Suite de cas de test # 1			
Classe testée	Utilitaire.h	**Branche**	master
Justification			
La classe utilitaire contient de nombreuses méthodes qui sont peuvent être utilisées partout dans le code dans le but de faciliter les calculs et d'augmenter la cohésion de 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 tel que prévu.			

Cas de test # 1	
Méthode testée	Utilitaire :: intersectionDeuxSegments (testIntersection)
Justification	
Explication du cas de test	
Cette méthode prend en paramètre 4 points de l'espace, forme deux segments par paire de points et return *true* si les deux droites s'interceptent. Il suffit de donner des points qui forment des droites qui s'interceptent nécessairement et s'assurer que la méthode fonctionne.	

Suite de cas de test # 2			
Classe testée	NoeudComposite.h	**Branche**	master
Justification			
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.

Puisque NoeudComposite est une classe abstraite, nous utilisons la classe NoeudCible, puisque celle-ci dérive directement de NoeudComposite.

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'assurer que c'est bel et bien le cas.	
Explication du cas de test	
Ce test permet de s'assurer que les objets de type *NœudsComposite* et ceux de classes qui en dérivent peuvent avoir des enfants, et que les méthodes *calculerProfondeur* et *obtenirNombreEnfants* 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	**Branche**	master

	VisiteurSelection.h VisiteurDuplication.h VisiteurDeplacement.h VisiteurPossibilite.h VisiteurSelectionInverse.h VisiteurSelectionMultiple.h VisiteurListeEnglobante.h		

Justification
La classe ArbreRenduINF2990.h, 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	ArbreRenduINF2990 :: initialiser (testDefaut) ArbreRenduINF2990 :: estDefaut(testDefaut) ArbreRenduINF2990 :: obtenirNombreEnfants(testDefaut) ArbreRenduINF2990 :: obtenirProprietes(testDefaut)
Justification	
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 *estDefaut* de la classe *ArbreRenduINF2990* 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	
Méthode testée	ArbreRenduINF2990 ::initialiserXML(testXmlInexistant)
Justification	
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 *initialiserXML* retourne false si le fichier n'existe pas, et true si le chemin est reconnu.	

Cas de test # 3

Méthodes testées	NoeudComposite::chercher(testPortails) NoeudAbstrait ::getTwin(testPortails) NoeudAbstrait ::obtenirType(testPortails)
Justification	
Explication du cas de test	
On s'assure que deux portails créés sont liées entre eux par l'attribut *twin_*. Une carte de jeu contenant deux portails est chargée, puis on cherche un portail dans l'arbre et on s'assure qu'il possède un nœud *twin* de type Portail.	

Cas de test # 4	
Méthode testée	ArbreRenduINF2990 :: creerNoeud(creerNoeudParUsine) UsineAbstraite ::creerNoeud(creerNoeudParUsine)
Justification	
Explication du cas de test	
On crée d'abord un nœud de type non reconnu avec la méthode *creerNoeud*, et on s'assure que la valeur retournée est un nullptr. Puis, on crée un nœud de type connu (*NœudCible*), 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	
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	
Explication du cas de test	
On s'assure qu'une table n'est pas sélectionnable, c'est-à-dire que l'attribut *selectionnable_* est défini à false à l'initialisation de l'arbre. La méthode *estSelectionnable* de la classe *NoeudAbstrait* 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	
Explication du cas de test	
Test du fonctionnement du *VisiteurDeselectionnerTout*. On initialise un arbre par défaut, on sélectionne un objet au hasard avec la méthode assignerSelection, 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	
Explication du cas de test	
Test du fonctionnement de *VisiteurRotationPoint* et de *VisiteurCentreDeMasse*. On initialise un arbre par défaut, on choisit un objet quelconque et on obtient sa valeur de rotation par la méthode *obtenirRotation* de *NoeudAbstrait*. Puis, on visite l'objet avec *VisiteurCentreDeMasse* pour obtenir le centre de la rotation, et on crée un vecteur de rotation arbitraire qui sera passé au *VisiteurRotationPoint* 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	VisiteurSelectionMultiple ::traiter(testSelectionMultiple) VisiteurSelectionMultiple :: obtenirNbObjetsSelectionne(testSelectionMultiple)
Justification	
Explication du cas de test	
Test du fonctionnement de *VisiteurSelectionMultiple*. On initialise un arbre par défaut et on ajoute deux cibles à la table. On crée ensuite un *VisiteurSelectionMultiple* en lui passant en paramètre les points pour créer une boîte de sélection de taille suffisante et on traite l'arbre. La méthode *obtenirNbObjetsSelectionne* du visiteur doit retourner le bon nombre d'objets sélectionnés.	

Cas de test # 10

Méthode testée	ArbreRenduINF2990 ::getEnfant(testSelectionInverse) VisiteurSelectionInverse ::traiter(testSelectionInverse)
Justification	
Explication du cas de test	
Test du fonctionnement de *VisiteurSelectionInverse*. On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un *VisiteurSelectionInverse* 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 *getNumero* de *NœudAbstrait*, ici l'identifiant *NoeudCible* 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	
Explication du cas de test	
Test du fonctionnement de *VisiteurDuplication*. On initialise un arbre par défaut et on crée ensuite un *VisiteurDuplication* et on traite l'arbre. Le nombre d'enfants total de la racine obtenu par la méthode *obtenirNombreEnfants* 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	
Explication du cas de test	
Test du fonctionnement de *VisiteurListeEnglobante*. On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un *VisiteurListeEnglobante* 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 boite englobante calculés par le visiteur correspondent à ceux obtenus par la méthode *obtenirVecteursBoite* de *NoeudAbstrait*.	

Cas de test # 13	
Méthode testée	NœudAbstrait ::getColorShift(testPalettes)
Justification	
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 droit) de joueurs différent est chargée, puis on cherche ces deux palettes avec la méthode *chercher* de *NoeudComposite* et on s'assure que leur attribut *colorShift* et différent avec la méthode getColorShift de *NoeudAbstrait*.	

Cas de test # 14	
Méthodes testées	VisiteurAgrandissement ::traiter(testAgrandissement) NœudAbstrait ::obtenirAgrandissement(testAgrandissementMur)
Justification	
Explication du cas de test	
Test du fonctionnement de *VisiteurAgrandissement* pour le cas général. On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un *VisiteurAgrandissement* en lui passant en paramètre un vecteur d'homothétie et on traite la cible avec celui-ci. On vérifie ensuite si l'attribut scale_ de la cible (obtenue avec la méthode *obtenirAgrandissement*) a été modifiée adéquatement (multiplication scalaire avec le vecteur d'homothétie).	

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

Cas de test # 16	
Méthodes testées	VisiteurSelection ::traiter(testSelection) NœudAbstrait ::estSelectionne(testSelection)
Justification	
Explication du cas de test	
Test du fonctionnement de *VisiteurSelection*. On initialise un arbre par défaut et on ajoute une cible à la table. On crée ensuite un *VisiteurSelection* 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 *getNumero* de *NœudAbstrait*, ici l'identifiant *NoeudCible* 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	
Explication du cas de test	
Test du fonctionnement de *VisiteurPossibilite*. 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 *impossible_* (*NoeudAbstrait*) de la cible est maintenant à *true*.	

Cas de test # 18	
Méthode testée	VisiteurDeplacement ::traiter(testDeplacement)
Justification	
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).	