IUT Sénart-Fontainebleau, UPEC
DEV 2.2
IUT Info 1A

Florent Madelaine
Luc Dartois
Luc Hernandez
2021-22

Programmation défensive et tests unitaires

Programmation défensive

Essentiellement, il s'agit de tout faire pour que les méthodes publiques ne puissent être utilisées que dans le cadre prévu et de tout faire pour garder nos objets dans un état cohérent.

Il s'agit donc ici d'une bonne pratique de génie logiciel permettant à un développeur qui utilise notre classe de recevoir des messages d'erreurs adaptés si il utilise notre travail d'une manière qui n'est pas normale.

Un premier ingrédient technique utile est de faire en sorte par exemple que tout objet passé en paramètre d'une méthode soit bien instancié (pas null). Une facilité offerte par les méthodes `Objects.requireNonNull(c,...` de la librairie `java.util.Objects`. Vous pouvez consulter la documentation pour plus de détails, mais en gros une exception `NullPointerException` sera levée si c est null, avec en option un message d'erreur adapté.

Un autre ingrédient utile consiste à ajouter un traitement à chaque méthode publique pour vérifier la ou les pré-conditions. En cas de problème, on peut lever une exception adaptée. En particulier, si les paramètres ne sont jamais acceptés, on peut soit changer la classe en introduisant un type d'entrée plus restreint, soit lever une `IllegalArgumentException`. Si c'est juste un soucis temporaire, on peut lever une `IllegalStateException`.

Vérifier

On peut tester les invariants et post-conditions avec le mécanisme d'assertion natif de java.

Il suffit d'ajouter `assert( ... un test ...)`. Vous pouvez ajouter un second argument optionnel qui est un message à afficher en cas de non réalisation de la condition de test (on souhaite que le test soit satisfait).

Il faut lancer java avec le flag `-ea` pour activer les assertions.

C'est un mode debug simple et efficace quand on code une méthode un peu compliquée.

On peut également faire des tests unitaires dans un main.

Exercice 1. Allez étudier le fichier `MonInt.java`, compilez le et étudier le résultat des tests en lançant avec ou sans l'option activant les assertions.

Notez bien les différents mécanismes évoqués ci-dessus. Le fichier comporte des exemples (parfois bêtes) de tous ces mécanismes.

Ouvrez un fichier `Ma-Touille-pour-les-tests.txt` et Copiez dedans des informations qui ont l'air utiles.

Mise en pratique

Suite à notre exercice de modélisation sur la partie modèle du jeu de Memory en TD, je vous donne aujourd'hui une « solution ». J'ai volontairement (ou pas, qui sait ?) laissé de nombreuses erreurs.

Il y a un exemple de fichier JUnit pour la classe `Carte`.

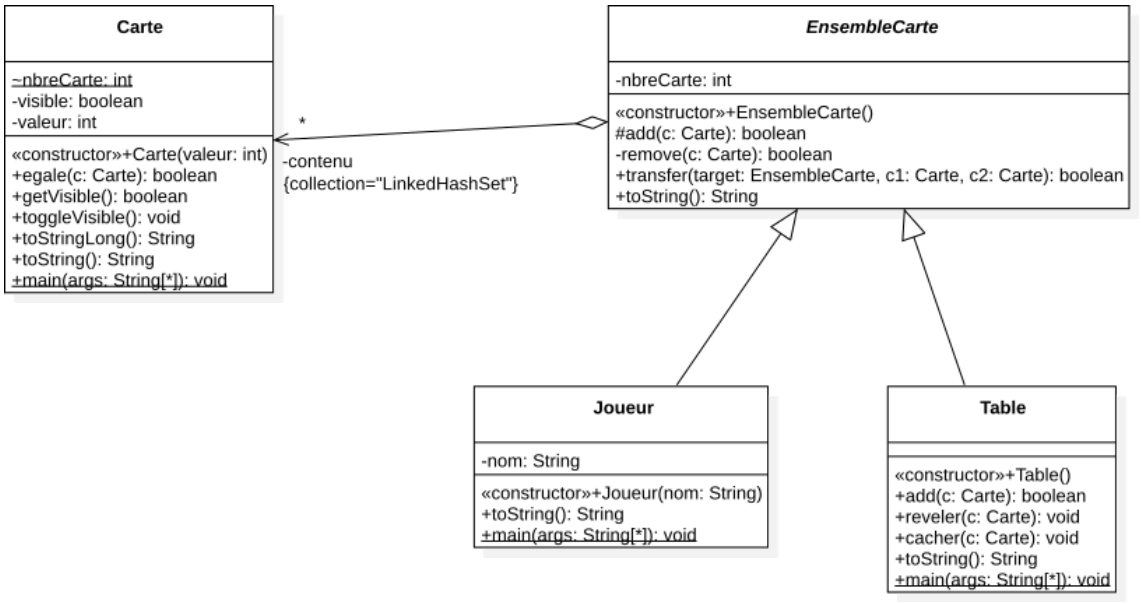

FIGURE 1 – Diagramme de Classe (outil reverse code de star UML)

Exercice 2. Examinez rapidement le diagramme de classe ci-dessus. Regardez le code `v0Memory.zip`. Compilez le. Testez le. Pour ce faire vous avez le main qui contient des tests à l'ancienne (avec des print). Réfléchissez à des assertions que vous pourriez ajouter dans le code.

Découverte des tests unitaires avec Junit

Pour préparer l'utilisation plus poussée de JUnit la semaine prochaine, vous pouvez dès à présent découvrir cet outil bien pratique.

Pour commencer, il faut indiquer à la compilation (à javac) où se trouve les deux jar nécessaires à l'utilisation de JUnit. À l'IUT ce sont respectivement `/usr/share/java/junit.jar` et `/usr/share/java/hamcrest-c`

Vous pouvez le faire de plusieurs manières. soit vous lancez javac avec l'option `-cp` en indiquant à chaque fois au moins ces deux fichiers.

Soit vous changez globalement votre `CLASSPATH`.

À l'IUT, il suffit de saisir cette ligne dans votre terminal
`CLASSPATH=".:/usr/share/java/junit.jar:/usr/share/java/hamcrest-core.jar"` (vous pouvez aussi adapter votre classpath dans un fichier de config comme `.bashrc` si vous voulez quelque chose de plus permanent).

Exercice 3. Lancez les tests JUnit proposés (classe TestCarte) en faisant :
`:$ java org.junit.runner.JUnitCore TestCarte` Des tests sont passés avec succès, d'autres non. Essayez de remédiez à la situation.

Bonne chasse au bugs !

Pour aller plus loin

Si vraiment vous êtes trop rapides, vous pouvez commencer le travail de la semaine prochaine.

Exercice 4. Inspirez vous des tests donnés et ajoutez des tests JUnit pour les autres classes. Trouvez tous les merveilleux bugs bien croquants que j'ai laissé partout (volontairement ou pas).