Polytechnique Montréal

**TP6-Log3430**

Travail de :

Xavier Brazeau (1854911)

Amine Kamal (1718831)

Maxime Bernier (1893115)

Remis à Mme Hiba Bagane

Jeudi 12 avril 2019

# Étapes précédant les tests

**Ajout aux classes**

**Dans ce travail pratique, nous devons rajouter un attribut type à la classe *Graph* ainsi que les méthodes *getter* et *setter*. Ensuite, nous avons utilisé la méthode *setter* dans la classe *GraphGenerator* pour le type selon la méthode. Subséquemment, nous avons ajouté une méthode à *BipartiteXExtended*. Notre choix s’est arrêté à getPartitonBlack(), qui retourne une des deux partitions constituant le graph bipartie. La couleur Black réfère au code de couleur utilisé dans le code précédemment**

**Tests à refaire**

**Étant donné que le code de différentes classes a été modifiées, nous devons nous interroger sur les tests à refaire. Pour commencer, nous nous référons au tableau suivant réalisé lors du dernier travail pratique:**

|  |  |
| --- | --- |
| **Classe étudiée** | **CFW de la classe étudiée** |
| BipartiteXExtended |  |
| BipartiteX | BipartiteXExtended |
| Graph | BipartiteXExtended  BipartiteX |
| In | BipartiteXExtended  BipartiteX  Graph |
| Queue | BipartiteXExtended  BipartiteX |
| Stack | BipartiteXExtended  BipartiteX  Graph |
| Bag | BipartiteXExtended  BipartiteX  Graph |

Tableau 1: CFW des classes étudiées

Trois constats concernant le tableau 1 doivent être mentionnés :

* *GraphGenerator* ne fait pas partie des classes concernées.
* *Graph* aura un impact sur deux classes : *BipartiteX* et *BipartiteXExtended*.
* Aucune classe sera affectée par la modification de *BipartiteXExtended*.

La prochaine étape consiste à réaliser les tests.

# Tests

**Critère d’adéquation**

Nous retenons le même cirtère d’adéquation que le dernier travail pratique, soit la couverture des branches. Ce choix fut arrêté à cause de sa simplicité. Nous utilisons JaCoCo pour l’évaluer. Il est à noter que les fonctions main sont ignorées.

**Queue & Bag & In & Stack**

Comme mentionné lors de l’analyse du tableau 1, ces classes ne sont pas affectées et ne nécessitent pas de nouveaux tests.

**Graph**

Les anciens tests furent modifiés pour avoir une vérification que l’attribut type reste *NULL* lors des tests. Pour la méthode *setter*, un test additionnel a été nécessaire. Le test consiste à générer un graphe avec un type indéfini pour ensuite le définir en s’assurant que la valeur est bien celle désirée.

**bipartiteX**

Les anciens tests furent modifiés pour avoir une vérification que l’attribut type reste *NULL* lors des tests.

**bipartiteXExtended**

Les anciens tests furent modifiés pour avoir une vérification que l’attribut type reste *NULL* lors des tests. De plus, la nouvelle méthode doit être testée. Nous l’avons à travers tous les tests à regardant que les partitions correspondaient à nos attentes. En effet, en créant les arcs nous-mêmes, nous connaissons les partitions.

# Questions

**Couverture des instructions :** Cette méthode de test consiste à passer par toutes les lignes de code du programme. Bien qu’elle permette de s’assurer que toutes les opérations fonctionnent et sont accessibles, cette méthode de test ne permet pas nécessairement de vérifier s’il peut y avoir des erreurs dans des cas différents que celui qu’on lui impose pour passer par toutes les instructions. Par exemple, peut-être qu’en passant un nombre entier positif, le programme passe par toute les branches, mais cela ne veut pas dire qu’il fonctionne parfaitement puisqu’il pourrait y avoir des erreurs en y passant une valeur à zéro.

**Couverture des branches :** La couverture des branches a des limitations similaires à la couverture des instructions mais est plus complète puisqu’en plus de couvrir toutes les instructions d’un programme, elle s’assure que tout le graphe de flot de contrôle d’un programme est parcouru.

**Couverture des conditions :** La couverture des conditions ajoute à la couverture des branches en s’assurant que les prédicats ont les bonnes sorties pour toutes les valeurs d’entrées possibles.

**Catégorie-partition :** Les tests catégories-partitions consistent à créer des catégories spécifiques pour chaque variable et de les associer à des contraintes rendant ces catégories valides ou invalides. Les catégories sont ensuite combinées selon trois critères : all-choice, each-choice et base choice. Un exemple de séparation en catégories d’une variable entière pourrait être par exemple ses valeurs negatives, sa valeur nulle et ses valeurs positives qui ont chacune des contraintes et des productions en sorties différentes.

**All-Choice :** Combinaison de toutes les catégories les unes avec les autres. Bien quie ce critère soit le plus complet, il devient également exponentiellement compliqué plus on y ajoute de catégories.

**Each-Choice :** Pour chaque catégorie, on y associe au moins une de ses valeurs dans chaque test. Contrairement à all-choice, ce critère est très facile à implémenter, mais il est également très facile de passer à côté de combinaisons de catégories pouvant faire ressortir des erreurs.

**Base-Choice :** Pour le base choice il suffit de choisir un cas de test de base et de changer une variable à la fois pour une catégorie différente. Ce type de test est un bon compromis entre all-choice et Each-choice.

**MADUM :** Le test MADUM consiste à faire des tests pour tester qu’un objet est bien construit. Il suffit de faire un tableau entre les attributs et les méthodes et de classer si les méthodes sont un transformer, un rapporteur, un constructeur ou autre pour l’attribut donné. Pour chaque t, c, r, o il faut appliquer une méthode de test pour vérifier si la méthode se comporte correctement. Étant une méthode de test en boite blanche, le test MADUM a plusieurs limitations, notamment, qu’elle est difficile à automatiser, qu’elle est basée sur le code et qu’elle n’étudie pas nécessairement les états d’une classe.

**Test d’intégration :** Les test d’intégration visent à nous expliquer quels objets tester dans quel ordre pour arriver à tester une classe en prenant compte de ses relations d’agrégation, d’association et d’héritage. Il y a plusieurs façons d’aborder les tests d’intégration :

*Intégration descendante*: Intégration testant du plus haut niveau (enfants) au plus bas niveau (parents) en procédant un niveau à la fois. Bien que ceci permet de tester une fonction en limitant les défauts, les défauts sont éliminés en utilisant de nombreux stubs puisque les classes intermédiaires (classes à la fois parent et enfant) ne peuvent être générés si leurs enfants n’ont pas étés implémentés.

*Intégration ascendante* : De bas en haut. Bien que le programme soit testé séquentiellement en ayant recours à un nombre minimum de stubs, ce type de test peut recourir à un nombre important de tests pour arriver à tester la classe mère puisque chaque enfant doit être implémenté et testé. Les défauts des niveaux supérieurs héritent également des défauts des niveaux inférieurs.