de les créer.
Dans ce but, le mocking a pour but de créer des faux objets que vous contrôlez intégralement. Chaque objet que vous « mockez »(i.e., initialisez) sera utilisable comme n'importe quel autre objet du même type. La principale différence est que vous définissez le retour de chaque appel de méthode, sans que le code de la méthode ne soit actuellement exécuté. Si vous avez une méthode tirant un nombre aléatoire, le mocking vous permet donc de lui faire retourner un résultat fixe, que vous choisissez.
Une librairie très connu de mocking est Mockito. [a] Il vous est conseillé d'aller sur leur site, comme celui-ci donne un exemple simple, mais suffisant, pour comprendre comment utiliser la librairie. Cependant, un exemple vous est aussi donné au Listing 2.

Listing 2 – Exemple de mocking

```java
import java.util.LinkedList;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
public class Mockito {

    @Test
    public void testList() {
        LinkedList mockedList = mock(LinkedList.class);
        when(mockedList.get(0)).thenReturn("first");
        System.out.println(mockedList.get(0)); // first
        System.out.println(mockedList.get(999));
        // null as not stubbed (i.e. no when)
    }
}
```

Notez que vous pouvez aussi utiliser Mockito pour « mockez »des méthodes statiques (voir `mockStatic` [b])

a. https ://site.mockito.org/
b. `https://javadoc.io/static/org.mockito/mockito-core/5.14.2/org/mockito/Mockito.html#48`

(a) Commencez par implémenter un objet `Lotto` qui tire au sort une combinaison de six nombres. Celui-ci possède une méthode `draw` générant une nouvelle combinaison et `check` testant si la combinaison donnée en paramètre est gagnante ou non. Vous pouvez supposer que une combinaison est un tableau d'entier de taille fixe.

(b) Implémentez ensuite un objet `Player` possédant une méthode `play()` retournant si l'utili-

sateur a gagné (vrai s'il gagne, faux sinon).

(c) Enfin, faites un (ou plusieurs) test(s) unitaire(s) testant la méthode `play` et son retour. Utilisez le principe du mocking afin de contrôler le test de la combinaison.

Exercice 2.3 : Des chiots anormaux (∗) 🟢

Cette question est inspirée d'une question de l'examen d'août 2024 à Charleroi. La question (comme vous le remarquerez sûrement) a été réécrite pour les TPs, mais la difficulté ainsi que le sujet testé sont égaux.

Durant votre passage journalier devant l'animalerie locale, vous voyez de nombreux chiots jouant ensemble à l'intérieur. Vous vous sentez hypnotisé par ces chiots et avez une incroyable envie d'aller les rejoindre et de vous rouler au sol avec eux.

Sans vous retenir, vous les rejoignez et vous amusez comme un fou avec les chiots. Vous vous sentez heureux entourés de tant de chiot, de couleurs différentes, avec des marques différentes, de races différentes, et pourtant, tous sont d'accord de jouer avec vous. Après leur avoir donné chacun une friandise, vous les quittez pour retourner à votre journée.

Sur le chemin, vous croisez une employée de l'animalerie qui vous a vu jouer et vous explique que certains chiots sont considérés comme des anomalies : ils ont des combinaisons de traits génétiques qui sont rares ou jamais vues jusqu'ici. Durant votre discussion, l'employée vous énonce que l'animalerie dispose d'un algorithme afin de détecter ces "anomalies", mais que celui-ci se trompe parfois.

Curieux de savoir à quel point cet algorithme ne fonctionne pas, vous décidez de faire la meilleure chose que vous pouvez faire : faire des tests unitaires pour tester toutes les combinaisons.

Selon l'employée, il y a 3 catégories considérées : la race, la couleur, les patterns visibles sur son pelage. Sont considérées 350 races, environ 15 couleurs et 20 patterns. Cela représente environ $350 * 15 * 20 = 105.000$ possibilités à tester. Incroyablement motivé, vous décidez de *ne pas* faire ça.

A la place, vous changez de technique. Vous allez uniquement faire certains des tests selon les paires. Après tout, il est empiriquement connu que ces anomalies viennent d'une combinaison de deux catégories en général. Vous avez un souvenir lointain, un peu flou, où votre professeur vous en avez parlé. A ce moment-là, vous vous disiez que ça ne serait jamais à l'examen et que vous n'utiliserez sûrement jamais ça. [2] Heureusement, vous vous souvenez du nom : il l'avait appelé le "pairwise testing".

Rappel théorique : Pairwise testing

Comme l'analogie de cet exercice le montre, il arrivera parfois des moments où vous serez incapable de tester toutes les entrées. Parfois, cela arrivera du à un nombre "infini" d'entrées possibles, parfois, juste du au grand nombre de combinaison possible.
Certaines techniques permettent de réduire significativement ce nombre en choisissant les "cas intéressants" à tester. Une de ces techniques s'appelle le "pairwise testing" (test par combinaison de pairs si on le traduit littéralement en français). Le principe se base sur des évidences empiriques que les bugs les plus communs viennent généralement d'un seul paramètre ou d'une paire de paramètre.
Vous pouvez en général suivre ces étapes afin de savoir quelles sont les combinaisons que vous devez tester selon le pairwise testing :

1. Réduire le nombre de possibilités via des classes d'équivalences : à la place de consi-

2. Si vous hésitez, relisez le début de cet énoncé

dérez toutes les entrées possibles, essayez de rassembler certaines entrées en classes. Par exemple, si vous avez un paramètre entier qui doit être en 0 et 50 pour une partie de l'algorithme, 50 et 100 pour un autre, et toute autre valeur est invalide, cela vous donne 3 classes : $[0-50]$ (VALID 1), $[51-100]$ (VALID 2), les autres (INVALID).

2. Trier les arguments de manière croissante selon leur nombre de possibilités. Par exemple : arg1 (4 possibilités), arg2 (3 possibilités), arg3 (2 possibilités).
3. Calculer le nombre de combinaisons en multipliant les deux plus grands nombres. Par exemple : $4 * 3 = 12$ tests.
4. Créez un tableau avec autant de colonnes que d'arguments, et autant de lignes que de combinaisons. Par exemple : tableau de 3 colonnes (3 arguments) et 12 lignes (12 combinaisons selon le calcul précédent).
5. Enfin, remplissez ce tableau. Commencez par remplir les deux premières colonnes en mettant toutes les paires possibles. Ensuite, compléter les suivantes afin que toutes les paires soient testées. Par exemple (où A, B, C, D sont les différentes classes d'équivalence pour chaque argument) :

arg1	**arg2**	**arg3**
A	A	A
A	B	A
A	C	B
B	A	B
B	B	A
B	C	A
C	A	A
C	B	B
C	C	A
D	A	A
D	B	B
D	C	A

(a) En ne considérant que vos 4 races préférées, 3 couleurs et 3 patterns (vous êtes libre de les choisir), énumérez les tests que vous devriez faire pour respecter le pairwise testing.

Le saviez-vous?

Le pairwise testing simplifie votre suite de test. Cependant, il est évident qu'il ne résoudra pas tous les bugs possibles comme une énumération complète. Selon votre point de vue, ça peut donc être considéré comme une "heuristique de test", ça va plus vite (pour vous et en éxécution), mais diminue un peu l'assurance que tout est bon en contre-partie.

Le saviez-vous?

Vous pouvez vous inspirer d'un outil externe ou librairie afin d'avoir une idée d'une des solutions. Par exemple, `allpairspy` [a] vous donne une solution parmi toute celle possible. Cependant, sachez qu'il ne vous donne pas forcément la même solution que vous, même si votre solution est possiblement correcte aussi.

a. `https://github.com/thombashi/allpairspy`

Exercice 2.4 : Fibonacci

Cette question est reprise de l'examen d'août 2014. Elle a été légèrement modifiée.

La suite de Fibonacci est une suite d'entiers très connue dans laquelle chaque entier est la somme des deux entiers précédents. Voici les dix premiers termes de cette suite :

```
0 1 1 2 3 5 8 13 21 34 ...
```

La méthode itérative de la classe Fibonacci permet de calculer la valeur du $n^{\text{ème}}$ élément de cette liste, en comptant à partir de 0.

Nous disposons des tests unitaires de la classe FibonacciTest pour tester cette méthode itérative.

(a) Prédisez le résultat de chacun des tests (succès, échec ou erreur) et expliquez pourquoi ce résultat se produit.
Nous parlons bien de prédire. Vous devez donc faire l'exercice avant *d'exécuter les tests.*

(b) Pour les tests qui produisent une erreur ou sont en échec, corrigez-les pour qu'ils réussissent (tout en restant des tests utiles).

(c) Écrivez un quatrième test unitaire test5() vérifiant que la méthode fibo est capable de calculer le 1000ème élément de la suite de Fibonacci en moins d'une seconde.

Exercice 2.5 : Les équations du second degré

Cette question est reprise de l'examen de janvier 2010. Elle a été reprise sans modification de son contenu.

Soit l'équation du second degré suivante : $ax^2+bx+c=0$. On crée une classe `SolveEquation` chargée de determiner les valeurs réelles de x telles que l'équation est vérifiée, en fonction des paramètres a, b et c. Les valeurs complexes représentant une solution de l'équation ne sont pas considérées. En fonction des paramètres a, b et c, il peut y avoir 0, 1, 2, ou une infinité de solutions pour x. La méthode `getNbSolutions` retourne ce nombre de solutions. Le cas $a = b = c = 0$ pour lequel il existe une infinité de solutions pour x n'est pas traité par cette méthode, qui lève une exception de type `ArithmeticException` lorsque cette situation se présente.

Les tests unitaires de la classe SolveEquationTest servent à tester la classe `SolveEquation`.

(a) Prédisez le résultat (succès, échec, erreur) pour les trois tests `test1()`, `test2()` et `test3()`. Pour chacun des résultats, expliquez pourquoi le résultat se produit.

(b) Corrigez ces tests afin qu'aucun d'entre-eux ne soient en échec ou en erreur.

(c) Écrivez un test unitaire `test4()` qui vérifie que l'équation $3x^2 = 0$ possède une et une seule solution.

(d) Proposez une nouvelle version de `test1()` qui évite d'appeler la méthode `getSolutions()` plusieurs fois, en utilisant la méthode `Pair.equals(Object)`.

2.2 Décorateur

Exercice 2.6 : Le grand maître de la décoration (∗)

Vous étiez en train de vous diriger vers le très réputé parc d'attraction DesignPatternsAreFunny quand vous apercevez de la lumière au loin. Curieux, vous vous dirigez vers la lumière.

Vous êtes surpris par un chiot traversant votre chemin. Vous vous approchez de lui pour le caresser et il se laisse faire. Alors que vous passiez le meilleur moment de votre journée avec le chiot, quelqu'un cria soudainement derrière vous, vous faisant peur, et le chiot couru vers lui. Il s'agissait en fait du maître de cet adorable animal.

Le maître avait constaté l'attention que vous donniez à son fidèle compagnon. « Il est mignon n'est-ce pas ? »vous demanda le maître, sur quoi vous ne pouvez qu'acquiescer. Le maître se présenta : « Je suis le maître de la décoration, et lui, c'est Hazel, mon fidèle compagnon ». Il continua : « Tu as l'air de bien aimer mon chiot. Je te propose un marché : si tu arrives à relever mon défi, je te montrerai une photo d'Hazel avec un chapeau de Noël ». Ce fut une proposition tellement alléchante que vous avez accepté à peine la phrase finie.

Il vous explique que son défi consiste à décorer un objet. « Facile, juste une ou deux guirlandes et c'est fini »vous pensez (à voix haute visiblement). Il rigola et vous expliqua qu'il ne parlait pas de ce genre de décoration mais qu'on s'en rapprochait.

Pour commencer, il vous demanda de retrouver le code de la calculatrice fait précédemment. « Comment est-il au courant de ça ? »vous demandez-vous.

Une fois que vous l'avez retrouvée [3], il vous éclaira un peu sur sa définition de décoration.

Rappel théorique : Le design pattern décorateur

Si vous mettez des décorations de Noël sur un arbre, vous ajoutez des décorations sur l'arbre sans changer l'arbre. Vous le rendez en quelque sorte plus attirant, mais l'arbre en dessous est toujours le même. De même, vous pouvez mettre des décorations au-dessus des décorations de Noël déjà présentes. Vous ne changez ni l'arbre ni les décorations précédentes, mais vous y rajoutez de la beauté.
Le design pattern décorateur se base sur ça. Nous voulons ajouter des fonctionnalités au-dessus d'un objet (déjà décoré ou non), sans pour autant modifier l'objet. Ainsi, ce design pattern fait intervenir 3 types de classes :

1. Une interface commune : cette classe abstraite ou interface définit comment interagir avec les objets (décorés ou non) du design pattern.
2. Une classe de base : c'est la classe de base que nous voulons décorer.
3. Une classe décorateur : c'est la classe qui va décorer celle de base ou une classe déjà décorée. Notez qu'il prend l'objet qu'il doit décorer en paramètre (souvent défini dans son constructeur).

Pour reprendre notre analogie, l'interface commune définirait qu'on pourrait regarder l'arbre, l'objet de base serait l'arbre et l'objet décorateur serait les décorations.
Si nous devions le représenter sous forme d'un diagramme de classe, cela ressemblerait à la Figure 1.

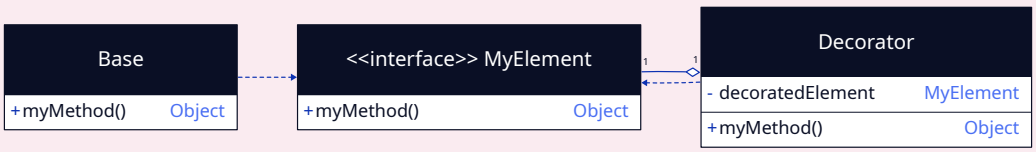

FIGURE 1 – Un diagramme de classe pour le design pattern décorateur

Ce design pattern est principalement utilisé quand nous voulons rajouter une fonctionnalité en se basant sur un objet existant sans pour autant changer l'objet existant. Comme

3. Alternativement, vous pouvez aussi le prendre des solutions sur Moodle si celles-ci sont postées.

exemple réel, BufferedFileReader est une classe Java décoratrice de FileReader. En réalité, le buffer utilise une instance de FileReader pour lire le fichier, mais y ajoute une fonctionnalité de buffer.

« Te voilà éclairer sur ce qu'est une décoration. C'est simple, non ? »s'empressa-t-il de demander. Votre réponse positive l'encouragea à enfin vous donner les détails de votre défi :

(a) Faites en sorte, en respectant le design pattern décorateur, d'avoir un cache sur l'exponentiation de votre calculatrice. En d'autres termes, créez une nouvelle calculatrice, dotée d'un cache. Si nous demandons à la calculatrice deux fois le même calcul d'exponentiation, la calculatrice ne le recalcule pas, mais reprend la réponse qu'elle a déjà calculée précédemment.

Le saviez-vous?

Si vous souhaitez rendre votre cache un peu plus efficace, vous pouvez regarder du côté de la méthode `computeIfAbsent` des sous-classes de `Map` (et donc de `HashMap`) en Java.

Impressionné par votre réponse, le grand maître de la décoration vous montra la photo promise. [4]

Exercice 2.7 : Un raspberry PI gaming ?

On aime tous les Raspberry PI, pas vrai ? Cependant, dans notre monde actuel, il est impensable de ne pas y ajouter un aspect un peu plus gaming, en y rajoutant des LEDs par exemple. Nous vous demandons donc :

(a) Implémentez une interface Computer ainsi qu'une implémentation RaspberryPI de cette interface. Plusieurs méthodes donnant une efficacité (un nombre) et un effet de fascination (un autre nombre).

(b) En respectant le design pattern décorateur, implémentez une classe RaspberryPIGaming qui améliore ces performances et cet effet de fascination pour un nombre donné. [5]

2.3 Observateur

Exercice 2.8 : Le grand maître de l'observation (∗)

Durant vos exercices quotidiens, vous vous sentez observé. Encore plus bizarre, chaque fois que vous faites un mouvement, vous entendez un cri aiguë qui décrit votre position : « Debout ! », suivi d'un rire, « Penché ! », suivi d'un autre rire. Après avoir attentivement observé les alentours, vous voyez une petite fille qui vous observe depuis le jardin de votre voisin. Prenant au jeu, vous continuez vos exercices et commencez à vous amuser à l'entendre crier chaque fois que vous changez de position. Au bout d'un moment, vous décidez d'aller lui parler. « C'est un exercice de TP après tout, faut bien faire avancer les choses... »pensez-vous.

Une fois arrivé près d'elle, vous lui demandez ce qu'elle fait. Elle vous répond très simplement « J'observe ! », suivi d'un rire. « Évidemment, question idiote »pensez-vous. Vous lui demandez

4. *Votre moteur de recherche préféré* est votre ami.
5. Qui a dit que les LEDs étaient juste cosmétique ?

ensuite pourquoi elle crie votre position chaque fois que vous levez votre main. Elle vous répond qu'un drôle de personnage qui passait par là lui a appris ce jeu en lui disant que cela s'appelait « L'observateur ». « Un monsieur drôle a joué avec moi à l'Observateur. Il m'a dit que chaque fois que je changerai de position, il décrirait en criant ma position. C'était marrant ! »Ce drôle de personnage, dans toute sa sagesse, l'a aussi prévenu de toujours demander au gens avant d'observer les gens. « Je voulais y jouer et tu étais là. »dit-elle avec un sourire.

Vous voyez un air de tristesse se former sur son visage quand vous lui dites que vous avez maintenant fini vos exercices. Elle recommence à sourire quand une idée fabuleuse lui vient à l'esprit : « Je sais ! Tu peux me donner un simulateur qui jouera avec moi. »Vous voilà donc parti dans la programmation d'un simulateur pour une petite fille, qui vous donne des détails sur comment il doit être fait. Il ne faudrait pas que son simulateur soit fait n'importe comment !

Rappel théorique : Le design pattern observateur

Vous avez certainement déjà rencontré ou pouvez imaginer un problème où vous souhaitez *observer* tous les changements d'un objet. C'est le principal but du design pattern Observateur. Il vous permet de demander à un objet de vous prévenir à chaque fois qu'il change.
Il fait principalement intervenir deux types de classes :

— Les observateurs : ce sont les classes qui vont être prévenues des changements

— Les observables (ou Subjects) : ce sont les classes qui seront observées / qui vont prévenir qu'elles ont changées.

De manière générale, un observateur va appeler la méthode `attach()` d'un observable. A partir de ce moment, l'observable le préviendra chaque fois qu'il change via sa méthode `notifyObservers()` qui notifiera tous les observateurs qui se sont attachés.
La Figure 2 donne un diagramme de classe de la structure du design pattern observer.

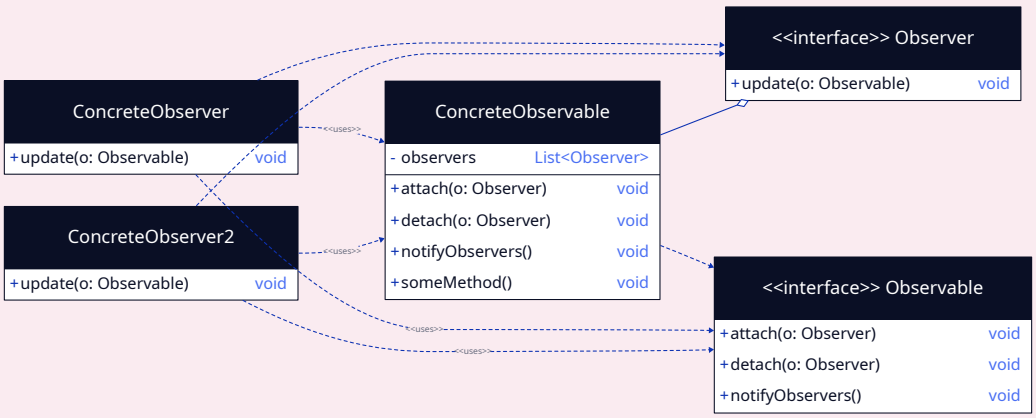

FIGURE 2 – Un diagramme de classe pour le design pattern observateur

(a) En respectant le design pattern observateur, implémentez un similateur de la situation décrite ci-dessus. Commencez par créer un objet `ConsentingPerson` qui contient juste une position, ainsi qu'une objet `ObservingGirl` qui servira à observer ce personnage. Pour le moment, cette classe contient juste une méthode qui affiche un message quand on lui donne une position en paramètre

(b) En respectant le design pattern Observateur, faites en sorte que chaque fois que la position

de votre objet `ConsentingPerson` change, votre objet `ObservingGirl` affiche un message.

Exercice 2.9 : Un progrès observé (*)

Vous étiez en train de développer une application de gestion de tâche. Le but était simplement que celle-ci vous permettent de créer des tâches et *d'observer* leur progrès au fur et à mesure du temps.

Vous avez pensé les choses d'une manière très simple. Une classe abstraite `Task` vous permet de représenter une tâche ainsi que sa logique métier. Elle contient aussi un attribut en entier représentant son progrès. Une classe `Manager` maintient une liste des tâches et affiche leur progrès dans le terminal au fur et à mesure. Elle vous permet de démarrer une tâche ainsi que de savoir combien de tâche sont en attente ou terminée.

(a) Implémentez les classes de base nécessaires. Profitez-en pour déjà implémenter deux exemples de tâches. Si vous n'avez pas envie de chercher, vous pouvez juste attendre dans une boucle for.

Le saviez-vous?

L'exercice ne vous demande pas de faire tourner les tâches en parallèle. Si vous savez déjà comment faire, vous pouvez l'implémenter. Veuillez cependant prendre les précautions qui s'imposent (bien que je ne vous y oblige pas).

Afin de savoir le progrès de chaque tâche, vous décidez d'utiliser un design pattern Observateur afin d'observer chaque tâche pour des changements dans son progrès.

(b) Implémentez un design pattern observateur dans le but d'observer chaque tâche et d'imprimez les changements de progrès dans le terminal.

« C'est sympa tout ça, mais j'aimerai bien garder une trace de ça dans un fichier, tout en continuant de voir l'avancement des tâches... »vous dites-vous.

(c) Implémentez un second observateur qui écrit dans un fichier le progrès. Profitez-en pour rajouter une indication de temps.

Le saviez-vous?

Vous pouvez obtenir une représention textuelle standard de l'heure actuelle via java.time.LocalDateTime.now().toString()

Exercice 2.10 : Ils sont toujours aussi mignons !

Le saviez-vous?

L'histoire de cet exercice continue celle d'un autre exercice (dans le design pattern Stratégie). Les exercices sont cependant indépendants. Il est donc faisable sans avoir fait l'autre exercice.

Vous voilà de retour à l'animalerie. Aussitôt que vous arrivez, vous voyez des chatons partout. N'ayant pas changé depuis la dernière animalerie, vous vous exclamez « Ils sont toujours aussi mignons ! ». Vous n'êtes cependant pas surpris quand les chats se faufilent dans des endroits hors de votre portée. Grâce à votre don, vous comprenez tout de suite que les chats souhaitent votre

aide en échange de se laisser câliner. Vous voilà *mystérieusement* dans une situation de déjà-vu, ce que les chats semblent remarquer. Cependant, ils vont font remarquer que vous avez des choses plus importantes à faire que de réfléchir à cela.

A votre très grande surprise, les chats ont le même objectif qu'avant : ennuyer le personnel de l'animalerie. Les chats vous expliquent que la stratégie que vous aviez proposée précédemment semble fonctionner. Cependant, ils sont incapables de jauger le désespoir grandissant chez le personnel. Ils vous demandent donc d'observer attentivement le personnel, et de reporter le moindre changement que vous remarquez.

Avant d'accepter, vous vous demandez comment les chats d'une autre animalerie peuvent savoir la stratégie que vous avez proposée ailleurs. Les chats communiquent-ils entre eux ? Est-ce un complot ? Cette question est vite chassé de votre esprit lorsqu'un chaton passe devant vous : « IL EST TROP MIGNON ! ».

Ayant oublié ce que vous pensiez, vous revenez à votre mission :

(a) Commencez par implémenter (ou adapter de cette *mystérieuse* impression de déjà-vu) toutes les classes de base de cet exercice (e.g., Chat, Employé, Personnage, etc.).

(b) A l'aide d'un design pattern Observateur, faites en sorte que vous (le personnage) puissiez observer chaque employé pour des quelconques changements. Les chats ont une bonne ouïe, vous pouvez donc chuchoter tous changements dans le terminal, ils comprendront.

Exercice 2.11 : Un joueur qui se déplace

Cette question est reprit de l'examen du 11 août 2023 à Mons.

Il vous est demandé de développer le squelette d'un jeu multijoueur en réseau. Le serveur maintiendra une liste de tous les joueurs actifs et leurs positions actuelles dans le monde du jeu. Chaque joueur contrôlera un personnage dans le jeu, et le serveur devrait diffuser des mises à jour à des vues indiquant la position de chaque personnages chaque fois qu'il se déplace. Veuillez écrire le code en langage Java qui implémente ce logiciel en utilisant le design pattern **Observer** pour implémenter le mécanisme de notification de changement de positions des personnages. L'utilisation du pattern Observer doit être claire et évidente dans le code. Il ne vous est pas demandé de développer des interactions en réseau. [6] Tout le code sera donc dans un seul projet Java, le but étant juste d'illustrer le pattern Observer. Vous devez donc :

— implémenter les classes et interfaces Java nécessaires pour ces données et fonctionnalités :
  — `Player` : représente un seul joueur dans le jeu. Cette classe doit avoir des champs pour un nom et la position x, y, et z de son personnage dans le monde du jeu.
  — Subject : une interface sujet
  — Engine : représente le moteur du jeu. Cette classe devrait avoir une liste de tous les joueurs actifs. Cette classe est le sujet. Elle dispose aussi notamment de méthodes qui permettent d'ajouter des joueur et de modifier leurs positions, etc..
  — Observer : une interface pour observer les mises à jour concernant un joueur. Cette classe devrait avoir une méthode de mise à jour update() lorsqu'un joueur se déplace.
  — Display : un observateur concret qui imprime le nom et la nouvelle position du joueur sur la console.
  — Display2 : une second observer concret qui est une classe dérivée de Display.
— implémenter une classe contenant une fonction main afin de démontrer le fonctionnement du logiciel :

6. Libre à vous de le faire chez vous. Cependant, c'est *largement* en dehors du cadre des TPs.

— Dans ce main, instanciez un Engine, des Player, un Display, un Display2. Faites ce qui est nécessaire pour que les observeurs soient attachés au sujet. Changez la position d'un Player et vérifiez que les deux observeurs sont notifiés. Détachez un des observateurs. Changez la position d'un Player et vérifiez que seul un des deux observeurs est notifié.

3 Gradle

3.1 Introduction à Gradle

Gradle est un outil permettant d'automatiser certaines tâches lors du développement de vos applications. En effet, lorsque vous souhaitez réaliser un *.jar* exécutable, il est souvent nécessaire de d'abord compiler votre projet Java et ensuite de créer le fichier *.jar*. Grâce à Gradle, il est possible d'automatiser ces deux processus derrière une seule commande à exécuter. Il est également possible d'inclure des commandes dans ces tâches qui permettent d'ajouter de nombreuses fonctionnalités, tel que déplacer des ressources de votre application dans les bons répertoires pour la mise en production, générer la documentation Java de votre application, etc. Gradle permet également d'importer des librairies externes dans votre application et de gérer les éventuelles dépendances pour le bon fonctionnement de celles-ci.

Nous utiliserons Gradle dans ce TP pour exécuter les applications et les tests unitaires, ainsi que pour importer JavaFX dans le projet.

3.2 Installer Gradle

Pour installer Gradle, commencez par installer le *Java Development Kit* (JDK). Pour ce faire, installez le JDK correspondant à votre système. Une fois le JDK installé, testez votre installation en ouvrant un terminal et en tapant les commandes `java –version`, puis `javac –version`. Les deux commandes doivent vous donner une version de Java.

Suivez ensuite les instructions sur le site officiel pour installer Gradle (`https://gradle.org/install/`). Entrez alors la commande `gradle -v`. Gradle vous affichera alors un message de bienvenue ainsi que différentes versions pour différents langage.

3.3 Création d'un projet Gradle

Ouvrez un terminal et utilisez la commande `cd` pour vous déplacer dans le dossier où vous souhaitez créer le projet. Créer un dossier à cet emplacement (via votre explorateur de fichiers ou avec la commande `mkdir`) et placez votre terminal à l'intérieur de celui-ci. Dans ce protocole, nous supposons que le projet se trouvera dans le répertoire `D:\\myproject`.

Utilisez alors la commande `gradle init` pour créer votre projet Gradle. Choisissez le type `application` et gardez les autres options par défaut.

Le code source du projet se trouve dans le dossier `app/src/main/java` et le code des tests unitaires est dans `app/src/test/java`.

3.4 Utiliser Gradle

3.4.1 En ligne de commande

Pour exécuter votre application en ligne de commande, utilisez la commande gradlew run. Un message indiquant BUILD SUCCESSFUL devrait s'afficher si tout est correct. De même, pour exécuter les tests unitaires, utilisez la commande gradlew test.

Notez qu'on utilise ici la commande gradlew et non plus gradle ! gradlew est un wrapper généré par Gradle. Il sera introduit dans la Section 3.7.

3.5 Rapport de testing

Lorsque vous exécutez les tests unitaires, un rapport au format HTML est généré par Gradle. Ce rapport se trouve dans `app\\build\\reports\\tests\\test`. Ouvrez le fichier index.html dans votre navigateur pour visualiser les informations concernant les tests qui ont réussi ou échoué.

3.6 Ajouter Mockito à Gradle

Si vous souhaitez développer des test unitaires qui utilisent des doublures d'objets avec Mockito, vous pouvez lire cette section. Si pas, passez à la section suivante.

Modifiez la section dependencies de votre *build.gradle* pour y ajouter le support de Mockito :

```
1  dependencies {
2      // Some other things
3
4      // Use JUnit Jupiter for testing.
5      testImplementation("org.junit.jupiter:junit-jupiter:5.9.3")
6
7      // Add mockito to the dependencies
8      // You can change the version to a newer one
9      implementation("org.mockito:mockito-core:5.7.0")
10 }
```

3.7 Le wrapper : Gradle sans installation

Si le projet doit être compilé sur une machine sans Gradle ou sur laquelle la version disponible est trop vieille, Gradle permet de générer un *wrapper*. Il s'agit d'un script se comportant comme Gradle mais qui, lors de la première utilisation, va télécharger la version de Gradle à utiliser.

3.7.1 Générer le wrapper

Pour générer le wrapper, vous devez ouvrir le dossier du projet dans l'invite de commande et utiliser la commande ci-dessous.

```
1  gradle wrapper --gradle-version 7.4.1
```

Pour une autre version de Gradle, remplacez le `7.4.1` par la version souhaitée. Cette commande va générer les deux fichiers `gradlew` et `gradlew.bat`. Le premier est le wrapper à utiliser sous GNU/Linux et MacOS, le second est à utliser sous Windows.

3.7.2 Utiliser le wrapper

Pour utiliser le wrapper Gradle, il suffit de remplacer la commande `gradle` par `./gradlew` sous MacOS ou GNU/Linux et par `gradlew.bat` sous Windows comme illustré ci-dessous.

```
1 // Remplacer...
2 gradle run
3 // par...
4 ./gradlew run      //sous GNU/Linux et MacOS
5 gradlew.bat run    //sous Windows
```

Dans un IDE, le wrapper Gradle est normalement utilisé par défaut.

3.7.3 Erreur : permission non-accordée

Sous MacOS ou GNU/Linux, il est possible que vous ayez une erreur de permission lorsque vous essayez d'exécuter le wrapper. Cette erreur vient simplement du fait que le fichier gradlew n'a pas la permission de s'exécuter sur votre machine. Pour résoudre ce problème, il suffit d'exécuter la commande ci-dessous en invite de commande.

```
1 chmod +x gradlew
```