TP

JUnit

Dans ce TP, vous allez utiliser JUnit pour mettre en place une suite de tests unitaires. Pour cela, nous allons utiliser une implémentation de listes chaînées comme cas d'étude.

1 Déroulement du TP

- Le TP est à faire par groupe d'**au plus deux personnes**.
- Ce TP est organisé en plusieurs parties, seule la première est à inclure dans le rendu, mais il est fortement recommandé de faire la partie *Couverture de code* si vous avez le temps car cela pourra vous être utile pour le projet de fin de semestre. Notez que dans la partie *Pour aller plus loin*, un exercice est dédié au *Test-Driven Development*.
- **Rendu** : Pour ce TP, le rendu est une archive (`devopsTP3_nombinome1-nombinome2.tar.gz`) incluant le code source fourni et les tests écrits pendant le TP. Pensez à ajouter un fichier AUTHORS contenant le nom des membres du binome.
  — Le rendu concerne la partie 3 du TP.
- La date limite pour rendre le TP est **Lundi 4 Mars**.
- Notez que le travail fait pendant ce TP sur le test de la classe `MyUnsortedList` (voir Section 3) sera réutilisé lors d'un prochain TP.

2 Note sur les IDEs

JUnit est supporté par plusieurs IDE. Dans ce TP, nous vous demandons cependant de travailler sous Eclipse. L'intégration de JUnit avec Eclipse est très simple (voir description ci-après).

L'utilisation de VS Code pour ce TP n'est pas conseillée (aucune aide ne sera apportée), à moins que vous ayez déjà une bonne expérience de développement en Java avec VS Code.

3 Une liste testée avec JUnit

Pour avoir l'occasion d'écrire de nombreux cas de tests avec JUnit sans passer trop de temps sur le code applicatif, nous vous proposons une implementation d'une liste générique [1] simplement chaînée. Les méthodes principales de cette liste chainée sont définies dans l'interface `UnsortedList`. Une première implementation vous est fournie dans la classe `MyUnsortedList`.

Le code de base vous est fourni dans l'archive `unsorted_list.tar.gz` disponible sur Moodle. Commencez par extraire l'archive et importez le projet dans Eclipse.

Spécification de l'interface `UnsortedList` L'interface `UnsortedList` spécifie les services fournis par une liste d'éléments non triés. Voici une courte description de cette interface, une description plus détaillée est disponible dans le code source.

- `boolean isEmpty()` : Retourne true si la liste ne contient aucun élément.
- `int size()` : Retourne le nombre d'éléments dans la liste.
- `void prepend(E element)` : Ajoute un élément au début de la liste.
- `void append(E element)` : Ajoute un élément à la fin de la liste.

1. Pour un rappel très rapide sur la généricité en Java : `http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/genericite.htm`

- `void insert(E element, int pos)` : Insère un élément à la position spécifiée. Si la valeur de `pos` est `0`, l'élément est inséré en tant que premier élément de la liste. Si la valeur de `pos` est égale à la taille de la liste, l'élément est inséré en tant que dernier élément. Une `IndexOutOfBoundsException` est levée si la position n'est pas valide.
- `E pop(void)` : Enlève l'élément en première position dans la liste et le retourne. Si la liste est vide une exception de type `EmptyListException` doit être levée.
- `E popLast(void)` : Enlève l'élément en dernier position dans la liste et le retourne. Si la liste est vide une exception de type `EmptyListException` doit être levée.
- `E remove(int pos)` : Enlève et retourne l'élément contenu à la position donnée. Une exception `IndexOutOfBoundsException` est levée si la position n'est pas valide.

Note Parmi les exceptions levées par les méthodes de l'interface `UnsortedList`, `EmptyListException` est une nouvelle exception que nous avons créée (voir code fourni), au contraire de `IndexOutOfBoundsException` qui est définie dans le package `java.lang`.

Remarque sur la classe `MyUnsortedList`

- Le constructeur de la classe `MyUnsortedList` est privé et ne doit pas être utilisé directement. En remplacement la classe `MyUnsortedList` fournit une méthode statique `of()` qui prend en entrée un nombre variable d'arguments et retourne une nouvelle liste de type `MyUnsortedList`. Example d'utilisation :

  ```
  UnsortedList<Integer> integers = MyUnsortedList.of(1, 2, 3, 4);
  ```

  Ce pattern, nommé *Factory Methods*, est inspiré des bonnes pratiques de développement Java et on le retrouve dans les classes de la librairie standard depuis Java 9 (voir ici).
- La classe `MyUnsortedList` implémente aussi la méthode `equals()` permettant de comparer deux listes ainsi que la méthode `toString()` qui retourne une représentation textuel de la liste. Ces méthodes vous seront utiles pour la rédaction des tests.

JUnit sous Eclipse Pour créer votre suite de tests JUnit et exécuter votre suite de tests sous Eclipse, veuillez procéder comme suit :

1. Après avoir extrait l'archive fournie, importer le code dans votre répertoire de travail. (*Menu File→Import – General – Existing project into workspace*)
2. Dans votre répertoire `unsortedlist`, créer un nouveau *Source Folder* nommé `test` au même niveau que le répertoire `src` existant.
3. Dans votre nouveau répertoire `test`, demandez la création d'un nouveau *JUnit test case*. Dans la fenêtre de configuration :
   - Sélectionnez `New JUnit 4 test`
   - Sélectionnez votre nouveau répertoire `test` comme *Source folder*
   - Définissez le package comme : `datastruct`
   - Définissez le nom de la classe de test comme : `MyUnsortedListTest`
4. Validez, vous êtes maintenant prêt à écrire et exécuter des tests
5. Pour exécuter les tests, click droit sur la classe de test : *Run As – JUnit Test*

Écriture des tests Vous devez écrire une classe `MyUnsortedListTest` qui comportera un ensemble de cas de tests JUnit vérifiant que les différentes méthodes de l'implémentation qui vous est proposée respectent bien leur spécification.

Vous pourrez exécuter directement les cas de tests depuis votre IDE (e.g., Eclipse). Vous corrigerez les bugs éventuels que vous trouverez (le code fourni contient au moins 2 bugs).

4 Couverture de code

Nous vous proposons de vous intéresser à la couverture de code.

En quelques mots, la couverture de code évalue le taux de code source d'un programme qui est testé. Différentes métriques sont utilisées dans ce contexte. Une métrique classique est d'exprimer la couverture en pourcentage de lignes couvertes.

L'outil EclEmma (`http://eclemma.org`) est un plugin libre Eclipse permettant d'évaluer la couverture de code pour Java. Il supporte notamment le framework de tests JUnit. Nous aborderons la couverture de code plus en détails lors du prochain cours.

EclEmma est normalement disponible sur les machines de TP. Sinon, il est facilement installable depuis *Eclipse Marketplace*.

Reportez vous à la documentation en ligne pour apprendre à utiliser EclEmma, et évaluez le taux de couverture pour votre liste simplement chaînée. EclEmma offre différentes métriques de couverture. Prenez le temps d'observer ces différentes métriques.

5 Pour aller plus loin

Si vous avez fini les parties précédentes, nous vous proposons dans cette partie plusieurs directions possibles pour approfondir des sujets en lien avec les problématiques DevOps. Ces directions sont indépendantes et peuvent être traitées dans l'ordre que vous souhaité.

5.1 Développement piloté par les tests

Dans cet exercice, nous vous proposons d'appliquer les principes du *Test-Driven Development* (TDD) : `https://fr.wikipedia.org/wiki/Test_driven_development`).

Pour appliquer une approche TDD stricte, voici les trois lois que vous devez suivre :

1. Écrivez un test qui échoue avant d'écrire le code de production correspondant.
2. Écrivez une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
3. Écrivez le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.

Concrètement, cela implique de suivre les étapes suivantes au cours de plusieurs cycles :

1. Écrire un seul test qui décrit une partie du problème à résoudre
2. Vérifier que le test échoue, autrement dit qu'il est valide, c'est-à-dire que le code se rapportant à ce test n'existe pas
3. Écrire juste assez de code pour que le test réussisse
4. Vérifier que le test passe, ainsi que les autres tests existants
5. Remanier le code, c'est-à-dire l'améliorer sans en altérer le comportement, qu'il s'agisse du code de production ou du code de test.

Pour trouvez plus d'infos sur TDD, la page suivante est un bon point de départ : `https://tdd.mooc.fi/1-tdd`

Pour appliquer l'approche TDD, nous vous proposons d'implémenter le *Kata*[2] appelé *string calculator* et décrit ici : `https://github.com/mwhelan/Katas/tree/master/Katas.StringCalculator`.

D'autres Katas sont décrit à l'adresse suivante :
`https://www.michael-whelan.net/code-katas-for-practicing-tdd/`

5.2 Amélioration de l'implementation `MyUnsortedList`

Comme vous l'avez vu dans les questions précédentes, les test unitaires permettent de découvrir la présence de bugs dans une implémentation existante. Un autre avantage est qu'ils permettent de découvrir rapidement des bugs dits "régression" quand l'implémentation est modifiée à posteriori en introduisant des bugs qui n'existaient pas auparavant.

Ici, nous vous proposons de réaliser deux modifications sur l'implémentation de MyUnsortedList. Ces modifications sont des optimisations, ils ne modifient pas la spécification, seulement son fonctionnement

2. Exercice de programmation.

interne. Dès lors, si votre ensemble de tests unitaires est éxhaustif, vous ne devriez pas avoir besoin besoin d'écrire de nouveau tests et les tests existants devraient vous prévenir de l'introduction de nouveaux bugs. Néanmoins, il est recommandé de relancer une évaluation de la couverture de code une fois les modifications faites pour s'en assurer et de renforcer vos tests si nécessaire.

Ajout en fin de la liste en temps constant ($\mathcal{O}(1)$) Vous devez modifier la classe `MyUnsortedList` afin que les méthodes ajoutant en fin de liste (`append` et `insert`) opérent en temps constant indépendant de la taille de la liste.

Après modification de l'implémentation, vérifiez-en le bon fonctionnement en faisant tourner vos tests.

Suppression en fin de liste en temps constant ($\mathcal{O}(1)$) Poursuivez vos modifications de la classe `MyUnsortedList` pour que les opérations de suppression en fin de liste (`popLast` et `remove`) opérent en temps constant indépendant de la taille de la liste.

Comme pour la modification précédente, assurez-vous que vos tests couvrent et valident vos modifications.

5.3 Ant

`Ant` peut être présenté que *l'équivalent* de `Makefile` pour les projets écrits en Java.

`Ant` n'a pas été décrit en cours, mais les slides de présentation de `Ant` sont disponibles à la suite des slides de présentation de `Makefile` : `https://tropars.github.io/downloads/lectures/DevOps/devops_4_builders.pdf`

Après avoir étudier les slides de présentation de `Ant`, nous vous proposons d'écrire un fichier de directives Ant (`build.xml`), pour compiler et tester automatiquement votre code. Votre `build.xml` devra contenir les cibles suivantes :

init : Crée notamment le répertoire où sera stockés les fichiers exécutables (`.class`)

compile : Compile les fichiers sources

test : Lance les tests

clean : Supprime les fichiers et répertoires créés pour la compilation et les tests

Quelques points à prendre en compte pour l'écriture de votre `build.xml` :

- Pensez aux dépendances entre les cibles
- Définissez des *properties* pour que votre `build.xml` reste relativement générique.
- Pour pouvoir compiler et exécuter les tests, vous allez devoir ajouter `junit.jar` au *classpath*. Vous pouvez observer l'emplacement de `junit.jar` depuis Eclipse où il a été normalement automatiquement ajouté au *classpath*.

5.4 Une présentation sur la rédaction des tests unitaires

Si vous voulez en apprendre plus sur les bonnes pratiques en termes de tests unitaires, nous vous proposons de suivre l'exposé suivant (en anglais), qui présente très bien le sujet : `https://www.youtube.com/watch?v=fr1E9aVnBxw`

6 Pour aller encore plus loin

6.1 Mutation testing

Un sujet en lien avec celui des tests unitaires et de la couverture de code, est celui du *mutation testing*. Le *mutation testing* est une manière plus avancée d'évaluer la qualité d'une suite de tests.

Pour une introduction sur le sujet, vous pouvez regarder l'outil PIT : `http://pitest.org/`

6.2 Skip-list

Si vous avez fini en avance, et/ou vous voulez aller plus loin, nous vous suggérons de traiter le cas d'une structure de donnée plus complexe : une *skip-list*.

Une skip-list est une liste chaînée triée, qui contient des pointeurs supplémentaires *vers l'avant*, ajoutés aléatoirement, servant à accélérer la recherche d'un élément dans la liste. Pour comprendre le fonctionnement d'une skip-list vous pouvez partir de la description fournie sur wikipedia : `http://en.wikipedia.org/wiki/Skip_list`.

En quelques mots, les opérations sur une skip-list ont en moyenne la même complexité que celles d'un arbre équilibré (même si elles sont bien moins efficaces dans le pire cas) tout en étant plus simple à mettre en œuvre.

Essayez d'implémenter une skip-list et de concevoir les tests unitaires correspondants.