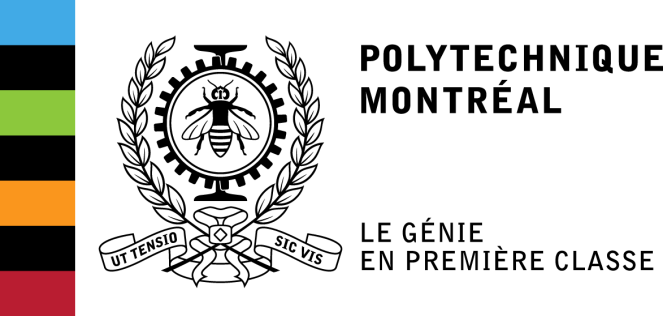
****

**FIRSTNAME LASTNAME MATRICULE**

**David TREMBLAY 1748125**

**Travail pratique #1**

**Introduction aux tests unitaires**

**Présenté à Rodrigo Morales**

**LOG3430- Méthode de test et de validation du logiciel**

**Groupe 03**

**Département de génie informatique et génie logiciel**

**Le 25 septembre 2017**

**École polytechnique de Montréal**

# Conception des tests unitaires

Pour réaliser nos tests unitaires sur la classe GraphGenerator.java, nous avons dû tout d’abord importer les classes Graph ainsi que GraphGenerator. Une classe de test a ensuite dû être crée et les annotations @BeforeClass et @AfterClass ont été ajoutée pour respectivement préparer les tests et nettoyer après les tests.

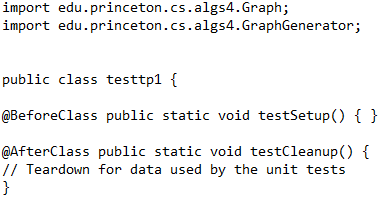


Figure 1: Initialisation de la classe de tests

Par la suite, nous avons réalisé nos tests graphe par graphe en suivant l’ordre de la classe GraphGenerator. Pour chaque graphe, les entrées invalides ont d’abord été testées suivi d’une combinaison d’entrées valides pour tester que les graphes étaient bien créés. Les entrées valides ont aussi été testées pour les valeurs limites puisque c’est à la frontière qu’on remarque le plus d’erreur habituellement. Pour tester les valeurs invalides, nous avons utilisé le mot clé « expected » avec la classe IllegalArgumentException pour indiquer à J-UNIT que le test doit donner une erreur et nous avons passé l’entrée qui génère cette erreur en paramètre pour la création du graphe (un nombre de sommets négatif par exemple).

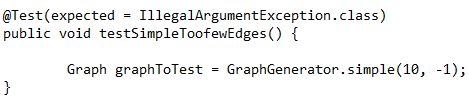


Figure 2:Conception des tests pour les entrées invalides

Pour vérifier le bon fonctionnement de la création de graphe avec des entrées valides, nous avons utilisé la méthode assertEquals() qui permet d’afficher un message pour faciliter la compréhension du test et qui compare un résultat avec la valeur à laquelle elle devrait être. Nous utilisons donc cette fonction pour s’assurer que le nombre de sommets, le nombres d’arcs (lorsque possible de savoir le nombre qu’on devrait obtenir) et le type de graphe (avec son typeName) pour s’assurer que le graphe a été créer correctement. Nous pouvons obtenir ces trois valeurs à l’aide des méthodes V(), E() et getTypeName(). Nous avons utilisé des messages explicites qui indique clairement à quoi le test sert. Les valeurs limites pour les sommets et les arcs ont aussi été testées pour être plus rigoureux dans nos tests et s’assurer qu’aucune erreur de se produit dans les frontières.

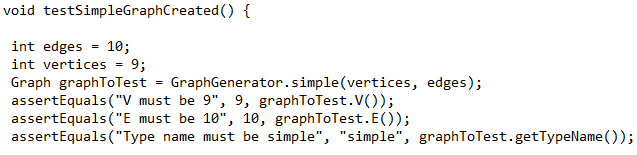


Figure 3: Conception des tests pour les entrées valides

Lorsque les graphes pouvaient être créée à l’aide d’une probabilité, des tests de probabilités invalides ainsi que de création de graphe de cette façon ont aussi été créé.

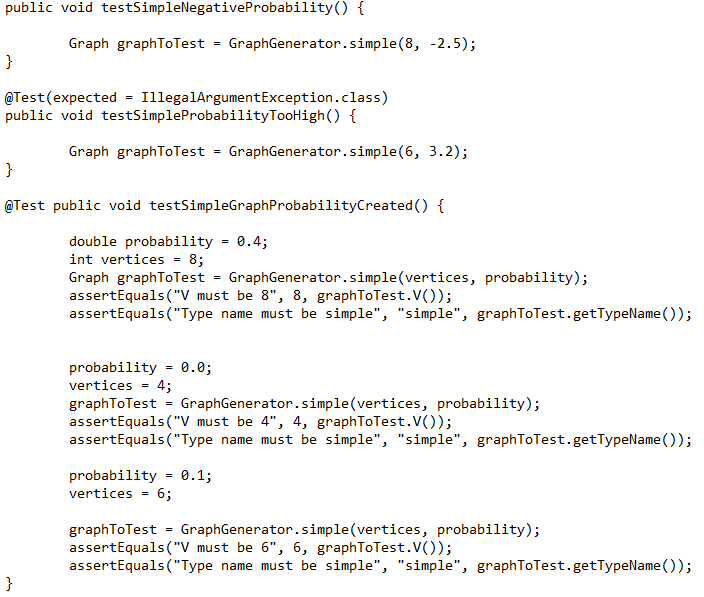


Figure 4: Conception de tests pour des graphes avec probabilités

# Énumération des tests

Nous avons utilisé des noms de tests explicites pour faciliter la compréhension du code. Ces tests sont énumérés ici mais analysés dans la section suivante.

## Type de graphe :

**simple** : testSimpleTooManyEdges(), testSimpleToofewEdges(), testSimpleGraphCreated(), testSimpleNegativeProbability(), testSimpleProbabilityTooHigh(), testSimpleGraphProbabilityCreated(), testSimpleInvalideInput()

**complete :** testCompleteGraphCreated(), testCompeteInvalideInput()

**completeBipartite :** testCompleteBipartiteCreated(), testCompleteBipartiteInvalideInput()

**bipartite** : testBipartiteTooManyEdges(), testBipartiteToofewEdges(), testBipartiteCreated(), testBipartiteNegativeProbability(), testBipartiteTooHighProbability(), testBipartiteWithProbabilityCreated(), testBipartiteInvalideInput()

**path**: testPathCreated(), testPathnvalideInput()

**binaryTree**: testBinaryTreeCreated(), testBinaryTreeInvalideInput()

**cycle** : testCycleCreated(), testCycleInvalideInput()

**eulerianCycle** : testEulerianCycleNegativeEdges(), testEulerianCycleNegativeVertices(), testEulerianCycleCreated(), testEulerianCycleInvalideInput()

**eulerianPath** : testEulerianPathNegativeEdges(), testEulerianPathNegativeVertices(), testEulerianPathCreated(), testPathInvalideInput()

**wheel** : testWheelTooFewVertices(), testWheelCreated(), testWheelInvalideInput

**star** : testStarTooFewVertices(), testStarCreated(), testStarInvalideInput

**regular** : testRegularVerticesDegreeEven(), testRegularCreated(), testRegularInvalideInput

**tree** : testTreeCreated(), testTreenvalideInput

# Analyse des tests

Bien qu’il n’aille pas de bloc throw/catch dans le cas d’un nombre de sommets invalides. La logique nous indique que c’est impossible. Tous les types de graphes qui ont été testés dans ce laboratoire ont donc été testés avec un nombre de sommets invalides pour s’assurer que ces erreurs étaient testées. Les cas de valeurs limites pour les cas valides ont aussi été testés pour chaque méthode afin d’avoir des tests plus rigoureux et parce que la frontière est un endroit propice aux erreurs. Pour tester les entrées valides, une entrée aléatoire valide a aussi été testée et des assertEquals ont permis de s’assurer que les sommets, arcs et type correspondait à la valeur attendue. Les méthodes nécessitant des tests particuliers sont détaillées ci-dessous et une analyse récapitulative sert de clôture à ce premier laboratoire par la suite.

### simple(int v, int E)

En analysant la méthode simple de GraphGenerator, on constate que les arcs doivent être >0 et que le nombre d’arcs doit être inférieur à V\*(V-1)/2. Pour tester cette classe, nous avons donc tester le cas où le nombre d’arcs est négatif ou lorsque le nombre est trop grand pour le nombre de sommets. Pour ce dernier test, nous avons utilisé 5 sommets et donc 5\*(5-1) /2 = 10 avec 122 arcs, ce qui est beaucoup trop d’arcs (on se rappelle qu’un graphe simple ne peut pas contenir de liens doubles ni de boucles).

### simple(int v, double p)

La probabilité devant être entre 0 et 1, deux cas de tests pour des entrées < 0 ou > 1 ont été testées. Les tests pour les entrées valides sont similaires à la méthode testée précédemment (tests sur les sommets et le type pour les cas limites et un cas aléatoire valide).

### complete(v)

Les tests pour la méthode complete() sont identique à un graphe simple mais nous nous assurons que le nombre d’arcs correspond à v\*(v-1)/2 qui correspond au nombre maximal d’arcs pour qu’un graphe soit à la fois simple et complet.

### bipartite(int V1, int V2, int E) et bipartite(int V1, int V2, double p)

L’analyse de cette méthode permet de constater que le paramètre E doit être supérieur à 0 tout en étant inférieur à V1\*V2. Nous avons testé ce dernier cas avec les paramètres 6, 10 et 80 respectivement pour s’assurer qu’une erreur IllegalArgumentException allait être lancée (80 > 6\*10). Pour le cas valide, nous pouvons constater que le nombre de sommet du graphe biparti doit être V1 + V2 et ce cas de test a été testé à l’aide des paramètres 12 et 10 et la ligne « assertEquals("V must be 22", 22, graphToTest.V()); ». Le reste des tests sont comme à l’habitude et décrit au début de cette section.

### cycle(int V)

Un test supplémentaire a été ajouté pour ce type de graphe. En effet, on sait qu’un cycle contient autant de sommet que d’arcs. Ce cas de test a donc été ajouté pour s’assurer que le nombre de sommets passé en paramètres correspond au nombre d’arcs.

### regular(int V, int k)

Une des cas invalide pour ce type de graphe est quand le nombre de sommets \* la constant k ne donne pas un chiffre pair. Nous avons donc un test testRegularVerticesDegreeEven() avec les paramètres 3 et 1 (3\*1 = 3 qui est impair) pour s’assurer que ce test retourne une erreur.

### completeBipartite(int V1, int V2), path(int V), binaryTree(int V), eulerianCycle(int V, int E), eulerianPath(int V, int E), wheel(int V), star(int V), tree(int V)

Les tests pour toutes ces méthodes sont identiques aux tests standards décrit au début de la section ci-haut.

En analysant notre méthode de conception ainsi que nos tests, nous pouvons affirmer que notre confiance sur le bon fonctionnement du programme est élevée. En effet, tous les cas invalides se trouvant dans la classe GraphGenerator.java ont été testés. De plus, nous avons ajouté un cas de test pour des sommets invalides pour chaque méthode puisqu’elle nous semblait pertinente. Les tests pour les entrées valides testent tous les paramètres de sortie possible (sommets, arcs et type) et sont répétés pour les valeurs limites pour encore plus de rigueur. On constate que tous les tests sont réalisés avec succès et qu’ils sont exhaustif, d’où notre confiance dans le bon fonctionnement du programme.