INF2990

Projet de logiciel graphique interactif

Tests logiciels

Version 16.3

Auteur original : Julien Gascon-Samson

Éditeurs : Olivier Gendreau
Martin Paradis

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.

Suite de cas de test

Suite de cas de test # 1			
Classe testée	NoeudAbstrait (NoeudAbstraitTest)	**Branche**	testUnitaires
Justification			
Comme dans les exemples de tests, uniquement les nœuds non-composites seront testés. De plus, il est intéressant de tester cette classe, parce qu'on peut tester les méthodes à la base de l'arbre de rendu. Nous testons les booléens liés à l'enregistrement des nœuds dans un fichier XML, ainsi que la couleur de sélection du nœud. Puisque la sélection d'un objet est important pour plusieurs fonctionnalités telles que la rotation, le déplacement, la mise à échelle et la duplication, ce test est important pour assurer le bon fonctionnement de ces fonctionnalités. Pour être cohérent avec les exemples de tests, nous avons utilisé la classe NoeudConeCube également.			

Cas de test # 1	
Méthode testée	NoeudAbstrait::estEnregistrable()
Justification	
Ce test est important, parce que cette méthode est essentielle pour le bon fonctionnement de la sauvegarde XML. La sauvegarde en fichier XML est utilisé pour chaque mode de jeu. Donc il est donc pertinent de tester si les nœuds qui doivent être enregistrés peuvent être enregistrés.	
Explication du cas de test	
On ajoute d'abord un nœud qui est initialement non-enregistrable. On modifie pour faire en sorte que le nœud est enregistrable et on vérifie si c'est le cas	
Cas de test # 2	
Méthode testée	NoeudAbstrait::attribuerCouleurSelection()
Justification	
L'assignation de la couleur d'un nœud est essentiel puisque la méthode de sélection que nous avons choisi est la sélection par couleur. Il est donc important que nous testons que la couleur est bien assignée pour la sélection d'un objet et la sélection de plusieurs objets.	
Explication du cas de test	
Tout d'abord, nous vérifions si la couleur du nœud a été bien initialisée, c'est-à-dire une couleur de (205,205,205). Ensuite, nous assignons une couleur aléatoire autre que la couleur au début.	

Suite de cas de test # 2			
Classe testée	NoeudPortail (NoeudPortailTest)	**Branche**	testUnitaires
Justification			
Il est intéressant de tester cette classe, parce qu'un nœud portail a une caractéristique spéciale qu'il ne partage pas avec les autres nœuds; un nœud portail doit toujours avoir un frère, qui a, à sa création, les mêmes attributs que le portail original. De plus, la vérification de la couleur du portail est différent des autres nœuds. Il est donc nécessaire de tester cette classe en plus de la classe NoeudAbstrait. Nous testons alors si le frère d'un portail est bien présent, après la création de l'original, et si la bonne couleur a été assignée.			

Cas de test # 1	
Méthode testée	NoeudPortail::setFrere() NoeudPortail::getFrere()
Justification	
Pour avoir un fonctionnement du portail, il faut s'assurer que le portail a un frère; un portail téléporte la rondelle à un autre portail. Il est donc nécessaire de vérifier qu'un portail a toujours un frère quand il est créé.	
Explication du cas de test	
Nous créons d'abord deux portails sans frère. Ensuite, nous assignons à chacun l'autre en tant que frère et nous vérifions si les frères ont été assignés.	

Cas de test # 2	
Méthode testée	NoeudPortail::verifierCouleur()
Justification	
Pour appliquer des nœuds un déplacement, une rotation ou supprimer un nœud, il faut vérifier la sélection par couleur de l'objet. Puisque chacun des objets à une façon de vérifier la couleur, il est pertinent de le tester.	
Explication du cas de test	
Nous créons d'abord un portail et une couleur. Ensuite, nous assignons cette couleur au nœud portail que nous avons créé. Enfin, nous vérifions si la couleur de l'objet portail est la même que la couleur créée au départ.	

Cas de test # 3	
Méthode testée	NoeudPortail::copierAttributs()
Justification	
Puisqu'un portail a un frère et que le frère a les mêmes attributs que le portail original lorsqu'il est créé, il faut être capable de copier les attributs d'un portail à l'autre. Il est donc pertinent de tester si la méthode qui copie les attributs d'un portail à un autre est fonctionnel.	
Explication du cas de test	
Au début, nous créons un autre portail. Ensuite, nous modifions le rayon du portail original. Nous copions les attributs du portail original au portail qui a été créé. Enfin, nous vérifions si les deux portails ont les mêmes attributs.	

Suite de cas de test # 3			
Classe testée	NoeudTable (NoeudTableTest)	**Branche**	testUnitaires
Justification			
La classe NoeudTable contient des propriétés qui affectent le comportement de la table. Pour assurer le bon fonctionnement de la rondelle, il faut tester cette classe. Nous testons donc le coefficient de rebondissement de la table, la friction de la table et le coefficient d'accélération. De plus, nous testons l'ajout d'une rondelle à la table. Puisque, dans le mode Test, le mode Partie Rapide et le mode Tournoi, il faut une rondelle pour jouer.			

Cas de test # 1	
Méthode testée	NoeudTable::modifierCoefRebondissement() NoeudTable::modifierAcceleration() NoeudTable::modifierFriction()
Justification	
Pour le fonctionnement de la physique de la rondelle et le fonctionnement des bonus accélérateurs, il faut tester les méthodes qui modifient le coefficient de rebondissement, le coefficient de friction et le coefficient d'accélération de la table.	
Explication du cas de test	
Nous établissons une valeur de départ pour les 3 valeurs. Ensuite, nous modifions ces valeurs et vérifions qu'ils ont la bonne valeur.	

Suite de cas de test # 4			
Classe testée	NoeudComposite (NoeudCompositeTest)	**Branche**	testUnitaires
Justification			
La classe NoeudTable contient des propriétés qui affectent le comportement de la table. Pour assurer le bon fonctionnement de la rondelle, il faut tester cette classe, puisque nous pouvons vérifier le bon fonctionnement des méthodes à la base de l'arbre de rendu qui n'ont pas été couvertes par NoeudAbstrait. Nous testons l'ajout d'enfants dans un nœud composite et la recherche d'un nœud enfant dans le nœud composite.			

Cas de test # 1	
Méthode testée	NoeudComposite::ajouter()
Justification	
Le patron Composite permet d'ajouter des nœuds feuilles et nœuds branches. Il faut s'assurer qu'il est possible d'ajouter des nœuds feuilles et que les nœuds sont bels et bien ajoutés. Il faut également s'assurer que tous les types de nœuds peuvent être ajoutés. Nous testons également la méthode qui permet d'effacer un enfant du nœud composite.	
Explication du cas de test	
Nous ajoutons d'abord un nœud composite sans enfant. Ensuite, nous ajoutons un enfant et vérifions le nombre d'enfants dans le nœud composite.	

Cas de test # 2	
Méthode testée	NoeudComposite::vider()

Justification
Puisqu'il est nécessaire de vider le nœud composite de tous ces enfants lorsque le nœud composite est détruit, il est nécessaire de tester cette méthode.

Explication du cas de test
Nous ajoutons d'abord 3 enfants dans le nœud composite. Ensuite, nous faisons appel à la méthode vider(). Enfin, nous vérifions si le nœud composite est vide.

Cas de test # 3	
Méthode testée	NoeudComposite::chercher(int id)

Justification
Il est nécessaire de pouvoir chercher un enfant d'un nœud composite par un identifiant, si la recherche par le type ne suffit pas. Il est donc important de tester cette méthode pour s'assurer que le bon nœud est renvoyé. De plus, il faut pouvoir chercher le nœud enfant afin d'appliquer une rotation, un déplacement, un mise à l'échelle, etc.

Explication du cas de test
Nous ajoutons d'abord un nœud bonus accélérateur dans un nœud composite. Ensuite, nous cherchons l'objet qui vient d'être ajouté par son identifiant. Nous vérifions ensuite si ce sont les mêmes nœuds.

Cas de test # 4	
Méthode testée	NoeudComposite::chercher(string typeNoeud)

Justification
Il est nécessaire de pouvoir appeler des fonctions sur les nœuds enfants du nœud composite. Pour cela, la recherche d'un nœud enfant dans un nœud composite est important. Donc, cette méthode est intéressant à tester, pour s'assurer qu'on obtient le bon nœud enfant. Cette méthode ne retourne qu'un nœud; s'il existe plusieurs nœuds de même type, le premier rencontré sera retourné.

Explication du cas de test
Nous ajoutons d'abord un nœud bonus accélérateur dans un nœud composite. Ensuite, nous cherchons l'objet qui vient d'être ajouté par le type de l'objet en string. Nous vérifions ensuite si ce sont les mêmes nœuds.

Suite de cas de test # 5			
Classe testée	VisiteurDeplacement (VisiteurDeplacementTest)	**Branche**	testUnitaires

Justification
Cette classe effectue le déplacement des objets. Puisque le contrôle des maillets par les joueurs et le déplacement des multiples objets en mode Édition utilisent ce visiteur, il est pertinent de tester cette classe pour assurer un déplacement fluide des maillets et de la rondelle. Nous testons le visiteur avec la classe NoeudPortail.

Cas de test # 1

Méthode testée	VisiteurDeplacement::visiter()

Justification
Les objets déplacés en mode Édition acceptent le visiteur déplacement lorsqu'il visite. Il est donc nécessaire de tester la méthode visiter pour s'assurer que le visiteur est bien implémenté.

Explication du cas de test
Nous créons d'abord un nœud portail qu'on assigne la position à (0,0,0). Ensuite, nous faisons appel à cette méthode pour déplacer cet objet. Enfin, pour la vérification, nous assurons que l'objet a été bien déplacé aux bonnes coordonnées.

Suite de cas de test # 5			
Classe testée	VisiteurRotation (VisiteurRotationTest)	**Branche**	testUnitaires

Justification
Cette classe effectue la rotation des objets. Puisque l'animation des portails pendant une partie et la rotation des multiples objets en mode Édition utilisent ce visiteur, il est pertinent de tester cette classe pour assurer une bonne rotation des objets. Nous testons le visiteur avec la classe NoeudPortail.

Cas de test # 1	
Méthode testée	VisiteurRotation::visiter()

Justification
Les objets qui effectuent une rotation dans le mode Édition acceptent le visiteur de rotation lorsqu'il visite. Il est nécessaire de tester la méthode visiter() pour s'assurer que le visiteur est bien implémenté et que les objets effectuent bien la rotation voulue.

Explication du cas de test
Nous créons d'abord un nœud portail comme pour le visiteur de déplacement. Nous modifions d'abord son angle de rotation. Ensuite, avec le visiteur, nous spécifions la variation en Y, qui sera ajoutée à l'angle de rotation. Enfin, nous vérifions que la somme est correcte.

Suite de cas de test # 5			
Classe testée	VisiteurEchelle (VisiteurEchelleTest)	**Branche**	testUnitaires

Justification
Cette classe modifie la taille des objets. Puisqu'il doit être permis de modifier la taille des objets dans le mode Édition, il est pertinent de tester cette classe pour assurer une bonne rotation des objets. Nous testons le visiteur avec la classe NoeudPortail.

Cas de test # 1	
Méthode testée	VisiteurEchelle::visiter()

Justification
Les objets voient un changement dans sa taille dans le mode Édition acceptent le visiteur de mise à échelle lorsqu'il visite. Il est nécessaire de tester la méthode visiter() pour s'assurer que le visiteur est bien implémenté et que les objets changent bien leur taille lorsque le visiteur est accepté.

Explication du cas de test
Tout d'abord, un rayon est assigné à notre nœud portail. Ensuite, nous appliquons une mise à l'échelle au rayon du portail. Nous vérifions si le rayon est bien calculé. Enfin, nous modifions encore une fois le rayon du portail et vérifions si le résultat est juste.

Suite de cas de test # 5			
Classe testée	VisiteurSuppression (VisiteurSuppressionTest)	**Branche**	testUnitaires
Justification			
Dans le mode Édition, il faut être capable de supprimer des objets qui sont sur la table. Donc, il est important de tester ce visiteur, pour s'assurer qu'il est possible de supprimer des objets sur la table. Nous testons le visiteur avec la classe NoeudBonusAccelerateur.			

Cas de test # 1	
Méthode testée	VisiteurSuppression::visiter()
Justification	
Les objets qui sont supprimés acceptent le visteur de suppression. Il est nécessaire de tester la méthode visiter() pour s'assurer que le visiteur est bien implémenté et que les objets sont bien supprimés lorsque le visiteur est accepté.	
Explication du cas de test	
Nous créons d'abord un bonus accélérateur. Ensuite, nous le sélectionnons, puisqu'un nœud doit être sélectionné avant d'être supprimé. Nous vérifions qu'il n'a pas encore été ajouté. Ensuite, nous l'ajoutons à notre arbre et vérifier s'il est dans l'arbre. Nous le supprimons et vérifions s'il a été supprimé de l'arbre.	

Suite de cas de test # 6			
Classe testée	ProfilVirtuel(ProfilVirtuelTest)	**Branche**	testUnitaires
Justification			
Les profils virtuels sont une partie intégrale d'un joueur virtuel et affecte son comportement. Le joueur virtuel est important pour la partie virtuelle, le tournoi et le mode test. De plus, les profils doivent être configurables dans le menu de Configuration. Il est donc pertinent de tester si le profil est bel et bien fonctionnel avec les méthodes de configuration sont fonctionnelles.			

Cas de test # 1	
Méthode testée	ProfilVirtuel::modifierNom() ProfilVirtuel::modifierVitesse() ProfilVirtuel::modifierPassivite()
Justification	
Dans le menu de configuration, le nom, la vitesse et le degré de passivité décrivent le profil d'un joueur virtuel. Il est important que les modifications que nous voulons apporter sur ces valeurs.	
Explication du cas de test	
Nous créons d'abord un joueur virtuel sans nom, vitesse et degré de passivité. Ensuite, nous modifions son nom, sa vitesse et sa passivité. Enfin, nous vérifions ses attributs pour vérifier	

que les modifications ont été bien faites.

Suite de cas de test # 7			
Classe testée	Partie (PartieTest)	**Branche**	testUnitaires
Justification			
Une partie est à la base des mode de jeu Partie Rapide et Tournoi. En effet, un tournoi peut être vu comme une série de parties. Puisque ces modes de jeu repose sur le bon fonctionnement d'une partie, il est important de tester cette classe pour s'assurer qu'il y a un vainqueur dès qu'un joueur atteint le nombre de buts nécessaire pour gagner.			

Cas de test # 1	
Méthode testée	Partie::assignerbutJ1() Partie::asignerbutJ2() Partie::obtenirVainqueur()
Justification	
Lorsqu'un joueur marque un but, il faut s'assurer que le nombre de buts du joueur est mis à jour. Ceci permet de déclarer un gagnant pour une partie. Il est donc nécessaire de tester les méthodes qui sont nécessaires pour le déroulement d'une partie	
Explication du cas de test	
Nous créons d'abord deux joueurs qui sont utilisés pour la partie, ainsi que le nombre de buts nécessaires pour gagner. Ensuite, nous faisons en sorte qu'un des joueurs atteint le nombre de buts nécessaires pour gagner. Enfin, nous vérifions si le bon joueur est déclaré vainqueur.	

Suite de cas de test # 8			
Classe testée	NoeudJoueur(NoeudJoueurTest)	**Branche**	testUnitaires
Justification			
Pour une partie, il existe toujours deux joueurs. Il faut être capable de déterminer le vainqueur avec le nombre de buts de chacun des joueurs. De plus, le nom du vainqueur doit être affiché à la fin d'une partie dans le cas d'une Partie Rapide. Il est donc important de tester cette classe pour avoir une partie fonctionnelle.			

Cas de test # 1	
Méthode testée	NoeudJoueur::assignerNom()
Justification	
Le nom d'un joueur doit pouvoir être affiché dans le cas d'un tournoi. Il est possible de modifier le nom d'un joueur dans le menu de configuration. Il faut donc tester s'il est possible de modifier le nom d'un joueur.	
Explication du cas de test	
Nous créons d'abord un joueur sans nom. Ensuite, nous lui donnons un nom et vérifions si le nom a été bien assigné.	

Cas de test # 2	
Méthode testée	NoeudJoueur::incrementerBut() NoeudJoueur::reinitialiserBut()

Justification
Dans le cas d'une partie rapide, il doit être possible de réinitialiser une partie en pesant sur un bouton pendant une partie. Il faut donc tester cette méthode pour s'assurer que le nombre de buts peut être réinitialisé et incrémenté.
Explication du cas de test
Nous créons d'abord un joueur. Nous faisons en sorte qu'il marque un but pour vérifier si le compte des buts est bon. Ensuite, nous réinitialisons le nombre de buts et nous nous assurons si le nombre de buts est nul.

Cas de test # 3	
Méthode testée	NoeudJoueur::assignerMaillet()
Justification	
Lorsqu'une partie est commencée, il faut que les deux joueurs aillent des maillets qu'ils peuvent contrôler. Il est donc nécessaire de tester la méthode qui assigne un maillet à chacun des joueurs.	
Explication du cas de test	
Nous créons d'abord un maillet. Ensuite, nous assignons au joueur créé le maillet nouvellement créé. Enfin, nous vérifions si le maillet qui a été assigné au joueur est le même maillet qui a été créé au début.	

Suite de cas de test # 9			
Classe testée	ProjectionOrtho (ProjectionOrthoTest)	**Branche**	testUnitaires
Justification			
Lorsqu'on redimensionne une fenêtre, la fenêtre graphique doit être mise à jour et les objets ne doivent pas changer de taille. Cette classe contient des méthodes qui sont nécessaires pour le redimensionnement de la fenêtre graphique. Il est donc nécessaire de tester des méthodes utilisées pour le redimensionnement de la fenêtre, qui était une fonctionnalité demandée au premier livrable. De plus, il doit être possible de faire un « zoom in » et un « zoom out ».			

Cas de test # 1	
Méthode testée	ProjectionOrtho::redimensionnerFenetre()
Justification	
Cette méthode est nécessaire pour le redimensionnement de la fenêtre graphique. En effet, cette méthode assure que le redimensionnement de la fenêtre virtuelle est proportionnelle à l'agrandissement de la clôture pour assurer que les objets ne changent pas de taille. Il est donc nécessaire de tester cette méthode pour assurer que le redimensionnement se fait correctement.	
Explication du cas de test	
Nous initialisons d'abord une vue. Ensuite, nous redimensionnons la fenêtre jusqu'à un certain point en x et y. Enfin, nous vérifions si le redimensionnement se fait jusqu'au nouveau point.	

Cas de test # 2	
Méthode testée	ProjectionOrtho::zoomerIn()
Justification	
Cette méthode est utilisée pour le « zoom in » de la fenêtre. Elle calcule la nouvelle largeur et	

hauteur de la fenêtre après le redimensionnement. Il est donc nécessaire de tester cette méthode pour s'assurer que le zoom est fonctionnel.
Explication du cas de test
Nous initialisons d'abord une vue. Ensuite, nous effectuons un zoom sur la fenêtre. Enfin, nous vérifions si le zoom a été fait.

Cas de test # 3	
Méthode testée	ProjectionOrtho::zoomerOut()
Justification	
Comme pour la méthode zoomerIn(), cette méthode est importante pour le fonctionnement du « zoom » (« zoom out » dans ce cas-ci).	
Explication du cas de test	
Nous initialisons d'abord une vue. Ensuite, nous effectuons un zoom sur la fenêtre. Enfin, nous vérifions si le zoom a été fait.	