



**POLYTECHNIQUE
MONTRÉAL**

LE GÉNIE
EN PREMIÈRE CLASSE

Département de génie informatique et génie logiciel

INF2990 – Projet logiciel graphique interactif

Tests logiciels

Présenté par

Philippe CARPHIN, 1693091

Yonni CHEN KUANG PIAO, 1678004

Aymen DJELLAL, 1691752

Samuel MILLETTE, 1685529

Nikolay RADOEV, 1683742

Emilio RIVERA, 1689355

Suite de cas de test # 1			
Classe testée	Utilitaire.h	Branche	master
Justification			
<p>La classe utilitaire contient de nombreuses méthodes qui peuvent être utilisées partout dans le code dans le but de faciliter les calculs et d'augmenter la cohésion des classes qui n'auront pas à définir ces méthodes elles-mêmes. Puisque certaines méthodes ont été ajoutées aux méthodes de base du cadriciel, il est important de les tester afin de s'assurer qu'elles accomplissent leur tâche telle que prévu.</p>			

Cas de test # 1	
Méthode testée	Utilitaire :: intersectionDeuxSegments (testIntersection)
Justification	
<p>Ce test permet de vérifier la sélection multiple par les boîtes englobantes. Les intersections entre deux segments sont utilisées pour traiter un cas spécifique de sélection.</p>	
Explication du cas de test	
<p>Cette méthode prend en paramètre 4 points de l'espace, forme deux segments par paire de points et return <i>true</i> si les deux droites s'intersectent. Il suffit de donner des points qui forment des droites qui s'intersectent nécessairement et s'assurer que la méthode fonctionne.</p>	

Suite de cas de test # 2			
Classe testée	NoeudComposite.h	Branche	master
Justification			
<p>La classe NoeudComposite dérive directement de la classe NoeudAbstrait, déjà testée en exemple. Elle nous permet donc également de vérifier le bon fonctionnement de méthodes à la base de l'arbre de rendu. Cependant, puisque cette classe se comporte différemment de NoeudAbstrait et est plus prêt de la vraie nature des objets qui seront insérés dans l'arbre de rendu, il est important de vérifier qu'elle fonctionne comme prévu.</p> <p>Puisque NoeudComposite est une classe abstraite, nous utilisons la classe NoeudCible, puisque celle-ci dérive directement de NoeudComposite.</p>			

--

Cas de test # 1	
Méthodes testées	NoeudComposite :: obtenirNombreEnfants(testEnfants) NoeudComposite :: calculerProfondeur(testEnfants) NoeudComposite :: ajouter(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 composite, il devrait être possible d'ajouter des nœuds enfants. Ce cas de test s'assure que c'est bel et bien le cas.	
Explication du cas de test	
Ce test permet de s'assurer que les objets de type <i>NœudsComposite</i> et ceux de classes qui en dérivent peuvent avoir des enfants, et que les méthodes <i>calculerProfondeur</i> et <i>obtenirNombreEnfants</i> fonctionnent correctement. On s'assure d'abord que le nœud n'a pas d'enfants, puis on lui ajoute un enfant et on vérifie que l'enfant est bien lié au parent et que la profondeur calculée est exacte. On s'assure donc que les enfants sont créés et supprimés correctement.	

Suite de cas de test # 3			
Classes testées	ArbreRenduINF2990.h UsineAbstraite.h NœudComposite.h NœudAbstrait.h VisiteurAgrandissement.h VisiteurDeselectionnerTout.h VisiteurCentreDeMasse.h VisiteurRotationPoint.h VisiteurSelection.h VisiteurDuplication.h VisiteurDeplacement.h VisiteurPossibilite.h VisiteurSelectionInverse.h VisiteurSelectionMultiple.h VisiteurListeEnglobante.h VisiteurSuppression.h	Branche	master
Justification			

La classe *ArbreRenduINF2990*, qui représente l'interface de l'arbre de rendu du projet, est une partie fondamentale qui permet de tester un très grand nombre de classes de méthodes dans la couche logique de l'application. On peut passer par cette classe pour effectuer des tests sur les usines, les nœuds et les visiteurs, entre autres. Ce sont sur ces éléments que cette suite de tests s'attarde plus particulièrement, dans le but d'assurer la qualité et le fonctionnement global de l'arbre de rendu du projet.

Cas de test # 1	
Méthodes testées	<i>ArbreRenduINF2990</i> :: initialiser (testDefault) <i>ArbreRenduINF2990</i> :: estDefault(testDefault) <i>ArbreRenduINF2990</i> :: obtenirNombreEnfants(testDefault) <i>ArbreRenduINF2990</i> :: obtenirProprietes(testDefault)
Justification	
Ce test permet de vérifier si la zone courante est bien la zone de jeu par défaut.	
Explication du cas de test	
Cette méthode teste l'initialisation de l'arbre créé par défaut lorsque l'application d'éditeur est ouverte. La méthode <i>estDefault</i> de la classe <i>ArbreRenduINF2990</i> permet facilement d'effectuer ce test, mais on s'assure également que l'arbre a des enfants, que parmi ceux-ci se trouvent une table, par exemple, et que les propriétés de la partie sont bien initialisées.	

Cas de test # 2 : <i>ArbreRenduINF2990Test</i> ::void testXML()	
Méthodes testées	<i>ArbreRenduINF2990</i> ::initialiserXML(string nomFichier)
Justification	
La lecture et le chargement d'une zone de jeu à partir d'un fichier XML figure parmi les requis plus importants du SRS. Nous voulons nous assurer qu'elle fonctionne parfaitement et que les cas où il est susceptible de se produire des erreurs ou des exceptions sont soit gérés soit évités.	
Explication du cas de test	
Ce test traite le cas où le fichier XML lu pour remplir l'arbre n'existe pas. On s'assure tout simplement que la méthode <i>initialiserXML</i> retourne faux si le fichier n'existe pas, et vrai si le chemin est reconnu. Seulement lorsque le fichier est reconnu, on procède à l'initialisation de l'arbre de rendu. Ce faisant, on évite le fait que l'utilisateur tente d'initialiser incorrectement l'arbre de rendu.	

Cas de test # 3 : <i>ArbreRenduINF2990::testPortails()</i>	
Méthodes testées	NoeudComposite::chercher(string nom)
Justification	
Les portails doivent être créés en paire, dupliquer en paire et supprimés en paire. Ce test nous permet de vérifier que deux jumeaux portails sont correctement reliés ensemble.	
Explication du cas de test	
Initialiser l'arbre de rendu avec un fichier XML contenant deux portails jumeaux. On s'assure que ces deux portails sont reliés grâce à un attribut qui est un pointeur vers un autre portail. On vérifie que ce pointeur n'est pas nul et qu'il pointe vers un portail.	

Cas de test # 4	
Méthode testée	ArbreRenduINF2990 :: creerNoeud(creerNoeudParUsine) UsineAbstraite :: creerNoeud(creerNoeudParUsine)
Justification	
Ce test nous permet de vérifier que la création des nœuds concrets se fait correctement grâce aux usines de fabrication. La création et l'ajout d'objets concrets dans la scène d'objets figure parmi les requis les plus importants du SRS.	
Explication du cas de test	
On crée d'abord un nœud de type non reconnu avec la méthode <i>creerNoeud</i> , et on s'assure que la valeur retournée est un nullptr. Puis, on crée un nœud de type connu (<i>NœudCible</i>), et on s'assure que le nom créé est du bon type.	

Cas de test # 5	
Méthodes testées	NœudAbstrait :: obtenirPositionRelative(boiteEnglobante) NœudAbstrait :: pointEstDansBoite(boiteEnglobante)
Justification	
Ce test nous permet de vérifier que les objets ne sortent pas de la zone de jeu. Les transformations appliquées aux objets peuvent faire en sorte que l'objet sort de la table. Nous voulons éviter ce cas en testant le fonctionnement des boîtes englobantes.	
Explication du cas de test	
On teste les limites de la boîte englobante autour d'un nœud concret de type quelconque	

par la méthode *pointEstDansBoite*. Il suffit de créer l'objet (placé en {0.0, 0.0, 0.0} par défaut) et de vérifier si un point se trouve dans la boîte lorsqu'il est centré en {0.0, 0.0, 0.0} et lorsqu'il est déplacé dans une direction non négligeable quelconque.

Cas de test # 6	
Méthode testée	NœudAbstrait :: estSelectionnable(selectionTable)
Justification	
Il y a certains objets de l'arbre de rendu que l'on ne veut pas modifier. Les modifications passent par la condition suivante : si le nœud est sélectionné, appliquer la transformation. Ainsi, nous voulons tester que certains objets ne soient pas sélectionnable afin de ne pas appliquer de transformations dessus.	
Explication du cas de test	
On s'assure qu'une table n'est pas sélectionnable, c'est-à-dire que l'attribut <i>selectionnable_</i> est défini à false à l'initialisation de l'arbre. La méthode <i>estSelectionnable</i> de la classe <i>NoeudAbstrait</i> retourne cet attribut pour nous. On s'assure également que les nœuds concrets d'autres types sont sélectionnables.	

Cas de test # 7	
Méthodes testées	NœudAbstrait :: assignerSelection(testDeselection) VisiteurDeselectionnerTout :: traiter(testDeselection)
Justification	
Ce test permet de vérifier un requis du SRS : lorsqu'un clic de souris n'est pas au-dessus d'un objet, il faut désélectionner tous les objets anciennement sélectionnés.	
Explication du cas de test	
Test du fonctionnement du <i>VisiteurDeselectionnerTout</i> . On initialise un arbre par défaut, on sélectionne un objet au hasard avec la méthode <i>assignerSelection</i> , puis on appelle le visiteur sur l'arbre et on vérifie que tous les nœuds sont désélectionnés.	

Cas de test # 8	
Méthodes testées	VisiteurCentreDeMasse :: traiter(testRotation) VisiteurRotationPoint :: traiter(testRotation)
Justification	
Ce test permet de vérifier un des requis du SRS concernant l'édition d'objet : la rotation	

simple et multiple.
Explication du cas de test
Test du fonctionnement de <i>VisiteurRotationPoint</i> et de <i>VisiteurCentreDeMasse</i> . On initialise un arbre par défaut, on choisit un objet quelconque et on obtient sa valeur de rotation par la méthode <i>obtenirRotation</i> de <i>NoeudAbstrait</i> . Puis, on visite l'objet avec <i>VisiteurCentreDeMasse</i> pour obtenir le centre de la rotation, et on crée un vecteur de rotation arbitraire qui sera passé au <i>VisiteurRotationPoint</i> avec le centre de rotation. On vérifie ensuite que la nouvelle valeur de rotation de l'objet est différente de l'originale.

Cas de test # 9	
Méthodes testées	<i>VisiteurSelectionMultiple</i> ::traiter(testSelectionMultiple) <i>VisiteurSelectionMultiple</i> :: obtenirNbObjetsSelectionne(testSelectionMultiple)
Justification	
Ce test permet de vérifier un des requis du SRS concernant l'édition d'objet : la sélection multiple.	
Explication du cas de test	
Test du fonctionnement de <i>VisiteurSelectionMultiple</i> . On initialise un arbre par défaut et on ajoute deux cibles à la table. On crée ensuite un <i>VisiteurSelectionMultiple</i> en lui passant en paramètre les points pour créer une boîte de sélection de taille suffisante et on traite l'arbre. La méthode <i>obtenirNbObjetsSelectionne</i> du visiteur doit retourner le bon nombre d'objets sélectionnés.	

Cas de test # 10	
Méthode testée	<i>ArbreRenduINF2990</i> ::getEnfant(testSelectionInverse) <i>VisiteurSelectionInverse</i> ::traiter(testSelectionInverse)
Justification	
Ce test permet de vérifier un des requis du SRS concernant la sélection.	
Explication du cas de test	
Test du fonctionnement de <i>VisiteurSelectionInverse</i> . On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un <i>VisiteurSelectionInverse</i> en lui passant en paramètre un point qui détermine la distance de la cible et le numéro identifiant du type de nœud (par la méthode <i>getNumero</i> de <i>NoeudAbstrait</i> , ici l'identifiant <i>NoeudCible</i> est retourné). On traite l'arbre avec le visiteur, et le bon objet est ensuite traité. Ici, la cible doit être désélectionnée par le passage du visiteur.	

Cas de test # 11	
Méthode testée	VisiteurDuplication ::traiter(testDuplication)
Justification	
Ce test vérifie la duplication des objets sélectionnés selon 3 cas. Premièrement, la duplication simple d'objets. Ensuite, la duplication multiple d'objets autour d'un centre de sélection. Finalement, la duplication des portails où la paire de portail doit être sélectionné avant de procéder à l'opération de duplication.	
Explication du cas de test	
Test du fonctionnement de <i>VisiteurDuplication</i> . On initialise un arbre par défaut et on crée ensuite un <i>VisiteurDuplication</i> et on traite l'arbre. Le nombre d'enfants total de la racine obtenu par la méthode <i>obtenirNombreEnfants</i> doit être plus élevé, puisque tous les enfants ont été dupliqués.	

Cas de test # 12	
Méthodes testées	VisiteurListeEnglobante ::traiter(testBoiteEnglobante) VisiteurListeEnglobante ::obtenirListeEnglobante(testBoiteEnglobante) NœudAbstrait :: obtenirVecteursBoite(testBoiteEnglobante)
Justification	
Ce test permet de vérifier le fonctionnement des boîtes englobantes.	
Explication du cas de test	
Test du fonctionnement de <i>VisiteurListeEnglobante</i> . On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un <i>VisiteurListeEnglobante</i> et on traite la cible avec celui-ci. On obtient alors la liste des boîtes englobantes de nœuds visités (ici le nœud cible seulement) contient bien le bon type de nœud et que les points de la boîte englobante calculés par le visiteur correspondent à ceux obtenus par la méthode <i>obtenirVecteursBoite</i> de <i>NoeudAbstrait</i> .	

Cas de test # 13	
Méthode testée	NœudAbstrait ::getColorShift(testPalettes)
Justification	
Les palettes des différents joueurs doivent être différenciables. Ce test permet de vérifier que ces palettes sont de différentes couleurs.	

Explication du cas de test	
Test que les couleurs des palettes sont différentes pour les joueurs 1 et 2. Une carte de jeu contenant deux palettes (une gauche et une droite) de joueurs différent est chargée, puis on cherche ces deux palettes avec la méthode <i>chercher</i> de <i>NoeudComposite</i> et on s'assure que leur attribut <i>colorShift</i> est différent avec la méthode <i>getColorShift</i> de <i>NoeudAbstrait</i> .	

Cas de test # 14	
Méthodes testées	VisiteurAgrandissement ::traiter(testAgrandissement) NœudAbstrait ::obtenirAgrandissement(testAgrandissementMur)
Justification	
Ce test vérifie que les objets sélectionnés ont bien subi un agrandissement ou un rétrécissement. Il s'agit d'un requis du SRS concernant l'édition d'objets.	
Explication du cas de test	
Test du fonctionnement de <i>VisiteurAgrandissement</i> pour le cas général. On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un <i>VisiteurAgrandissement</i> en lui passant en paramètre un vecteur d'homothétie et on traite la cible avec celui-ci. On vérifie ensuite si l'attribut <i>scale_</i> de la cible (obtenue avec la méthode <i>obtenirAgrandissement</i>) a été modifiée adéquatement (multiplication scalaire avec le vecteur d'homothétie).	

Cas de test # 15	
Méthodes testées	VisiteurAgrandissement ::traiter(testAgrandissementMur) NœudAbstrait ::obtenirAgrandissement(testAgrandissementMur)
Justification	
Ce test vérifie que l'agrandissement du mur respecte le document des requis ainsi que le rapport d'élicitation.	
Explication du cas de test	
Test du fonctionnement de <i>VisiteurAgrandissement</i> dans le cas d'un mur. On initialise un arbre par défaut et on ajoute un mur à la table. On crée ensuite un <i>VisiteurAgrandissement</i> en lui passant en paramètre un vecteur d'homothétie et on traite le mur avec celui-ci. On vérifie ensuite si l'attribut <i>scale_</i> du mur (obtenue avec la méthode <i>obtenirAgrandissement</i>) a été modifiée adéquatement (seule la composante y de <i>scale_</i> est multipliée par la composante y du vecteur d'homothétie).	

Cas de test # 16	
Méthodes testées	VisiteurSelection ::traiter(testSelection) NœudAbstrait ::estSelectionne(testSelection)
Justification	
Ce test vérifie si la sélection d'objets se fait correctement. Il s'agit d'une fonctionnalité majeure puisque toutes les modifications (agrandissement, déplacement, rotation, etc.) passent par la condition de sélection.	
Explication du cas de test	
Test du fonctionnement de <i>VisiteurSelection</i> . On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un <i>VisiteurSelection</i> en lui passant en paramètre un point qui détermine la distance de la cible et le numéro identifiant du type de nœud (par la méthode <i>getNumero</i> de <i>NœudAbstrait</i> , ici l'identifiant <i>NoeudCible</i> est retourné). On traite l'arbre avec le visiteur, et le bon objet est ensuite traité. Ici, la cible doit être sélectionnée par le passage du visiteur.	

Cas de test # 17	
Méthode testée	VisiteurPossibilite ::traiter(testPossibilite) NœudAbstrait ::estImpossible(testPossibilite) NœudAbstrait ::assignerPositionRelative(testPossibilite)
Justification	
Ce test vérifie si le visiteur modifie correctement un attribut des nœuds sélectionnés.	
Explication du cas de test	
Test du fonctionnement de <i>VisiteurPossibilite</i> . On initialise un arbre par défaut et on teste d'abord que tous les nœuds enfants de la table se trouvent dans les limites de celle-ci. Puis, on ajoute une cible et on lui assigne la position {1000.0, 1000.0, 1000.0}, qui se trouve en dehors de la table. On visite ensuite l'arbre avec le visiteur pour confirmer que l'attribut <i>impossible_</i> (<i>NoeudAbstrait</i>) de la cible est maintenant à <i>true</i> .	

Cas de test # 18	
Méthode testée	VisiteurDeplacement ::traiter(testDeplacement)
Justification	
Le déplacement des objets sélectionnés figure parmi les requis les plus importants du	

SRS concernant l'édition d'objets. Ce test permet de vérifier que le déplacement simple et multiple fonctionnent parfaitement.

Explication du cas de test

Test du fonctionnement de *VisiteurDeplacement*. On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un *VisiteurDeplacement* en lui passant en paramètre un vecteur de déplacement et on traite la cible avec celui-ci. On vérifie ensuite si la position relative de la cible (obtenue avec la méthode *obtenirPositionRelative*) a été modifiée adéquatement (addition avec le vecteur de déplacement).