

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

Groupe 03

**Travail pratique #3**

**Test boîte blanche d'une application**

**Présenté à Rodrigo MORALES**

**David TREMBLAY 1748125**

**Taleb SOULI 1770491**

Département de génie informatique et génie logiciel

Le 30 octobre 2017

École polytechnique de Montréal

**Table des matières**

[Retour sur les deux premiers laboratoires 3](#_Toc496655646)

[Conception des tests boîte blanche 4](#_Toc496655647)

[Énumération des tests 5](#_Toc496655648)

[Type de graphe 5](#_Toc496655649)

[Analyse des tests 5](#_Toc496655650)

[simple(int v, int E) 6](#_Toc496655651)

[simple(int v, double p) 7](#_Toc496655652)

[complete(v) 7](#_Toc496655653)

[bipartite(int V1, int V2, int E) et bipartite(int V1, int V2, double p) 7](#_Toc496655654)

[cycle(int V) 7](#_Toc496655655)

[regular(int V, int k) 7](#_Toc496655656)

[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) 8](#_Toc496655657)

# Retour sur les deux premiers laboratoires

Lors du premier laboratoire, des tests unitaires ont été développés sur la classe GraphGenerator.java fournie sur le site Moodle du cours. Pour se faire, tous les types de graphes pouvant être possiblement créés ont été analysés afin de développer ces tests. Les tests unitaires ont servi à vérifier le bon fonctionnement de certaines parties des méthodes de la classe. Par exemple, les cas renvoyant à des exceptions de type IlligalArgumentException.class ont été testés pour s'assurer qu'une exception du bon type était renvoyée. Les cas de tests pour les entrées fonctionnelles ont aussi été testés pour s'assurer que le graphe contenait les bons paramètres (nom ainsi que nombre de sommets et arcs si possible). À la fin de ce laboratoire, nos tests passaient tous et permettaient d'avoir un plus grand niveau de confiance sur le fonctionnement général des méthodes de la classe. Ce TP était une initiation aux tests unitaires JUnit et nous a permis d'apprendre comment concevoir des tests de façon efficace et concise.

Pour chaque graphe, nous avons, d’abord,  testé les entrées invalides suivies 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’erreurs habituellement. Nous avons réussi relever différents types d’exceptions,**IllegalArgumentException**, **ArrayIndexOutOfBoundsException** et **NegativeArraySizeException,** lors des tests avec des valeurs invalides, 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).

Pour vérifier le bon fonctionnement de la création de graphes 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 nous assurer que le nombre de sommets, le nombre d’arcs (lorsque possible de savoir le nombre qu’on devrait obtenir) et le type de graphe (avec son typeName) pour nous assurer que le graphe a été créé correctement. Nous pouvons obtenir ces trois valeurs à l’aide des méthodes V(), E() et getTypeName(). Nous avons utilisé des messages explicites qui indiquent clairement à quoi le test sert. Les valeurs limites pour les sommets et les arcs ont aussi été testées pour être

Bien qu’il n’aille pas de bloc try/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 types correspondaient 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.

Par suite, des tests boîte noire ont été développés pour trois méthodes de la classe GraphGenerator lors du deuxième laboratoire. Les tests boîte noire visaient à tester sans se soucier de l'implémentation et seulement en se basant sur les spécifications (qui était de créer un certain type de graphe dans le cas présent). De plus, nos cas de tests ont été testés à l'aide des méthodes **Each Choice**  et **All Choice** pour encore plus de rigueur et pour s'assurer qu'une certaine combinaison de valeur ne produisait pas une sortie inattendue. Les tests développés étaient tous concluants et les spécifications semblent donc être respectées.

# Conception des tests boîte blanche

Cette fois-ci, nous testons la classe GraphGenerator en entier à l'aide de tests boîte blanche, c'est-à-dire, en testant en fonction de l'implémentation des fonctions. Nous utilisons l'outil d'analyse de couverture Jacoco qui permet de tester les lignes et les branches de notre code. Le but de ce laboratoire est de réaliser des tests de manière à obtenir une couverture totale de toutes les méthodes.

Pour se faire, nous devions analyser chaque branche des méthodes et être sûres de toutes les parcourir. Par exemple, dans le cas d'un **if**, les conditions **true** et **false** devaient être testées pour tester les deux branches. Cette procédure a dû être réalisée pour l'ensemble des méthodes de la classe à l'étude. Ceci n'a pas été très ardu puisque nos tests lors du TP1 et TP2 avaient été bien réalisés et couvraient déjà une grande partie des branches et des lignes. Nous avons tout d'abord commencé par exécuter un rapport Jacoco intermédiaire avec le code du TP2 pour voir la couverture obtenue, puis avons réalisé cette étape avec les tests du TP1 et TP2. Les méthodes n'étant pas totalement couvertes ont alors été examinées pour examiner les parties de code non testées pour remédier à la situation. Cette étape a ensuite été refaite jusqu'à obtention d'une couverture complète sur la classe en entier.

# Énumération des tests

Nous avons utilisé des noms de tests explicites pour faciliter la compréhension du code. Ces tests sont énumérés ci-dessous, mais analysés dans la section qui suit.

## Type de graphe

**Simple :**testSimpleTooManyEdges(), testSimpleTooFewEdges(), testSimpleTooFewVertices(), testSimpleGraphCreated(), testSimpleGraphLimits(), testSimpleNegativeProbability(), testSimpleProbabilityTooHigh(), testSimpleGraphProbabilityCreated(), testSimpleGraphProbabilityNegativeVertices(), testSimpleGraphProbabilityLimites().

**Complete :** testCompleteGraphCreated(), testCompleteGraphLimits(), testCompleteGraphNegativeVertices().

**CompleteBipartite:** testCompleteBipartiteCreated(), testCompleteBipartiteLimits(), testCompleteBipartiteNegativeVertices().

**Bipartite:**testBipartiteTooManyEdges(), testBipartiteTooFewEdges(), testBipartiteCreated(), testBipartiteLimits(), testBipartiteNegativeVertices(), testBipartiteNegativeProbability(), testBipartiteTooHighProbability(), testBipartiteWithProbabilityCreated(), testBipartiteWithProbabilityLimits(), testBipartiteWithProbabilityNegativeVertices().

**Path:**testPathCreated(), testPathLimits(), testPathNegativeVertices().

**BinaryTree:**testBinaryTreeCreated(), testBinaryTreeLimits(), testBinaryTreeNegativeVertices().

**Cycle:**testCycleCreated(), testCycleLimits(), testCycleNullVertices(), testCycleNegativeVertices().

**EulerianCycle:**testEulerianCycleNegativeEdges(), testEulerianCycleNegativeVertices(), testEulerianCycleNull(), testEulerianCycleCreated(), testEulerianCycleLimits().

**EulerianPath :**testEulerianPathNegativeEdges(), testEulerianPathNegativeVertices(), testEulerianPathCreated(), testEulerianPathLimit().

**Wheel** **:**testWheelTooFewVertices(), testWheelCreated(), testWheelLimits().

**Star** **:**testStarTooFewVertices(), testStarCreated(), testStarLimits().

**Regular :**testRegularVerticesDegreeEven(), testRegularCreated(), testRegularNegativeVertices(), testRegularNegativeDegree(), testRegularLimits().

**Tree** **:** testTreeCreated(), testTreeLimits(), testTreeNegativeVertices(), testTreeNullVertices().

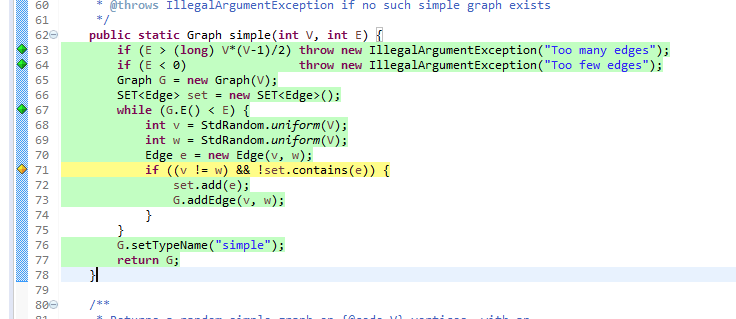
# Analyse des tests

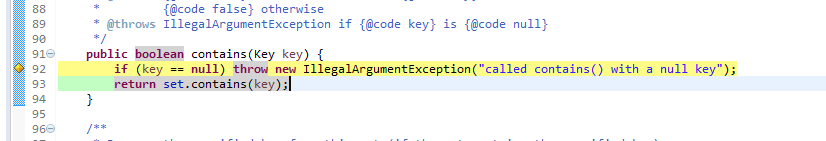
Notre approche se base sur des tests invalides suivis d’une combinaison des tests valides pour garantir le passage par toutes les branches d’une même méthode, ainsi on aboutit à 100% de couverture pour chaque méthode. Les entrées valides ont aussi été testées pour les valeurs limites. Nous avons réussi à relever différents types d’exceptions,**IllegalArgumentException**,**ArrayIndexOutOfBoundsException**et **NegativeArraySizeException**. En suivant cette approche de passer par tous les chemins possibles, tous les types de graphes ont été testés avec un nombre de sommets invalides pour s’assurer que ces erreurs étaient testées, puis avec des cas de valeurs valides qui prennent en compte le cas limite 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 types correspondaient à la valeur attendue. Ainsi nous avons abouti à 100% des couvertures de tests.

## 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 testé le cas où le nombre d’arcs est négatif ou lorsque le nombre est trop grand pour le nombre de sommets. Puis nous avons fait des tests avec des valeurs valides pour passer de tous les chemins possibles.

**Note :** pour le graphe simple, nous avons eu 100% pour instructions manquées et 90% pour branches manquées; en fait cette méthode utilise des valeurs random, donc nous avons utilisé la méthode SetSeed pour fixer la valeur random. Cette approche n’était pas suffisante, car pour passer par ce chemin **if((v!=w)&&!set.contains(e))**, il fallait changer la methode contains pour qu’elle retourne **false**, ainsi on peut passer par ce chemin et avoir 100% de test. En revanche la méthode est conçue de sorte que **set.contains(e)**retourne toujours **vrai**, d’où il nous est impossible d’avoir 100% de couverture.





## simple(int v, double p)

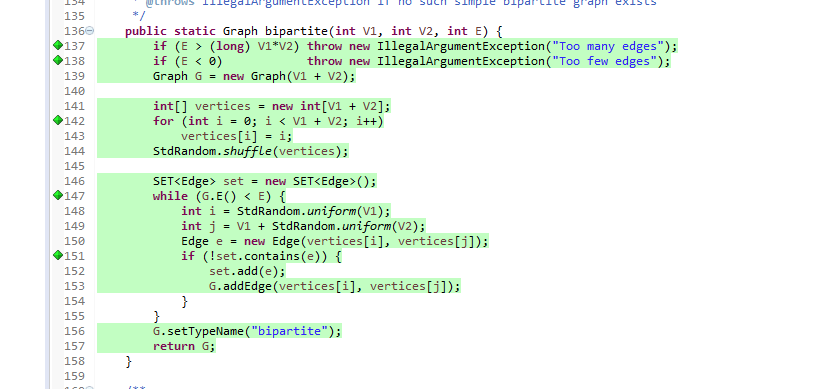
La probabilité devant être entre 0 et 1, deux cas de tests pour des entrées < 0 ou > 1 ont été testées. Les tests pour les entrées valides sont similaires à la méthode testée précédemment. Ce qui fait que nos tests ont couvert toutes les possibilités.

## complete(v)

Les tests pour la méthode complete() sont identiques 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 pour nous assurer qu’une erreur **IllegalArgumentException** allait être lancée. Pour le cas valide, nous pouvons constater que le nombre de sommets du graphe biparti doit être V1 + V2, donc on a choisi un cas de test pour satisfaire à ce cas. En plus de ces tests-là, on a fait les tests des cas valides pour arriver à 100% de couverture.

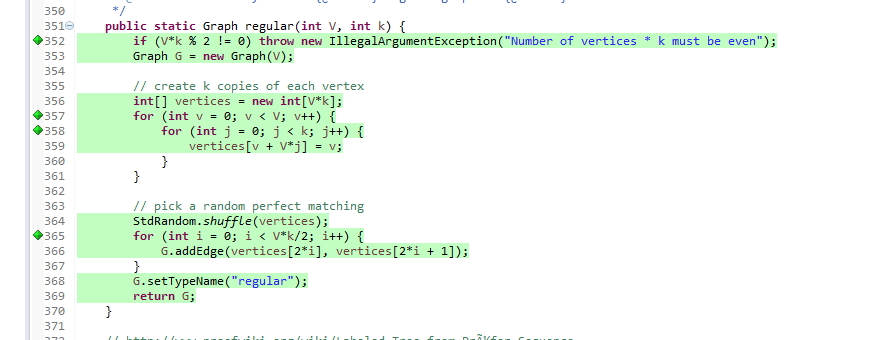


## cycle(int V)

Un test supplémentaire a été ajouté pour ce type de graphe. En effet, on sait qu’un cycle contient autant de sommets que d’arcs. Ce cas de test a donc été ajouté pour s’assurer qu’on passe par tous les cas possibles et avoir une couverture complète.

## regular(int V, int k)

Un des cas invalides pour ce type de graphe est quand le nombre de sommets \* la constant k ne donne pas un chiffre pair. Nous avons donc fait un test testRegularVerticesDegreeEven()pour nous assurer que ce test retourne une erreur, puis on a pris des valeurs valides pour accomplir tous les chemins possibles.



## 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 décrits au début de la section ci-haut. 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’elles permettaient d’augmenter la couverture des tests. Les tests des entrées valides couvrent tous les paramètres de sortie possible (sommets, arcs et type) et sont répétés pour les valeurs limites pour englober toutes les branches de chaque méthode et avoir plus de couvertures.