# **Tâche à réaliser**

Dans le cadre du cours LOG1000, vous avez notamment vu le fonctionnement des tests unitaires. Vous avez travaillé avec l’outil *cppunit* pour créer vos tests unitaires, les maintenir et les exécuter. L’équipe académique a donc retenu cet outil pour le projet intégrateur INF2990.

La création et l’exécution de cas de test constituent une très bonne pratique du génie logiciel. Ainsi, pour intégrer le concept des tests unitaires à votre projet, il est demandé de créer des suites de tests qui seront exécutés lors de la correction du deuxième livrable. Il est également demandé que tous les tests s’exécutent avec succès. En vous référant au document « Aide au développement », vous trouverez plus d’informations sur la façon dont nous avons intégré *cppunit* au cadriciel de base du projet et sur la manière de générer vos suites de cas de tests.

Bien évidemment, il ne serait pas réaliste de vous demander de tester exhaustivement chacune des classes de votre projet. Dans la pratique, il est rare que des tests aussi exhaustifs soient demandés puisque cela demanderait trop de ressources. Une de vos tâches en tant que futur(e) ingénieur(e) logiciel consiste à identifier quelles classes il serait davantage pertinent de tester au moyen de tests unitaires. Sans donner de règle universelle, généralement, les tests les plus intéressants sont ceux qui visent à tester :

· les fonctions effectuant un calcul quelconque et retournant une valeur;

· les algorithmes;

· les structures de données;

· les conditions booléennes.

Nous vous demandons de créer des suites de cas de test pour couvrir un minimum de 6 classes. Idéalement, la plupart des méthodes de ces classes devraient être testées. À cette fin, nous demandons que vous ayez des cas de test pour couvrir un minimum de 20 méthodes, réparties à votre choix parmi les 6 classes à tester. Notez que vous n’avez pas l’obligation de tester uniquement du code que vous avez écrit; vous pouvez également tester du code faisant partie à la base du cadriciel, tel que l’arborescence des nœuds.

Vous devez remplir un tableau pour chaque classe testée. En complément à la vérification de vos tableaux, le correcteur jettera un coup d’œil à vos suites de cas de test lors de l’évaluation du livrable.

Notez que les tests unitaires fournis en exemple avec le cadriciel (classe *NoeudAbstraitTest* par exemple) ne sont pas comptabilisés pour la couverture des tests.

# **Grille de correction**

L’évaluation des tests logiciels vaut 5% du cours. Le correcteur appliquera la grille de correction détaillée ci-dessous. Cette grille tient compte des deux éléments suivants :

· la **pertinence** et la **justification** des cas de test choisis parmi les différentes suites de cas de test, ainsi que les tableaux remplis;

· la **qualité** de l’implémentation des cas de test.

## **Choix des cas de test**

|  |  |
| --- | --- |
| **2,5 pts** | Les cas de test choisis sont **très pertinents**.  **ET**  La justification est **très claire et complète**. |
| **2 pts** | Les cas de test choisis sont **pertinents** ou **très pertinents**.  **ET**  La justification est **généralement claire** et **complète**. |
| **1,5 pt** | Les cas de test choisis sont **pertinents**.  **ET**  La justification est **moyennement claire** et/ou **incomplète**. |
| **1 pt** | Les cas de test choisis sont **peu pertinents**.  **ET/OU**  La justification est **floue** ou **incomplète**. |
| **0 pt** | Les cas de test choisis ne sont **pas pertinents** ou sont **manquants**.  **ET/OU**  La justification est **très floue**, **incomplète** ou **absente**. |

## **Implémentation des cas de test**

|  |  |
| --- | --- |
| **2,5 pts** | La qualité de l’implémentation des cas de test est **excellente**. |
| **2 pts** | La qualité de l’implémentation des cas de test est **très bonne**. |
| **1,5 pt** | La qualité de l’implémentation des cas de test est **bonne**. |
| **1 pt** | La qualité de l'implémentation des cas de test est **médiocre**. |
| **0 pt** | La qualité de l'implémentation des cas de test est **nettement insuffisante**. |

# **Travail à effectuer**

Pour chacune des classes que vous avez décidé de tester, vous devez :

· identifier la classe testée (et le nom de la classe contenant la suite de cas de test);

· identifier la branche appropriée;

· justifier la pertinence de tester cette classe.

Pour chacune des méthodes que vous avez décidé de tester, vous devez :

· identifier la méthode testée (et le nom de la méthode du cas de test);

· justifier la pertinence de tester cette méthode;

· expliquer ce qu’effectue le cas de test.

Vous pouvez utiliser les tableaux suivant pour entrer les informations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #** | | | |
| **Classe testée** |  | **Branche** |  |
|  | | | |
| **Justification** | | | |
|  | | | |

|  |  |
| --- | --- |
| **Cas de test #** | |
| **Méthode testée** |  |
|  | |
| **Justification** | |
|  | |
|  | |
| **Explication du cas de test** | |
|  | |
|  | |

# **Exemple**

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test # 1** | | | |
| **Classe testée** | NoeudAbstrait (NoeudAbstraitTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| Nous testons ici un nœud non-composite*.* Il est intéressant de tester cette classe puisque nous pouvons vérifier le bon fonctionnement des méthodes à la base de l’arbre de rendu. Nous testons notamment certaines conditions booléennes et nous assurerons qu’aucun nœud « enfant » ne peut être ajouté.  Puisque NoeudAbstrait est une classe abstraite, nous utilisons la classe NoeudConeCube, puisque celle-ci dérive directement de NoeudAbstrait. | | | |

|  |  |
| --- | --- |
| **Cas de test # 1** | |
| **Méthode testée** | NoeudAbstrait::assignerPositionRelative (testPositionRelative) |
|  | |
| **Justification** | |
| Comme la position relative d’un objet est utilisée dans plusieurs contextes (rendu à l’écran, édition des objets, collisions, etc.), il est primordial de s’assurer que la méthode fonctionne correctement. | |
|  | |
| **Explication du cas de test** | |
| On s’assure que le vecteur contenant la position relative est initialement un vecteur nul. Ensuite, on assigne une position relative non nulle au nœud et on s’assurer que la position relative a été modifiée de la même façon. | |
|  | |
| **Cas de test # 2** | |
| **Méthode testée** | NoeudAbstrait::obtenirType (testType) |
|  | |
| **Justification** | |
| Le *type* d’un nœud est caractérisé par une chaîne de caractère. Comme cette dernière est également utilisée comme identifiant pour les usines, il est préférable de s’assurer que le nœud a été créé avec le bon type. | |
|  | |
| **Explication du cas de test** | |
| On s’assurer que le nœud créé pour le test comporte le bon type, c’est-à-dire que son type est celui attendu et ne correspond pas aux types des autres objets. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test # 3** | |
| **Méthode testée** | NoeudAbstrait::estSelectionne (testSelection) |
|  | |
| **Justification** | |
| La sélection d’un nœud est à la base même de l’édition de la zone de jeu. Il est donc pertinent de tester son fonctionnement. | |
|  | |
| **Explication du cas de test** | |
| Un nœud peut être sélectionné seulement s’il est sélectionnable. Il s’agit d’un état booléen qui dépend aussi d’un autre état. Nous testons chaque paire conditionnelle possible (vrai-vrai, vrai-faux, faux-vrai, faux-faux) pour nous assurer que les états sont cohérents avec ce qui est attendu. | |
|  | |
| **Cas de test # 4** | |
| **Méthode testée** | NoeudAbstrait::obtenirNombreEnfants (testEnfants) |
|  | |
| **Justification** | |
| Le patron Composite permet de manier des nœuds feuilles et des nœuds branches à l’aide d’une interface commune. Pour un nœud feuille, il ne devrait pas être possible d’ajouter des nœuds enfants. Ce cas de test s’assurer que c’est bel et bien le cas. | |
|  | |
| **Explication du cas de test** | |
| On s’assure initialement que le nœud ne contient aucun enfant. Par la suite, on tente d’ajouter un enfant et on s’assure que celui-ci n’a pas été ajouté. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test #5** | |
| **Méthode testée** | NoeudAbstrait::assignerEstEnregistrable (testEnregistrable) |
|  | |
| **Justification** | |
| Il est très pertinent de tester cette fonction, puisque cette fonction permet d'assigner l'état d'être enregistrable ou non d’un noeud. De plus, la fonction estEnregistrable() est aussi testée en même temps. En effet, celles-ci vont servir dans nos fonctions d’enregistrement en xml. Il est important de savoir s’il est enregistrable ou non pour être capable de s’adapter aux deux conditions possibles. Nos fonctions de sauvegarde sont servies tout au long du projet, surtout dans l’état édition. | |
|  | |
| **Explication du cas de test** | |
| On assigne une valeur booléenne au noeud à l’aide de assignerEstEnregistrable(bool), et nous vérifions s’il est enregistrable grâce à la fonction estEnregistrable(). Deux cas sont possibles, dont le cas non enregistrable et le cas enregistrable. | |
|  | |

# **Suite de cas de test**

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #2** | | | |
| **Classe testée** | VisiteurDeplacement (VisiteurDeplacementTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| Assigner une position de déplacement à l'aide du visiteur déplacement.  Ce test permet de tester si le visiteur déplacement fonctionne bien. Il est utile  de le tester, puisque l'édition d'une zone de jeu est une partie importante du  livrable 1. Pour éditer, le déplacement d'objet est une partie très importante.  De plus, il est aussi possible de vérifier en même temps si la fonction  accepter(VisiteurAbstrait\* visiteur) est fonctionnelle. La fonction accepter(...)  de chaque noeud est primordiale, car elle permet d'accéder à tous nos visiteurs. | | | |

|  |  |
| --- | --- |
| **Cas de test #1** | |
| **Méthode testée** | VisiteurDeplacement::visiter (testDeplacement) |
|  | |
| **Justification** | |
| Il est utile de le tester, puisque l'édition d'une zone de jeu est une partie importante du  livrable 1. Le but est de pouvoir concevoir notre propre zone de jeu au départ. Il doit donc être possible de déplacer un objet sur la scène d’édition. | |
|  | |
| **Explication du cas de test** | |
| Nous avons tout d’abord vérifié que la position initiale est bien un vecteur nu à l’aide de EGAL\_ZERO défini dans utilitairel. Par la suite, on modifie la position de l’objet. Dans notre cas, nous avons décidé d’assigner la position (5.5, 5.5, 5.5) par hasard. Ensuite, on vérifie que la position a bel et bien été modifiée en vérifiant si la soustraction entre sa nouvelle position, qui devrait être 5.5, 5.5, 5.5, et les valeurs(5.5, 5.5, 5.5) est égal à 0. | |
|  | |

# 

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #3** | | | |
| **Classe testée** | VisiteurMiseEchelle  (VisiteurMiseAEchelleTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| Agrandir ou diminuer un objet à l'aide du visiteur mise à l'échelle.  Ce test permet de tester si le visiteur mise à l'échelle fonctionne bien. Il est utile  de le tester, puisque l'édition d'une zone de jeu est une partie importante du  livrable 1. Pour éditer, une mise à échelle d'un objet est une partie très importante.  De plus, il est aussi possible de vérifier en même temps si la fonction  accepter(VisiteurAbstrait\* visiteur) est fonctionnelle. La fonction accepter(...)  de chaque noeud est primordiale, car elle permet d'accéder à tous nos visiteurs. | | | |

|  |  |
| --- | --- |
| **Cas de test #1** | |
| **Méthode testée** | VisiteurMiseEchelle::visiter (testMiseAEchelle) |
|  | |
| **Justification** | |
| Il est utile de le tester, puisque l'édition d'une zone de jeu est une partie importante du  livrable 1. Le but est de pouvoir concevoir notre propre zone de jeu au départ. Il doit donc être possible d’agrandir et de diminuer la dimension d’un objet sur la scène d’édition. | |
|  | |
| **Explication du cas de test** | |
| Nous avons vérifié si le rayon est plus grand que son rayon initial que nous avons donné au départ qui est par hasard de 1, à l’aide de setRayonBonus(double), lorsque nous forçons un facteur d’échelle, dont 2, une autre valeur arbitraire, avec setFacteur(double). | |
|  | |

# 

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #4** | | | |
| **Classe testée** | VisiteurRotation  (VisiteurRotationTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| Faire tourner un objet dans le sens horaire ou anti-horaire d'une montre  à l'aide du visiteur roation.  Ce test permet de tester si le visiteur rotation fonctionne bien. Il est utile  de le tester, puisque l'édition d'une zone de jeu est une partie importante du  livrable 1. Pour éditer, la rotation d'un objet est une partie très importante.  De plus, il est aussi possible de vérifier en même temps si la fonction  accepter(VisiteurAbstrait\* visiteur) est fonctionnelle. La fonction accepter(...)  de chaque noeud est primordiale, car elle permet d'accéder à tous nos visiteurs. | | | |

|  |  |
| --- | --- |
| **Cas de test #1** | |
| **Méthode testée** | VisiteurRotation::visiter (testRotation) |
|  | |
| **Justification** | |
| Il est utile de le tester, puisque l'édition d'une zone de jeu est une partie importante du  livrable 1. Le but est de pouvoir concevoir notre propre zone de jeu au départ. Il doit donc être possible de tourner un objet sur la scène d’édition. | |
|  | |
| **Explication du cas de test** | |
| Nous vérifions si l’angle de rotation est nul au départ à la création de l’objet. Par la suite, nous vérifions si l’angle de rotation est plus grand que son angle de départ lorsque nous donnons un delta de 0.1. C’est-à-dire lorsqu’il tourne dans le sens horaire d’un montre à une indentation de 0.1. Encore une fois, nous avons décidé de façon arbitraire que l’objet allait tourner dans le sens positif. | |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #5** | | | |
| **Classe testée** | ProjectionOrtho | **Branche** | Master |
|  | | | |
| **Justification** | | | |
| la classe projectionOrtho est celle qui implémente plusieurs aspects de la vue de notre zone de jeu. nous avons plusieurs méthodes qui y sont définies telles que le zoom in le zoom out le redimensionnement, obtenirrapportDaspect et obtenir matrice. Ainsi l’on veut être sur que ces différentes méthodes fonctionnent correctement d'où le choix de tester les trois premières citées ci dessus. | | | |

|  |  |
| --- | --- |
| **Cas de test #1** | |
| **Méthode testée** | ProjectionOrtho::ZoomerIn()  (ZoomInTest) |
|  | |
| **Justification** | |
| cette méthode permet d’augmenter la taille de notre zone de jeu et tous les éléments qui y sont contenu. nous voulons nous assurer que les valeurs attendues sont exactement celles obtenues. | |
|  | |
| **Explication du cas de test** | |
| Pour ce faire nous instancions deux vecteurs de taille deux. l'un pour récupérer la largeur et la hauteur de notre fenêtre virtuelle avant le zoomerIn de notre fenêtre et l’autre pour récupérer la largeur et la hauteur de notre fenêtre virtuelle après le zoom in nous ré-appliquons l’algorithme du zoom in sur les valeurs de notre premier vecteur ensuite on vérifie si ces derniers équivalent au deuxième vecteur. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test #2** | |
| **Méthode testée** | ProjectionOrtho::ZoomerOut()  (ZoomOutTest) |
|  | |
| **Justification** | |
| cette méthode permet de diminuer la taille de notre zone de jeu et tous les éléments qui y sont contenu. nous voulons nous assurer que les valeurs attendues sont exactement celles obtenues. | |
|  | |
| **Explication du cas de test** | |
| Pour ce faire nous instancions deux vecteurs de taille deux. l'un pour récupérer la largeur et la hauteur de notre fenêtre virtuelle avant le zoomerOut de notre fenêtre et l’autre pour récupérer la largeur et la hauteur de notre fenêtre virtuelle après le zoom out nous ré-appliquons l’algorithme du zoom out sur les valeurs de notre premier vecteur ensuite on vérifie si ces derniers équivalent au deuxième vecteur. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test #3** | |
| **Méthode testée** | ProjectionOrtho::RedimensionnerFenetre() (RedimensionerFenetreTest) |
|  | |
| **Justification** | |
| Cette méthode permet d’ajuster les coordonnées de la fenêtre virtuelle suite à un redimensionnement de la clôture. L'agrandissement de la fenêtre virtuelle est proportionnel à l'agrandissement de la clôture afin que les objets gardent la même grandeur apparente lors du redimensionnement de la fenêtre. Nous voulons nous assurer que les valeurs attendues sont exactement celles obtenues. | |
|  | |
| **Explication du cas de test** | |
| Nous créons deux vecteurs de dimension deux. Le premier permet de récupérer les coordonnées initiale de la fenêtre. Ensuite nous appelons la méthode redimensionnerFenetre sur l'instanciation de l'objet projectionOrtho on récupère donc les dimensions de la fenêtre grâce au deuxième vecteur. Nous créons également deux vecteurs de dimension deux. Le premier permet de récupérer les coordonnées initiale de la cloture. Ensuite nous appelons la méthode redimensionnerFenetre sur l'instanciation de l'objet projectionOrtho on récupère donc les dimensions de la fenêtre grâce au deuxième vecteur en notant bien évidemment que les nouvelles valeurs qui sont transmises à notre clôture sont celles passer en paramètre à notre fonction. Et on vérifie grâce à notre algorithme si les dimensions obtenues sont exactes | |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #6** | | | |
| **Classe testée** | NoeudComposite (NoeudCompositeTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| Nous testons ici un nœud composite*.* Il est intéressant de tester cette classe, puisque nous pouvons vérifier le bon fonctionnement des méthodes dans ce noeud. Nous testons notamment l’effacement d’un objet sélectionné, la recherche d’un enfant, le nombre d’enfants retournés par le parent et le désélectionnement des noeuds. Nous testons aussi si nous pouvons vider un parent de ses enfants et nous nous assurons enfin si un enfant par exemple la rondelle peut être tuée. Pour faire ces tests, nous avons créé un noeudComposite, dont une table. Nous pouvons utiliser les noeudRondelle, noeudMaillet et noeudBonus pour tester nos fonctions, puisque ceux-ci dérivent d’un noeud composite. | | | |

|  |  |
| --- | --- |
| **Cas de test #1** | |
| **Méthode testée** | NoeudComposite::effacerSelection (testEffacerSelection) |
|  | |
| **Justification** | |
| effacer une selection(un objet sélectionné)  Ce test permet de tester si la fonction effacerSelection() fonctionne bien.  Il est utile de le tester, puisque l'édition d'une zone de jeu est une partie  importante du livrable 1. Pour éditer, la suppression d'un objet est une  partie très importante. De plus, il est aussi possible de vérifier en même  temps si la fonction ajouter(NoeudAbstrait\* enfant) est fonctionnelle.  La fonction ajouter(...) est primordiale, car elle permet d'ajouter  différents enfants à notre arbre permettant alors de les manipuler | |
|  | |
| **Explication du cas de test** | |
| On vérifie qu’initialement, la table n’a pas d’enfant à l’aide des deux fonctions obtenirNombreEnfants() et calculerProfondeur(). Par la suite, on essaye d’ajouter un noeud quelconque. Dans notre cas, nous avons ajouté un NoeudBonus. Ensuite, on vérifie si le bonus a été ajouté. On sélectionne le bonus avec assignerSelection(bool), et on appelle la fonction pour le supprimer, effacerSelection(). Pour finir, on vérifie que la table n’a plus d’Enfant après l’effacement. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test #2** | |
| **Méthode testée** | NoeudComposite::tuerRondelle (testTuerRondelle) |
|  | |
| **Justification** | |
| tuer la rondelle  Ce test permet de tester si la fonction tuerRondelle() fonctionne bien.  Il est utile de le tester, puisque nous devons être capable de tuer la  rondelle lorsque nous passons du mode test au mode d'édition, ainsi que  lorsque nous sortons de partie rapide et de tournoi. En effet, nous ne  devons plus voir la rondelle sur la table rendu en mode d'édition.  De plus, il est intéressant de le tester, car le bon fonctionnement  de celle-ci implique que les fonctions pour tuer les 2 maillets sont  également fonctionnelles. | |
|  | |
| **Explication du cas de test** | |
| On vérifie qu’initialement, la table n’a pas d’enfant à l’aide des fonctions obtenirNombreEnfants() et calculerProfondeur(). On essaye par la suite d’ajouter une rondelle en la créant et en utilisant ajouter(NoeudAbstrait \*enfant). Puis, on vérifie si la rondelle a été ajouté. La fonction pour tuer la rondelle a été appelée ensuite. On vérifie que la table n’a plus d’enfant après l’effacement. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test #3** | |
| **Méthode testée** | NoeudComposite::obtenirNombreEnfants(testobtenirNombreEnfants) |
|  | |
| **Justification** | |
| Etant donné que nous avons un noeud composite NoueudTable qui est le socle de notre air de jeu, il est important pour nous de nous rassurer que ce noeud contient exactement le nombre d’enfants qu’on lui assigne. | |
|  | |
| **Explication du cas de test** | |
| Au départ, on se rassure que la table ne contient aucun enfant. On lui ajoute un enfant (un maillet par exemple) et on teste si la méthode retourne effectivement la valeur 1. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test #4** | |
| **Méthode testée** | NoeudComposite::vider(testVider) |
|  | |
| **Justification** | |
| Nous avons besoin de jouer à plusieurs reprises. On doit ainsi pouvoir vider la table de ses objets (ses enfants) après chaque partie du jeu. | |
|  | |
| **Explication du cas de test** | |
| test1: Quelque soit le nombre d’enfants que contient NoeudTable, on vide la table et on vérifie si elle n’a plus d’enfants.  test2: La table étant vide, on ajoute nous-mêmes un bonus accélérateur sur la table et on vide encore la table puis on vérifie qu’elle n’a aucun enfant. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test #5** | |
| **Méthode testée** | NoeudComposite:: chercher(const std::string& typeNoeud)  (chercherTest) |
|  | |
| **Justification** | |
| Pour faire une sélection qui est une partie charnière de notre projet on doit être capable de chercher les éléments de notre arbre et ainsi appliquer diverses opérations sur lui.  d’autre part chercher permet de savoir si un noeud est dans l’arbre ou pas. | |
|  | |
| **Explication du cas de test** | |
| pour réaliser ce test nous avons créé une table lors de l’initialisation des objets pour le test de notre classe. nous créons dans la methode de test, trois noeuds de type différent soit noeud Bonus, noeud Maillet et noeud but. Ensuite nous les ajoutons à notre table. enfin on vérifie si la recherche dans l’arbre des noeuds but, maillet et bonus donne respectivementle noeud 1, noeud 2 et noeud3. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test #6** | |
| **Méthode testée** | NoeudComposite::deselectionnerTout (testDeselectionnerTout) |
|  | |
| **Justification** | |
| Il est très intéressant de tester la fonction désélectionner tout. En effet, cette fonction permet de désélectionner tous les enfants, ainsi que le noeud. Elle nous sert à enlever la sélection lors de la manipulation d’objet dans le mode d’édition. Donc, il est possible d’affirmer que cette fonction est très utile, puisqu’elle sert beaucoup lors du livrable 1 en mode édition. | |
|  | |
| **Explication du cas de test** | |
| On vérifie que le noeud composite n’est pas sélectionné, de même que ses enfants à l’aide de selectionExiste(). On crée par la suite des objets, les ajoute au noeud composite et on les sélectionne avec ajouter(NoeudAbstrait \*enfant) et assignerSelection(bool). On vérifie encore une fois s’il existe des objets sélectionnés. Cette deuxième vérification devrait nous retourner true. Pour finir, on désélectionne tout, et on vérifie si le noeud composite ou ses enfants sont sélectionnés. En quelque sorte, dans ce seul test, nous avons pu tester deux fonctions, dont deselectionnerTout() et selectionExiste(). | |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #7** | | | |
| **Classe testée** | NoeudBonus (NoeudBonusTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| Nous avons choisi NoeudBonus arbitrairement pour tester la fonction attribuerCouleur(). Par contre, il est possible de dire que cette fonction est fonctionnelle après le test pour tous les autres noeuds. Peu importe le noeud, le test pour attribuerCouleur est très important, car la sélection par couleur dépend de celle-ci. | | | |

|  |  |
| --- | --- |
| **Cas de test #1** | |
| **Méthode testée** | NoeudBonus::attribuerCouleur (testAttribuerCouleur) |
|  | |
| **Justification** | |
| Il est pertinent de faire ce test, puisque c’est cette fonction qui permet d’attribuer et d’assigner une couleur aux objets affichés à l’écran. Grâce aux couleurs, il est possible de différencier clairement chaque objet qui est sur la table. De plus, la sélection par couleur peut être fonctionnelle seulement s’il y a une couleur pour chaque objet. De manière arbitraire, nous avons testé uniquement cette fonction pour le NoeudBonus. Cependant, ce test permet de vérifier implicitement le bon fonctionnement de cette même fonction dans les autres noeuds. | |
|  | |
| **Explication du cas de test** | |
| On récupère la première couleur de l’objet à l’aide de getCouleur1(), getCouleur2() et getCouleur3(). Par la suite, on vérifie que la première couleur de l’objet est la bonne couleur. | |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #8** | | | |
| **Classe testée** | Camera (CameraTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| Nous avons décidé de tester la fonction deplacerXY dans la classe camera. Ce test est intéressant à faire, puisqu’il permet d’assurer que le déplacement de la vue à l’aide du déplacement de la caméra fonctionne bien. | | | |

|  |  |
| --- | --- |
| **Cas de test #1** | |
| **Méthode testée** | Camera::deplacerXY (testDeplacerXY) |
|  | |
| **Justification** | |
| Il est pertinent de tester cette fonction, puisqu’elle permet de déplacer la vue à l’aide des touches flèchées du clavier. Le déplacement de la vue se fait à l’aide du déplacement de la caméra dans le plan perpendiculaire à la direction visée. Cette fonction prend en paramètre son déplacement en x et en y. En déplaçant la vue, il est possible de voir la caméra “bouger”. | |
|  | |
| **Explication du cas de test** | |
| Le premier test consiste à affirmer les valeurs initiales pour le déplacement de la caméra. C’est-à-dire qu’on vérifie la position de la caméra à l’aide de EGAL\_ZERO et obtenirPosition(), ainsi que vérifier le point visé par la caméra avec EGAL\_ZERO et obtenirPointVise(). Par la suite, on modifie la position et le point visé avec les fonctions assignerPosition(const glm::dvec3) et assignerPointVise(const glm::dvec3). Après la modification, le deuxième test consiste à vérifier que la position et le point visé ont bien été modifiés. | |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test # 9** | | | |
| **Classe testée** | NoeudMaillet(NoeudMailletTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| En raison, de l’implémentation prochaine des collisions, ils nous sera utile de savoir que nous recevons et implémentant les bonnes valeurs des variables rayon\_ et angleRotation\_ afin d’être en mesure de bien implémenter les collisions ainsi que la physique du jeu qui vaut 20% de la note. Donc très important d’être certain de cela. | | | |

|  |  |
| --- | --- |
| **Cas de test # 1** | |
| **Méthode testée** | NoeudMailletTest::testGetRayonMaillet |
|  | |
| **Justification** | |
| Lors que nous aurons a faire la physique, les maillets vont souvent entrer en contact avec la rondelle. Donc, les calcules au niveau de la physique en auront énormément besoin. Ainsi, il est très important d’être sûr que cette méthode fonctionne correctement. | |
|  | |
| **Explication du cas de test** | |
| Pour ce test, nous avons au préalable initialisé le constructeur du maillet avec un rayon de notre choix. Nous récupérons ensuite cette valeur et la stockons dans une variable temporaire. Au final, nous comparons ces 2 variables dans un CPPUNIT\_ASSERT. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test # 2** | |
| **Méthode testée** | NoeudMailletTest::testSetRayonMaillet |
|  | |
| **Justification** | |
| Dans le mode Edition, il est possible de modifier la taille du maillet, via l’option dans la barre, les icones ou en spécifiant une valeur directement dans un textBox dans le panel à droite. Ils nous est donc primordiale de pouvoir modifier la valeur du rayon du maillet afin que nous puissions modifier l'interaction entre le maillet et la rondelle dans le jeu. | |
|  | |
| **Explication du cas de test** | |
| Pour ce teste, il faut que le teste précédant fonctionne (donc on doit pouvoir recupérer les valeur du rayon). Ensuite, nous créons une variable temporaire qui contient la nouvelle valeur du rayon. Après, nous modifions la valeur du rayon du maillet via la fonction que nous voulons tester. Par la suite, on utilise la fonction tester précédemment (getRayonMaillet) afin de récupérer la valeur de la variable de la classe NoeudMaillet (rayon\_) et la stocker dans une variable temporaire. Finalement, on compare les 2 variables temporaires que l’on a créé. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test # 3** | |
| **Méthode testée** | NoeudMailletTest::testGetAngle |
|  | |
| **Justification** | |
| Comme pour la méthode getRayon(), il nous est impératif d’être sûr et certain de pouvoir récupérer la valeur de l’angle qui elle aussi est un point très important pour le calcule de la physique qui sera à réaliser plus tard. | |
|  | |
| **Explication du cas de test** | |
| La méthode qui teste cela suit le même principe que celle du cas 1. On récupère d’abord la valeur du rayon du maillet dans une variable temporaire afin de la comparer plus tard avec celle qu’on a mit dans le constructeur de l’objet maillet. Si elles sont les mêmes, le teste réussi. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test # 4** | |
| **Méthode testée** | NoeudMailletTest::testSetAngle |
|  | |
| **Justification** | |
| Comme pour le rayon, on doit être en mesure de s’assurer que notre angle peut être modifié afin d’ajuster nos objets suite à une collision entre eux. | |
|  | |
| **Explication du cas de test** | |
| Pour ce test, nous avons fait en sorte de modifier l’angle de notre maillet via une variable qui a déjà été initialisée. Ensuite, il faudra qu’on puisse récupérer notre variable via la fonction getRayonMaillet() que nous avons testé précédemment est la soustraire avec notre variable dont nous avons initialement modifier l’angle via la fonction que nous testons. Si la fonction fonctionne correctement, on a zéro comme résultat et la fonction utilitaire::EGAL\_ZERO retournera un vrai comme boolean qui nous confirmera que notre fonction est bien implémentée. | |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Suite de cas de test #10** | | | |
| **Classe testée** | Utilitaire (UtilitaireTest) | **Branche** | master |
|  | | | |
| **Justification** | | | |
| La classe Utilitaire est très utilisée dans le projet. Dans notre cas, nous trouvons pertinent de tester la fonction DANS\_LIMITESXY(double, double, double, double, double, double). En effet, cette fonction nous a permis de définir notre table. C’est-à-dire qu’il était possible de délimiter la table, laissant aucun objet sortir de celle-ci ou même empêchant l’ajout d’objet à l’extérieur de la table. | | | |

|  |  |
| --- | --- |
| **Cas de test #1** | |
| **Méthode testée** | Utilitaire::DANS\_LIMITESXY (testLimite) |
|  | |
| **Justification** | |
| Cas de test: vérifie s'il est dans l'intervalle  Ce test permet de tester si les objets mis sur la table sont dans  la zone de jeu. En effet, nous avons délimité la table à l'aide  de cette fonction, empêchant les objets de sortir des limites imposées. | |
|  | |
| **Explication du cas de test** | |
| On vérifie si une valeur quelconque se retrouve à l'intérieur de la zone de jeu  Dans notre cas, la zone de jeu (table) est définie de -70 à 70 en x et -16 à 16 en y. Par la suite, on vérifie si une valeur quelconque se retrouve bel et bien à l'extérieur de la zone de jeu. | |
|  | |