

INF2990

Projet de logiciel graphique interactif

Tests logiciels

Version 14.1

Auteur original : Julien Gascon-Samson

Éditeur : Olivier Gendreau

# 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 tests choisis parmi les différents jeux de tests, ainsi que les tableaux remplis;
* la **qualité** de l’implémentation des cas de tests.

## Choix des cas de tests

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

## Implémentation des cas de tests

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

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #1*** | | |
| **Classe :** | FacadeModele (ConfigSceneTest) | |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | Nous testons les méthodes de la façade au modèle permettant de sauvegarder la configuration d’une zone de jeu. Cela nous permet de nous assurer que la configuration sera conforme en tout temps et qu’elle peut être restaurée lors du chargement d’une carte. | |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | testSauvegardeChargement : Nous testons l’enregistrement et la restauration de valeurs de configuration valides. | FacadeModele::configurationCouranteconfigTest;  configTest.frequenceAst\_=0.5;  configTest.accelerationBon\_=20;  configTest.nombreSta\_=10;  configTest.difficulteCarte\_=10;  FacadeModele::obtenirInstance()->assignerConfiguration(configTest);  // On sauvegarde le XML.  FacadeModele::obtenirInstance()->enregistrerConfiguration(**"ConfigSceneTest.xml"**);  // On remet les valeurs par défaut  ConfigScene::freqAsteroides\_=1;  ConfigScene::accBonus\_=0;  ConfigScene::nbrStations\_=1;  ConfigScene::difficulte\_=9000;  // On charge le XML  FacadeModele::obtenirInstance()->chargerConfiguration(**"ConfigSceneTest.xml"**);  // On vérifie si les valeurs de test sont celles qui ont bien été chargées  CPPUNIT\_ASSERT(ConfigScene::freqAsteroides\_==0.5);  CPPUNIT\_ASSERT(ConfigScene::accBonus\_==20);  CPPUNIT\_ASSERT(ConfigScene::nbrStations\_==10);  CPPUNIT\_ASSERT(ConfigScene::difficulte\_==10);  // On réaffecte les valeurs par défaut  configTest.frequenceAst\_=1;  configTest.accelerationBon\_=0;  configTest.nombreSta\_=1;  //On laisse la difficulté à une valeur correcte pour pouvoir enregistrer.  configTest.difficulteCarte\_=10;  // On resauvegarde le XML  FacadeModele::obtenirInstance()->enregistrerConfiguration(**"ConfigSceneTest.xml"**); |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testSauvegardeInvalide : Nous testons l’échec d’enregistrement en cas de valeurs de configuration invalides. Nous assumons qu’une carte sauvegardée, si chargée aura nécessairement des valeurs de configuration valides. | FacadeModele::configurationCouranteconfigTest;  configTest.frequenceAst\_=9000;//valeur incorrecte  // Il faut quand même instancier les autres valeurs, sinon elles sont toutes incorrectes.  configTest.accelerationBon\_=0;  configTest.nombreSta\_=1;  configTest.difficulteCarte\_=10;  FacadeModele::obtenirInstance()->assignerConfiguration(configTest);  CPPUNIT\_ASSERT(false==FacadeModele::obtenirInstance()->enregistrerConfiguration(**"ConfigSceneFailTest.xml"**));  // Restauration de la valeur testée  configTest.frequenceAst\_=1;  // Mise à jour de la seconde valeur.  configTest.accelerationBon\_=-1;  FacadeModele::obtenirInstance()->assignerConfiguration(configTest);  CPPUNIT\_ASSERT(false==FacadeModele::obtenirInstance()->enregistrerConfiguration(**"ConfigSceneFailTest.xml"**));  // Restauration de la valeur testée  configTest.accelerationBon\_=0;  // Mise à jour de la troisième valeur.  configTest.nombreSta\_=9000;  FacadeModele::obtenirInstance()->assignerConfiguration(configTest);  CPPUNIT\_ASSERT(false==FacadeModele::obtenirInstance()->enregistrerConfiguration(**"ConfigSceneFailTest.xml"**));  // Restauration de la valeur testée  configTest.nombreSta\_=1;  // Mise à jour de la dernière valeur.  configTest.difficulteCarte\_=9000;  FacadeModele::obtenirInstance()->assignerConfiguration(configTest);  CPPUNIT\_ASSERT(false==FacadeModele::obtenirInstance()->enregistrerConfiguration(**"ConfigSceneFailTest.xml"**)); |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #2*** | | |
| **Classe :** | VisiteurEnregistrer (VisiteurEnregistrerTest) | |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | Nous vérifions que l’enregistrement par le visiteur ne corrompt pas les nœuds. C’est important puisque la carte de ne sera pas sauvée correctement si ce visiteur ne fonctionne pas. | |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | testSauvegardeStation : Nous testons la sauvegarde du nœud le plus vital de tout le jeu. Nous sommes plus particulièrement intéressés par les propriétés graphiques de la station, puisque nous nous occuperons de la gestion plus spécifique ultérieurement. Aussi, nous ne testons ni le biais, ni les points de vie puisque le type réel de la station est résolu en cours de compilation, mais affiche une erreur lors des tests unitaires et les static cast ne nous semblaient pas rigoureux. | TiXmlDocumentdocTest;  arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[5])->setTixmlRoot(docTest);    NoeudAbstrait\*stationInit=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[STATION]);  //L'enregistrement se fait jusqu'à 4 décimales seulement et est arrondi.  stationInit->assignerPositionRelative(Vecteur3(-30.1234,-30.2000,30.6005));  stationInit->setRotAngle(-666.6666);  stationInit->setGrandissement(30.1);  //Enregistrement du noeud.  stationInit->accepterVisiteur(\*arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[5]));  docTest.SaveFile(**"TestSauvegardeStation.xml"**);  //Chargement du noeud.  TiXmlDocumentnouvDocument(**"TestSauvegardeStation.xml"**);  nouvDocument.LoadFile();  TiXmlHandlehandle(&nouvDocument);  TiXmlElement\*beel;  TiXmlElement\*chargement=handle.FirstChildElement(**"Type"**).Element();  arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[6])->setTixmlRoot(\*chargement);  NoeudAbstrait\*staCharge=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[STATION]);  staCharge->accepterVisiteur(\*arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[6]));  CPPUNIT\_ASSERT(stationInit->obtenirType()==staCharge->obtenirType());  CPPUNIT\_ASSERT(stationInit->obtenirParent()==staCharge->obtenirParent());  //Position relative  //x  CPPUNIT\_ASSERT((stationInit->obtenirPositionRelative()[0]+erreurToleree)>staCharge->obtenirPositionRelative()[0]);  CPPUNIT\_ASSERT((stationInit->obtenirPositionRelative()[0]-erreurToleree)<staCharge->obtenirPositionRelative()[0]);  //y  CPPUNIT\_ASSERT((stationInit->obtenirPositionRelative()[1]+erreurToleree)>staCharge->obtenirPositionRelative()[1]);  CPPUNIT\_ASSERT((stationInit->obtenirPositionRelative()[1]-erreurToleree)<staCharge->obtenirPositionRelative()[1]);  //z  CPPUNIT\_ASSERT((stationInit->obtenirPositionRelative()[2]+erreurToleree)>staCharge->obtenirPositionRelative()[2]);  CPPUNIT\_ASSERT((stationInit->obtenirPositionRelative()[2]-erreurToleree)<staCharge->obtenirPositionRelative()[2]);  //angle  CPPUNIT\_ASSERT((stationInit->getRotAngle()+erreurToleree)>staCharge->getRotAngle());  CPPUNIT\_ASSERT((stationInit->getRotAngle()-erreurToleree)<staCharge->getRotAngle());  //agrandissement  CPPUNIT\_ASSERT((stationInit->getGrandissement()+erreurToleree)>staCharge->getGrandissement());  CPPUNIT\_ASSERT((stationInit->getGrandissement()-erreurToleree)<staCharge->getGrandissement());    //On efface les noeuds  arbreCopie\_->vider(); |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testSauvegardeMultiple : Nous testons la sauvegarde de plusieurs nœuds. Tous les nœuds sauvegardés sont enregistrés correctement et leurs valeurs de base, soit la position, l’angle de rotation et le facteur d’agrandissement ne sont pas altérés malgré le grand volume d’information. | TiXmlDocumentdocTest;    NoeudAbstrait\*bar=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[1]);  NoeudAbstrait\*bon=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[2]);  NoeudAbstrait\*dep=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[3]);  NoeudAbstrait\*sta=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[5]);  NoeudAbstrait\*vor\_1=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[7]);  NoeudAbstrait\*vor\_2=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[7]);  arbreCopie\_->lieVortex();  bar->assignerPositionRelative(Vecteur3(10,20,30));  bon->assignerPositionRelative(Vecteur3(40,50,60));  sta->assignerPositionRelative(Vecteur3(-10,-20,-30));  vor\_1->assignerPositionRelative(Vecteur3(60,0,0));  vor\_2->assignerPositionRelative(Vecteur3(-60,0,0));  arbreCopie\_->ajouter(bar);  arbreCopie\_->ajouter(bon);  arbreCopie\_->ajouter(dep);  arbreCopie\_->ajouter(sta);  arbreCopie\_->ajouter(vor\_1);  arbreCopie\_->ajouter(vor\_2);  //enregistrement ordonné  for(inti=0;i<arbreCopie\_->getEnfants().size();++i)  {  TiXmlElement\*nouvNoeud=newTiXmlElement(**"node"**);  arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[5])->setTixmlRoot(\*nouvNoeud);  arbreCopie\_->getEnfants()[i]->accepterVisiteur(\*arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[5]));  docTest.LinkEndChild(nouvNoeud);  }  docTest.SaveFile(**"TestSauvegardeMultiple.xml"**);    //chargement ordonné  TiXmlDocumentnouvDocument(**"TestSauvegardeMultiple.xml"**);  nouvDocument.LoadFile();  TiXmlHandlehandle(&nouvDocument);  TiXmlElement\*beel;  TiXmlElement\*chargement=handle.FirstChildElement(**"node"**).Element();  for(chargement;chargement;chargement=chargement->NextSiblingElement())  {  beel=chargement->FirstChildElement(**"Type"**);  constchar\*nomNoeudQuery=beel->Attribute(**"type"**);  //On crée un noeud du bon type  NoeudAbstrait\*nouvNoeud=arbreCopie\_->creerNoeud(nomNoeudQuery);  if(nomNoeudQuery==NULL)  return;  //On passe le handle  arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[6])->setTixmlRoot(\*beel);  //On le traite  nouvNoeud->accepterVisiteur(\*arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[6]));  //On ajoute l'élément à l'arbre de rendu  arbreCopie\_->ajouter(nouvNoeud);  }  //Le vecteur d'enfants. On a deux fois la même chose.  vector<NoeudAbstrait\*>vecEnfants=arbreCopie\_->getEnfants();  CPPUNIT\_ASSERT(vecEnfants.size()==12);    CPPUNIT\_ASSERT(vecEnfants[6]->obtenirType()==bar->obtenirType());  CPPUNIT\_ASSERT(vecEnfants[6]->obtenirPositionRelative()==bar->obtenirPositionRelative());  CPPUNIT\_ASSERT(vecEnfants[6]->getRotAngle()==bar->getRotAngle());  CPPUNIT\_ASSERT(vecEnfants[6]->getGrandissement()==bar->getGrandissement());  CPPUNIT\_ASSERT(vecEnfants[7]->obtenirType()==bon->obtenirType());  CPPUNIT\_ASSERT(vecEnfants[7]->obtenirPositionRelative()==bon->obtenirPositionRelative());  CPPUNIT\_ASSERT(vecEnfants[7]->getRotAngle()==bon->getRotAngle());  CPPUNIT\_ASSERT(vecEnfants[7]->getGrandissement()==bon->getGrandissement());  CPPUNIT\_ASSERT(vecEnfants[8]->obtenirType()==dep->obtenirType());  CPPUNIT\_ASSERT(vecEnfants[8]->obtenirPositionRelative()==dep->obtenirPositionRelative());  CPPUNIT\_ASSERT(vecEnfants[8]->getRotAngle()==dep->getRotAngle());  CPPUNIT\_ASSERT(vecEnfants[8]->getGrandissement()==dep->getGrandissement());  CPPUNIT\_ASSERT(vecEnfants[9]->obtenirType()==sta->obtenirType());  CPPUNIT\_ASSERT(vecEnfants[9]->obtenirPositionRelative()==sta->obtenirPositionRelative());  CPPUNIT\_ASSERT(vecEnfants[9]->getRotAngle()==sta->getRotAngle());  CPPUNIT\_ASSERT(vecEnfants[9]->getGrandissement()==sta->getGrandissement());  CPPUNIT\_ASSERT(vecEnfants[10]->obtenirType()==vor\_1->obtenirType());  CPPUNIT\_ASSERT(vecEnfants[10]->obtenirPositionRelative()==vor\_1->obtenirPositionRelative());  CPPUNIT\_ASSERT(vecEnfants[10]->getRotAngle()==vor\_1->getRotAngle());  CPPUNIT\_ASSERT(vecEnfants[10]->getGrandissement()==vor\_1->getGrandissement());  CPPUNIT\_ASSERT(vecEnfants[11]->obtenirType()==vor\_2->obtenirType());  CPPUNIT\_ASSERT(vecEnfants[11]->obtenirPositionRelative()==vor\_2->obtenirPositionRelative());  CPPUNIT\_ASSERT(vecEnfants[11]->getRotAngle()==vor\_2->getRotAngle());  CPPUNIT\_ASSERT(vecEnfants[11]->getGrandissement()==vor\_2->getGrandissement());    arbreCopie\_->vider(); |
| **Cas test #3 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testSauvegardeAsteroide : Nous vérifions que l’enregistrement d’un type de nœud non-enregistrable fait en sorte que rien n’est ajouté au fichier xml. Pour nous en assurer, nous vérifions que la sauvegarde dans un document vide laisse ce dernier sans bornes d’attribut. | TiXmlDocumentdocTest;  arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[5])->setTixmlRoot(docTest);  NoeudAbstrait\*ast=arbreCopie\_->creerNoeud(NOMS\_NOEUDS[0]);  arbreCopie\_->ajouter(ast);  ast->accepterVisiteur(\*arbreCopie\_->getVisiteur(ArbreRenduINF2990::TYPE\_VISITEUR[5]));  docTest.SaveFile(**"SauvegardeAsteroideTest.xml"**);  TiXmlDocumentnouvDocument(**"SauvegardeAsteroideTest.xml"**);  nouvDocument.LoadFile();  // Un document vide ne devrait pas avoir de premier élément. Une sauvegarde laisse un minimum de 4 éléments.  TiXmlElement\*bidon=nouvDocument.FirstChildElement();  // On vérifie que le document est bel et bien vide.  CPPUNIT\_ASSERT(bidon==NULL);    arbreCopie\_->vider(); |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #3*** | | |
| **Classe :** | VisiteurDeplacement(VisiteurDeplacementTest) | |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | l est important de tester le VisiteurDeplacement, car cette classe est à la base de de l'outil du mode édition : le déplacement. Pour faire ce teste on vérifiera si la positon des objets pouvant être déplacer est la bonne et si la position des objets ne pouvant pas être déplacer reste la même. | |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | TestTraiterAsteroide :  Dans ce test, on vérifie que autant pour une valeur positive ou négative de déplacement, l'objet ne bouge pas, car il ne doit pas pouvoir être utilisé par cette outil. | void VisiteurDeplacementTest::testTraiterAsteroide()  {  NoeudAbstrait\* asteroide = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[0]);  // cas 1: valeur de deplacement positif  asteroide->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = 50;  dy\_ = 50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  asteroide->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(asteroide->obtenirPositionRelative() == Vecteur3(25,25,0));  // cas 2: valeur de deplacement negatif  asteroide->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = -50;  dy\_ = -50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  asteroide->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(asteroide->obtenirPositionRelative() == Vecteur3(25,25,0));  } |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | TestTraiterBarriere :  Dans ce test, on vérifie que autant pour une valeur positive ou négative de déplacement, l'objet bouge correctement. | void VisiteurDeplacementTest::testTraiterBarriere()  {  NoeudAbstrait\* barriere = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[1]);  // cas 1: valeur de deplacement positif  barriere->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = 50;  dy\_ = 50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  barriere->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(barriere->obtenirPositionRelative() == Vecteur3(75,75,0));  // cas 2: valeur de deplacement negatif  barriere->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = -50;  dy\_ = -50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  barriere->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(barriere->obtenirPositionRelative() == Vecteur3(-25,-25,0));  } |
| **Cas test #3 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | TestTraiterBonus :  Dans ce test, on vérifie que autant pour une valeur positive ou négative de déplacement, l'objet bouge correctement. | void VisiteurDeplacementTest::testTraiterBonus()  {  NoeudAbstrait\* bonus = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[2]);  // cas 1: valeur de deplacement positif  bonus->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = 50;  dy\_ = 50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  bonus->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(bonus->obtenirPositionRelative() == Vecteur3(75,75,0));  // cas 2: valeur de deplacement negatif  bonus->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = -50;  dy\_ = -50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  bonus->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(bonus->obtenirPositionRelative() == Vecteur3(-25,-25,0));  } |
| **Cas test #4 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterDepart :  Dans ce test, on vérifie que autant pour une valeur positive ou négative de déplacement, l'objet bouge correctement. | NoeudAbstrait\* depart = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[3]);  // cas 1: valeur de deplacement positif  depart->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = 50;  dy\_ = 50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  depart->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(depart->obtenirPositionRelative() == Vecteur3(75,75,0));  // cas 2: valeur de deplacement negatif  depart->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = -50;  dy\_ = -50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  depart->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(depart->obtenirPositionRelative() == Vecteur3(-25,-25,0));  } |
| **Cas test #5 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterLaser :  Dans ce test, on vérifie que autant pour une valeur positive ou négative de déplacement, l'objet ne bouge pas, car il ne doit pas pouvoir être utilisé par cette outil. | void VisiteurDeplacementTest::testTraiterLaser()  {  NoeudAbstrait\* laser = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[4]);  // cas 1: valeur de deplacement positif  laser->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = 50;  dy\_ = 50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  laser->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(laser->obtenirPositionRelative() == Vecteur3(25,25,0));  // cas 2: valeur de deplacement negatif  laser->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = -50;  dy\_ = -50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  laser->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(laser->obtenirPositionRelative() == Vecteur3(25,25,0));  } |
| **Cas test #6 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterStation :  Dans ce test, on vérifie que autant pour une valeur positive ou négative de déplacement, l'objet bouge correctement. | void VisiteurDeplacementTest::testTraiterStation()  {  NoeudAbstrait\* station = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[5]);  // cas 1: valeur de deplacement positif  station->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = 50;  dy\_ = 50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  station->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(station->obtenirPositionRelative() == Vecteur3(75,75,0));  // cas 2: valeur de deplacement negatif  station->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = -50;  dy\_ = -50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  station->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(station->obtenirPositionRelative() == Vecteur3(-25,-25,0));  } |
| **Cas test #7 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterVaisseau :  Dans ce test, on vérifie que autant pour une valeur positive ou négative de déplacement, l'objet ne bouge pas, car il ne doit par pouvoir être utilisé pas cette outil. | void VisiteurDeplacementTest::testTraiterVaisseau()  {  NoeudAbstrait\* vaisseau = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[6]);  // cas 1: valeur de deplacement positif  vaisseau->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = 50;  dy\_ = 50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  vaisseau->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(vaisseau->obtenirPositionRelative() == Vecteur3(25,25,0));  // cas 2: valeur de deplacement negatif  vaisseau->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = -50;  dy\_ = -50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  vaisseau->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(vaisseau->obtenirPositionRelative() == Vecteur3(25,25,0));  } |
| **Cas test #8 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterVortex :  Dans ce test, on vérifie que autant pour une valeur positive ou négative de déplacement, l'objet bouge correctement. | void VisiteurDeplacementTest::testTraiterVortex()  {  NoeudAbstrait\* vortex1 = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[7]);  NoeudAbstrait\* vortex2 = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[7]);  // cas 1: valeur de deplacement positif  vortex1->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = 50;  dy\_ = 50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  vortex1->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(vortex1->obtenirPositionRelative() == Vecteur3(75,75,0));  // cas 2: valeur de deplacement negatif  vortex1->assignerPositionRelative(Vecteur3(25,25,0));  dx\_ = -50;  dy\_ = -50;  arbreCopie\_->getVisiteur("deplacer")->setDiff(dx\_, dy\_);  vortex1->accepterVisiteur(\*arbreCopie\_->getVisiteur("deplacer"));  CPPUNIT\_ASSERT(vortex1->obtenirPositionRelative() == Vecteur3(-25,-25,0));  } |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #4*** | | |
| **Classe :** | VisiteurDupliquer(VisiteurDupliquerTest) | |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | Il est important de tester le VisiteurDupliquer, car cette classe est à la base de de l'outil du mode édition : la duplication. Pour faire ce teste on vérifiera si un objet est créé contenant les mêmes caractéristiques qu'un objet pouvant être dupliqué et ne crée rien si l'objet ne peut pas être dupliqué. | |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | TestTraiterAsteroide :  Dans ce test, on vérifie qu'aucun objet ne sera créé, car il ne doit pas pouvoir être utilisé par cette outil. | void VisiteurDupliquerTest::testTraiterAsteroide()  {  NoeudAbstrait\* asteroide = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[0]);  asteroide->assignerPositionRelative(Vecteur3(25,25,0));  asteroide->setGrandissement(1.5);  asteroide->setRotAngle(90);  dx\_ = 50;  dy\_ = 50;  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  asteroide->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vector<NoeudAbstrait\*>duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size() == 0);  for(int i=0; i<duplicat.size();i++)  duplicat[i]->setDuplique(false);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->vider();  } |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | TestTraiterBarriere :  Dans ce test, on s'assure qu'un objet a été créé, qu'il a les mêmes caractéristiques que l'objet à dupliquer(angle de rotation, facteur de grandissement, type) et que la position du nouvel objet est la bonne. | void VisiteurDupliquerTest::testTraiterBarriere()  {  NoeudAbstrait\* barriere = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[1]);  barriere->assignerPositionRelative(Vecteur3(25,25,0));  barriere->setGrandissement(1.5);  barriere->setRotAngle(90);  dx\_ = 50;  dy\_ = 50;  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  barriere->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vector<NoeudAbstrait\*>duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size()>0);  CPPUNIT\_ASSERT(duplicat[0]->obtenirPositionRelative() == Vecteur3(75,75,0));  CPPUNIT\_ASSERT(duplicat[0]->getGrandissement() == 1.5);  CPPUNIT\_ASSERT(duplicat[0]->getRotAngle() == 90);  CPPUNIT\_ASSERT(duplicat[0]->obtenirType() == barriere->obtenirType());  for(int i=0; i<duplicat.size();i++)  duplicat[i]->setDuplique(false);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->vider();  } |
| **Cas test #3 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | TraiterBonus :  Dans ce test, on s'assure qu'un objet a été créé, qu'il a les mêmes caractéristiques que l'objet à dupliquer(angle de rotation, facteur de grandissement, type) et que la position du nouvel objet est la bonne. | void VisiteurDupliquerTest::testTraiterBonus()  {  NoeudAbstrait\* bonus = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[2]);  bonus->assignerPositionRelative(Vecteur3(25,25,0));  bonus->setGrandissement(1.5);  bonus->setRotAngle(90);  dx\_ = 50;  dy\_ = 50;  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  bonus->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vector<NoeudAbstrait\*>duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size()>0);  CPPUNIT\_ASSERT(duplicat[0]->obtenirPositionRelative() == Vecteur3(75,75,0));  CPPUNIT\_ASSERT(duplicat[0]->getGrandissement() == 1.5);  CPPUNIT\_ASSERT(duplicat[0]->getRotAngle() == 90);  CPPUNIT\_ASSERT(duplicat[0]->obtenirType() == bonus->obtenirType());  for(int i=0; i<duplicat.size();i++)  duplicat[i]->setDuplique(false);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->vider();  } |
| **Cas test #4 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterDepart :  Dans ce test, on vérifie qu'aucun objet ne sera créé, car il ne doit pas pouvoir être utilisé par cette outil. | void VisiteurDupliquerTest::testTraiterDepart()  {  NoeudAbstrait\* depart = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[3]);  depart->assignerPositionRelative(Vecteur3(25,25,0));  depart->setGrandissement(1.5);  depart->setRotAngle(90);  dx\_ = 50;  dy\_ = 50;  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  depart->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vector<NoeudAbstrait\*>duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size() == 0);  for(int i=0; i<duplicat.size();i++)  duplicat[i]->setDuplique(false);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->vider();  } |
| **Cas test #5 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterLaser :  Dans ce test, on vérifie qu'aucun objet ne sera créé, car il ne doit pas pouvoir être utilisé par cette outil. | void VisiteurDupliquerTest::testTraiterLaser()  {  NoeudAbstrait\* laser = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[4]);  laser->assignerPositionRelative(Vecteur3(25,25,0));  laser->setGrandissement(1.5);  laser->setRotAngle(90);  dx\_ = 50;  dy\_ = 50;  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  laser->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vector<NoeudAbstrait\*>duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size() == 0);  for(int i=0; i<duplicat.size();i++)  duplicat[i]->setDuplique(false);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->vider();  } |
| **Cas test #6 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterStation :  Dans ce test, on s'assure qu'un objet a été créé, qu'il a les mêmes caractéristiques que l'objet à dupliquer(angle de rotation, facteur de grandissement, type) et que la position du nouvel objet est la bonne. | void VisiteurDupliquerTest::testTraiterStation()  {  NoeudAbstrait\* station = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[5]);  station->assignerPositionRelative(Vecteur3(25,25,0));  station->setGrandissement(1.5);  station->setRotAngle(90);  dx\_ = 50;  dy\_ = 50;  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  station->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vector<NoeudAbstrait\*>duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size()>0);  CPPUNIT\_ASSERT(duplicat[0]->obtenirPositionRelative() == Vecteur3(75,75,0));  CPPUNIT\_ASSERT(duplicat[0]->getGrandissement() == 1.5);  CPPUNIT\_ASSERT(duplicat[0]->getRotAngle() == 90);  CPPUNIT\_ASSERT(duplicat[0]->obtenirType() == station->obtenirType());  for(int i=0; i<duplicat.size();i++)  duplicat[i]->setDuplique(false);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->vider();  } |
| **Cas test #7 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterVaisseau :  Dans ce test, on vérifie qu'aucun objet ne sera créé, car il ne doit pas pouvoir être utilisé par cette outil. | void VisiteurDupliquerTest::testTraiterVaisseau()  {  NoeudAbstrait\* vaisseau = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[6]);  vaisseau->assignerPositionRelative(Vecteur3(25,25,0));  vaisseau->setGrandissement(1.5);  vaisseau->setRotAngle(90);  dx\_ = 50;  dy\_ = 50;  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  vaisseau->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vector<NoeudAbstrait\*>duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size() == 0);  for(int i=0; i<duplicat.size();i++)  duplicat[i]->setDuplique(false);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->vider();  } |
| **Cas test #8 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | testTraiterVortex  Dans ce test, on s'assure que si les deux sont sélectionnés, deux objets ont été créés, qu'ils aient les mêmes caractéristiques que les objets à dupliquer(angle de rotation, facteur de grandissement, type) et que les positions des nouveaux objets sont les bonnes. Sinon, aucun objet ne sera créé. | void VisiteurDupliquerTest::testTraiterVortex()  {  NoeudAbstrait\* vortex1 = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[7]);  NoeudAbstrait\* vortex2 = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->creerNoeud(NOMS\_NOEUDS[7]);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->ajouter(vortex1);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->ajouter(vortex2);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->lieVortex();  vortex1->assignerPositionRelative(Vecteur3(25,25,0));  vortex2->assignerPositionRelative(Vecteur3(-25,-25,0));  vortex1->setGrandissement(1.5);  vortex2->setGrandissement(1.75);  vortex1->setRotAngle(90);  vortex2->setRotAngle(-90);  dx\_ = 50;  dy\_ = 50;  // cas 1: seulement un vortex est selectionne et on tente de le dupliquer  vortex1->assignerSelection(true);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  vortex1->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vector<NoeudAbstrait\*>duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size() == 0);  vortex1->assignerSelection(false);  // cas 2: les deux vortex sont selctionnes et on tente de les dupliquer  vortex1->assignerSelection(true);  vortex2->assignerSelection(true);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer")->setDiff(dx\_, dy\_);  vortex1->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  vortex2->accepterVisiteur(\*FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getVisiteur("dupliquer"));  duplicat = FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->getEnfantsADupliquer();  CPPUNIT\_ASSERT(duplicat.size()>0);  CPPUNIT\_ASSERT(duplicat[0]->obtenirPositionRelative() == Vecteur3(75,75,0));  CPPUNIT\_ASSERT(duplicat[0]->getGrandissement() == 1.5);  CPPUNIT\_ASSERT(duplicat[0]->getRotAngle() == 90);  CPPUNIT\_ASSERT(duplicat[0]->obtenirType() == vortex1->obtenirType());  CPPUNIT\_ASSERT(duplicat[1]->obtenirPositionRelative() == Vecteur3(25,25,0));  CPPUNIT\_ASSERT(duplicat[1]->getGrandissement() == 1.75);  CPPUNIT\_ASSERT(duplicat[1]->getRotAngle() == -90);  CPPUNIT\_ASSERT(duplicat[1]->obtenirType() == vortex2->obtenirType());  cerr<<"duplicat.size()"<<duplicat.size();  for(int i=0; i<duplicat.size();i++)  duplicat[i]->setDuplique(false);  FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990()->vider();  } |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #5*** | | |
| **Classe :** | VisiteurRotation(VisiteurRotationTest) | |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | Il est important de tester le VisiteurRotation, car cette classe est à la base de de l'outil du mode édition : la rotation. Pour faire ce teste on vérifiera si un objet a la bonne position et la bonne angle de rotation et ne bouge pas si l'objet ne peut pas effectuer de rotation. | |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | TestTraiterAsteroide :  Dans ce test, on vérifie que l'objet ne change pas de position ou d'angle lorsque l'angle est positive, négative ou supérieur à 360 lorsque le point de pivot est le centre de l'objet ou distinct, car elle ne peut subir de rotation. | void VisiteurRotationTest::testTraiterAsteroide()  {  NoeudAbstrait\* asteroide = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[0]);  // cas 1: valeur de Rotation positif  //cas 1.1: point de pivot est le centre de l'objet  tester(asteroide , Vecteur3(25,25,0) , 90 , Vecteur3(25,25,0) , 0);  //cas 1.2: point de pivot distint  tester(asteroide , Vecteur3(25,26,0) , 90 , Vecteur3(25,25,0) , 0);  // cas 2: valeur de Rotation negatif  tester(asteroide , Vecteur3(25,25,0) , -90 , Vecteur3(25,25,0) , 0);  // cas 3: valeur de Rotation supperieur a 360  tester(asteroide , Vecteur3(25,25,0) , 450 , Vecteur3(25,25,0) , 0);  }  void VisiteurRotationTest::tester(NoeudAbstrait\* noeud, Vecteur3 centre, double angle, Vecteur3 positionPrevu, double anglePrevu)  {  noeud->assignerPositionRelative(Vecteur3(25,25,0));  noeud->setRotAngle(0);  centre\_ = centre;  angle\_ = angle;  arbreCopie\_->getVisiteur("rotation")->setAngle(angle\_);  arbreCopie\_->getVisiteur("rotation")->setDiff(centre\_[0], centre\_[1]);  noeud->accepterVisiteur(\*arbreCopie\_->getVisiteur("rotation"));  CPPUNIT\_ASSERT(noeud->obtenirPositionRelative() == positionPrevu);  CPPUNIT\_ASSERT(noeud->getRotAngle() == anglePrevu);  } |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | TestTraiterBarriere :  Dans ce test, on vérifie que l'objet ait la bonne position et le bon angle lorsque l'angle est positive, négative ou supérieur à 360 lorsque le point de pivot est le centre de l'objet ou distinct. | void VisiteurRotationTest::testTraiterBarriere()  {  NoeudAbstrait\* barriere = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[1]);  // cas 1: valeur de Rotation positif  //cas 1.1: point de pivot est le centre de l'objet  tester(barriere , Vecteur3(25,25,0) , 90 , Vecteur3(25,25,0) , 90);  //cas 1.2: point de pivot distinct  tester(barriere , Vecteur3(25,26,0) , 90 , Vecteur3(26,26,0) , 90);  // cas 2: valeur de Rotation negatif  //cas 2.1: point de pivot est le centre de l'objet  tester(barriere , Vecteur3(25,25,0) , -90 , Vecteur3(25,25,0) , -90);  //cas 2.2: point de pivot distinct  tester(barriere , Vecteur3(25,26,0) , -90 , Vecteur3(24,26,0) , -90);  // cas 3: valeur de Rotation supperieur a 360  //cas 3.1: point de pivot est le centre de l'objet  tester(barriere , Vecteur3(25,25,0) , 450 , Vecteur3(25,25,0) , 450);  //cas 3.2: point de pivot distinct  tester(barriere , Vecteur3(25,26,0) , 450 , Vecteur3(26,26,0) , 450);  }  void VisiteurRotationTest::tester(NoeudAbstrait\* noeud, Vecteur3 centre, double angle, Vecteur3 positionPrevu, double anglePrevu)  {  noeud->assignerPositionRelative(Vecteur3(25,25,0));  noeud->setRotAngle(0);  centre\_ = centre;  angle\_ = angle;  arbreCopie\_->getVisiteur("rotation")->setAngle(angle\_);  arbreCopie\_->getVisiteur("rotation")->setDiff(centre\_[0], centre\_[1]);  noeud->accepterVisiteur(\*arbreCopie\_->getVisiteur("rotation"));  CPPUNIT\_ASSERT(noeud->obtenirPositionRelative() == positionPrevu);  CPPUNIT\_ASSERT(noeud->getRotAngle() == anglePrevu);  } |
| **Cas test #3 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | TestTraiterBonus :  Dans ce test, on vérifie que l'objet ait la bonne position et le bon angle lorsque l'angle est positive, négative ou supérieur à 360 lorsque le point de pivot est le centre de l'objet ou distinct. | void VisiteurRotationTest::testTraiterBonus()  {  NoeudAbstrait\* bonus = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[2]);  // cas 1: valeur de Rotation positif  //cas 1.1: point de pivot est le centre de l'objet  tester(bonus , Vecteur3(25,25,0) , 90 , Vecteur3(25,25,0) , 90);  //cas 1.2: point de pivot distinct  tester(bonus , Vecteur3(25,26,0) , 90 , Vecteur3(26,26,0) , 90);  // cas 2: valeur de Rotation negatif  //cas 2.1: point de pivot est le centre de l'objet  tester(bonus , Vecteur3(25,25,0) , -90 , Vecteur3(25,25,0) , -90);  //cas 2.2: point de pivot distinct  tester(bonus , Vecteur3(25,26,0) , -90 , Vecteur3(24,26,0) , -90);  // cas 3: valeur de Rotation supperieur a 360  //cas 3.1: point de pivot est le centre de l'objet  tester(bonus , Vecteur3(25,25,0) , 450 , Vecteur3(25,25,0) , 450);  //cas 3.2: point de pivot distinct  tester(bonus , Vecteur3(25,26,0) , 450 , Vecteur3(26,26,0) , 450);  }  void VisiteurRotationTest::tester(NoeudAbstrait\* noeud, Vecteur3 centre, double angle, Vecteur3 positionPrevu, double anglePrevu)  {  noeud->assignerPositionRelative(Vecteur3(25,25,0));  noeud->setRotAngle(0);  centre\_ = centre;  angle\_ = angle;  arbreCopie\_->getVisiteur("rotation")->setAngle(angle\_);  arbreCopie\_->getVisiteur("rotation")->setDiff(centre\_[0], centre\_[1]);  noeud->accepterVisiteur(\*arbreCopie\_->getVisiteur("rotation"));  CPPUNIT\_ASSERT(noeud->obtenirPositionRelative() == positionPrevu);  CPPUNIT\_ASSERT(noeud->getRotAngle() == anglePrevu);  } |
| **Cas test #4 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterDepart :  Dans ce test, on vérifie que l'objet ait la bonne position et le bon angle lorsque l'angle est positive, négative ou supérieur à 360 lorsque le point de pivot est le centre de l'objet ou distinct. | void VisiteurRotationTest::testTraiterDepart()  {  NoeudAbstrait\* depart = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[3]);  // cas 1: valeur de Rotation positif  //cas 1.1: point de pivot est le centre de l'objet  tester(depart , Vecteur3(25,25,0) , 90 , Vecteur3(25,25,0) , 90);  //cas 1.2: point de pivot distinct  tester(depart , Vecteur3(25,26,0) , 90 , Vecteur3(26,26,0) , 90);  // cas 2: valeur de Rotation negatif  //cas 2.1: point de pivot est le centre de l'objet  tester(depart , Vecteur3(25,25,0) , -90 , Vecteur3(25,25,0) , -90);  //cas 2.2: point de pivot distinct  tester(depart , Vecteur3(25,26,0) , -90 , Vecteur3(24,26,0) , -90);  // cas 3: valeur de Rotation supperieur a 360  //cas 3.1: point de pivot est le centre de l'objet  tester(depart , Vecteur3(25,25,0) , 450 , Vecteur3(25,25,0) , 450);  //cas 3.2: point de pivot distinct  tester(depart , Vecteur3(25,26,0) , 450 , Vecteur3(26,26,0) , 450);  }  void VisiteurRotationTest::tester(NoeudAbstrait\* noeud, Vecteur3 centre, double angle, Vecteur3 positionPrevu, double anglePrevu)  {  noeud->assignerPositionRelative(Vecteur3(25,25,0));  noeud->setRotAngle(0);  centre\_ = centre;  angle\_ = angle;  arbreCopie\_->getVisiteur("rotation")->setAngle(angle\_);  arbreCopie\_->getVisiteur("rotation")->setDiff(centre\_[0], centre\_[1]);  noeud->accepterVisiteur(\*arbreCopie\_->getVisiteur("rotation"));  CPPUNIT\_ASSERT(noeud->obtenirPositionRelative() == positionPrevu);  CPPUNIT\_ASSERT(noeud->getRotAngle() == anglePrevu);  } |
| **Cas test #5 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterLaser :  Dans ce test, on vérifie que l'objet ne change pas de position ou d'angle lorsque l'angle est positive, négative ou supérieur à 360 lorsque le point de pivot est le centre de l'objet ou distinct, car elle ne peut subir de rotation. | void VisiteurRotationTest::testTraiterLaser()  {  NoeudAbstrait\* laser = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[4]);  // cas 1: valeur de Rotation positif  //cas 1.1: point de pivot est le centre de l'objet  tester(laser , Vecteur3(25,25,0) , 90 , Vecteur3(25,25,0) , 0);  //cas 1.2: point de pivot distint  tester(laser , Vecteur3(25,26,0) , 90 , Vecteur3(25,25,0) , 0);  // cas 2: valeur de Rotation negatif  tester(laser , Vecteur3(25,25,0) , -90 , Vecteur3(25,25,0) , 0);  // cas 3: valeur de Rotation supperieur a 360  tester(laser , Vecteur3(25,25,0) , 450 , Vecteur3(25,25,0) , 0);  }  void VisiteurRotationTest::tester(NoeudAbstrait\* noeud, Vecteur3 centre, double angle, Vecteur3 positionPrevu, double anglePrevu)  {  noeud->assignerPositionRelative(Vecteur3(25,25,0));  noeud->setRotAngle(0);  centre\_ = centre;  angle\_ = angle;  arbreCopie\_->getVisiteur("rotation")->setAngle(angle\_);  arbreCopie\_->getVisiteur("rotation")->setDiff(centre\_[0], centre\_[1]);  noeud->accepterVisiteur(\*arbreCopie\_->getVisiteur("rotation"));  CPPUNIT\_ASSERT(noeud->obtenirPositionRelative() == positionPrevu);  CPPUNIT\_ASSERT(noeud->getRotAngle() == anglePrevu);  } |
| **Cas test #6 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterStation :  Dans ce test, on vérifie que l'objet ait la bonne position et le bon angle lorsque l'angle est positive, négative ou supérieur à 360 lorsque le point de pivot est le centre de l'objet ou distinct. | void VisiteurRotationTest::testTraiterStation()  {  NoeudAbstrait\* station = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[5]);  // cas 1: valeur de Rotation positif  //cas 1.1: point de pivot est le centre de l'objet  tester(station , Vecteur3(25,25,0) , 90 , Vecteur3(25,25,0) , 90);  //cas 1.2: point de pivot distinct  tester(station , Vecteur3(25,26,0) , 90 , Vecteur3(26,26,0) , 90);  // cas 2: valeur de Rotation negatif  //cas 2.1: point de pivot est le centre de l'objet  tester(station , Vecteur3(25,25,0) , -90 , Vecteur3(25,25,0) , -90);  //cas 2.2: point de pivot distinct  tester(station , Vecteur3(25,26,0) , -90 , Vecteur3(24,26,0) , -90);  // cas 3: valeur de Rotation supperieur a 360  //cas 3.1: point de pivot est le centre de l'objet  tester(station , Vecteur3(25,25,0) , 450 , Vecteur3(25,25,0) , 450);  //cas 3.2: point de pivot distinct  tester(station , Vecteur3(25,26,0) , 450 , Vecteur3(26,26,0) , 450);  }  void VisiteurRotationTest::tester(NoeudAbstrait\* noeud, Vecteur3 centre, double angle, Vecteur3 positionPrevu, double anglePrevu)  {  noeud->assignerPositionRelative(Vecteur3(25,25,0));  noeud->setRotAngle(0);  centre\_ = centre;  angle\_ = angle;  arbreCopie\_->getVisiteur("rotation")->setAngle(angle\_);  arbreCopie\_->getVisiteur("rotation")->setDiff(centre\_[0], centre\_[1]);  noeud->accepterVisiteur(\*arbreCopie\_->getVisiteur("rotation"));  CPPUNIT\_ASSERT(noeud->obtenirPositionRelative() == positionPrevu);  CPPUNIT\_ASSERT(noeud->getRotAngle() == anglePrevu);  } |
| **Cas test #7 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TraiterVaisseau :  Dans ce test, on vérifie que l'objet ne change pas de position ou d'angle lorsque l'angle est positive, négative ou supérieur à 360 lorsque le point de pivot est le centre de l'objet ou distinct, car elle ne peut subir de rotation. | void VisiteurRotationTest::testTraiterVaisseau()  {  NoeudAbstrait\* vaisseau = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[6]);  // cas 1: valeur de Rotation positif  //cas 1.1: point de pivot est le centre de l'objet  tester(vaisseau , Vecteur3(25,25,0) , 90 , Vecteur3(25,25,0) , 0);  //cas 1.2: point de pivot distint  tester(vaisseau , Vecteur3(25,26,0) , 90 , Vecteur3(25,25,0) , 0);  // cas 2: valeur de Rotation negatif  tester(vaisseau , Vecteur3(25,25,0) , -90 , Vecteur3(25,25,0) , 0);  // cas 3: valeur de Rotation supperieur a 360  tester(vaisseau , Vecteur3(25,25,0) , 450 , Vecteur3(25,25,0) , 0);  }  void VisiteurRotationTest::tester(NoeudAbstrait\* noeud, Vecteur3 centre, double angle, Vecteur3 positionPrevu, double anglePrevu)  {  noeud->assignerPositionRelative(Vecteur3(25,25,0));  noeud->setRotAngle(0);  centre\_ = centre;  angle\_ = angle;  arbreCopie\_->getVisiteur("rotation")->setAngle(angle\_);  arbreCopie\_->getVisiteur("rotation")->setDiff(centre\_[0], centre\_[1]);  noeud->accepterVisiteur(\*arbreCopie\_->getVisiteur("rotation"));  CPPUNIT\_ASSERT(noeud->obtenirPositionRelative() == positionPrevu);  CPPUNIT\_ASSERT(noeud->getRotAngle() == anglePrevu);  } |
| **Cas test #8 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | TestTraiterVortex :  Dans ce test, on vérifie que l'objet ait la bonne position et le bon angle lorsque l'angle est positive, négative ou supérieur à 360 lorsque le point de pivot est le centre de l'objet ou distinct. | {  NoeudAbstrait\* vortex1 = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[7]);  NoeudAbstrait\* vortex2 = arbreCopie\_->creerNoeud(NOMS\_NOEUDS[7]);  // cas 1: valeur de Rotation positif  //cas 1.1: point de pivot est le centre de l'objet  tester(vortex1 , Vecteur3(25,25,0) , 90 , Vecteur3(25,25,0) , 90);  //cas 1.2: point de pivot distinct  tester(vortex1 , Vecteur3(25,26,0) , 90 , Vecteur3(26,26,0) , 90);  // cas 2: valeur de Rotation negatif  //cas 2.1: point de pivot est le centre de l'objet  tester(vortex1 , Vecteur3(25,25,0) , -90 , Vecteur3(25,25,0) , -90);  //cas 2.2: point de pivot distinct  tester(vortex1 , Vecteur3(25,26,0) , -90 , Vecteur3(24,26,0) , -90);  // cas 3: valeur de Rotation supperieur a 360  //cas 3.1: point de pivot est le centre de l'objet  tester(vortex1 , Vecteur3(25,25,0) , 450 , Vecteur3(25,25,0) , 450);  //cas 3.2: point de pivot distinct  tester(vortex1 , Vecteur3(25,26,0) , 450 , Vecteur3(26,26,0) , 450);  }  void VisiteurRotationTest::tester(NoeudAbstrait\* noeud, Vecteur3 centre, double angle, Vecteur3 positionPrevu, double anglePrevu)  {  noeud->assignerPositionRelative(Vecteur3(25,25,0));  noeud->setRotAngle(0);  centre\_ = centre;  angle\_ = angle;  arbreCopie\_->getVisiteur("rotation")->setAngle(angle\_);  arbreCopie\_->getVisiteur("rotation")->setDiff(centre\_[0], centre\_[1]);  noeud->accepterVisiteur(\*arbreCopie\_->getVisiteur("rotation"));  CPPUNIT\_ASSERT(noeud->obtenirPositionRelative() == positionPrevu);  CPPUNIT\_ASSERT(noeud->getRotAngle() == anglePrevu);  } |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #6*** | | |
| **Classe :** | ProjectionOrtho.cpp | |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | Nous testons ici la classe de gestion de la projection orthogonale. Il est intéressant de tester cette classe car la projection est un facteur important du jeu : la manipulation en est relativement complexe. De plus, bien quelques algorithmes de cette classe sont un peu complexes et promptes à causer des erreurs dues à l’inattention ou à l’incompréhension du fonctionnement de la projection. | |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | testZoomStandard : Nous testons le zoom standard, très utile pour avoir une vue appropriée. On incrémente en continu le zoom d’un certain facteur, en vérifiant que la fenêtre virtuelle résultante corresponde au changement souhaité, jusqu’à atteindre la valeur maximale (que l’on teste pour vérifier qu’elle n’est pas dépassé). Puis on fait l’inverse. | //Initialisation  double xMinFenetre, xMaxFenetre, yMinFenetre, yMaxFenetre;  int xMinCloture, xMaxCloture, yMinCloture, yMaxCloture;  projection\_->obtenirCoordonneesFenetreVirtuelle(xMinFenetre, xMaxFenetre, yMinFenetre, yMaxFenetre);  //Zoom in  while ( (xMaxFenetre - xMinFenetre) > zoomInMax\_ && (yMaxFenetre - yMinFenetre) > zoomInMax\_ )  {  //On vérifie que la clôture ne change pas  projection\_->obtenirCoordonneesCloture(xMinCloture, xMaxCloture, yMinCloture, yMaxCloture);  CPPUNIT\_ASSERT(xMinCloture == xMinClotureInitiale\_ && xMaxCloture == xMaxClotureInitiale\_ && yMinCloture == yMinClotureInitiale\_ && yMaxCloture == yMaxClotureInitiale\_);  //On vérifie que les coordonnées fenêtre diminuent correctement  projection\_->obtenirCoordonneesFenetreVirtuelle(xMinFenetre, xMaxFenetre, yMinFenetre, yMaxFenetre);  double xMinFenetreIncremente, xMaxFenetreIncremente, yMinFenetreIncremente, yMaxFenetreIncremente;  projection\_->zoomerIn();  projection\_->obtenirCoordonneesFenetreVirtuelle( xMinFenetreIncremente, xMaxFenetreIncremente, yMinFenetreIncremente, yMaxFenetreIncremente );  //On regarde si le maximum a pas été atteint  if (xMinFenetre == xMinFenetreIncremente && xMaxFenetre == xMaxFenetreIncremente && yMinFenetre == yMinFenetreIncremente && yMaxFenetre == yMaxFenetreIncremente)  {  CPPUNIT\_ASSERT( (xMaxFenetreIncremente - xMinFenetreIncremente) / incrementZoom\_ < zoomInMax\_ );  CPPUNIT\_ASSERT( (yMaxFenetreIncremente - yMinFenetreIncremente) / incrementZoom\_ < zoomInMax\_ );  break;  }  //Sinon, on vérifie que les incréments correspondent  CPPUNIT\_ASSERT( abs ( (xMaxFenetreIncremente - xMinFenetreIncremente) \* incrementZoom\_ - (xMaxFenetre - xMinFenetre) ) < erreurToleree\_);  CPPUNIT\_ASSERT( abs ( (yMaxFenetreIncremente - yMinFenetreIncremente) \* incrementZoom\_ - (yMaxFenetre - yMinFenetre) ) < erreurToleree\_);  }  //Zoom out  while ( (xMaxFenetre - xMinFenetre) < zoomOutMax\_ && (yMaxFenetre - yMinFenetre) < zoomOutMax\_ )  {  //On vérifie que la clôture ne change pas  projection\_->obtenirCoordonneesCloture(xMinCloture, xMaxCloture, yMinCloture, yMaxCloture);  CPPUNIT\_ASSERT(xMinCloture == xMinClotureInitiale\_ && xMaxCloture == xMaxClotureInitiale\_ && yMinCloture == yMinClotureInitiale\_ && yMaxCloture == yMaxClotureInitiale\_);  //On vérifie que les coordonnées fenêtre augmentnt correctement  projection\_->obtenirCoordonneesFenetreVirtuelle(xMinFenetre, xMaxFenetre, yMinFenetre, yMaxFenetre);  double xMinFenetreIncremente, xMaxFenetreIncremente, yMinFenetreIncremente, yMaxFenetreIncremente;  projection\_->zoomerOut();  projection\_->obtenirCoordonneesFenetreVirtuelle( xMinFenetreIncremente, xMaxFenetreIncremente, yMinFenetreIncremente, yMaxFenetreIncremente );  //On regarde si le minimum n'a pas été atteint  if (xMinFenetre == xMinFenetreIncremente && xMaxFenetre == xMaxFenetreIncremente && yMinFenetre == yMinFenetreIncremente && yMaxFenetre == yMaxFenetreIncremente)  {  CPPUNIT\_ASSERT( (xMaxFenetreIncremente - xMinFenetreIncremente) \* incrementZoom\_ > zoomOutMax\_ );  CPPUNIT\_ASSERT( (yMaxFenetreIncremente - yMinFenetreIncremente) \* incrementZoom\_ > zoomOutMax\_ );  break;  }  //Sinon, on vérifie que les incréments correspondent  CPPUNIT\_ASSERT( abs ( (xMaxFenetreIncremente - xMinFenetreIncremente) / incrementZoom\_ - (xMaxFenetre - xMinFenetre) ) < erreurToleree\_);  CPPUNIT\_ASSERT( abs ( (yMaxFenetreIncremente - yMinFenetreIncremente) / incrementZoom\_ - (yMaxFenetre - yMinFenetre) ) < erreurToleree\_);  } |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testRedimensionnement : On vérifie que le redimensionnement de la fenêtre ne donne pas lieu à des aberrations qui empêcheraient un utilisateur de jouer correctement. On essaie simplement une valeur d’agrandissement et une valeur de rapetissement de la fenêtre, et on vérifie que la clôture et la fenêtre virtuelle sont mis à jour correctement. | //Un test d'aggrandissement  int setXMinClot = 0, setXMaxClot = 768, setYMinClot = 0, setYMaxClot = 823;  int xMinClot, xMaxClot, yMinClot, yMaxClot;  double xMinFenetre, xMaxFenetre, yMinFenetre, yMaxFenetre;  projection\_->redimensionnerFenetre(Vecteur2i (setXMinClot,setYMinClot ), Vecteur2i (setXMaxClot, setYMaxClot));  projection\_->obtenirCoordonneesCloture( xMinClot,xMaxClot ,yMinClot, yMaxClot);  projection\_->obtenirCoordonneesFenetreVirtuelle(xMinFenetre, xMaxFenetre, yMinFenetre, yMaxFenetre);  //On s'assure que la clôture soit modifiée  CPPUNIT\_ASSERT(xMinClot == setXMinClot && xMaxClot == setXMaxClot && yMinClot == setYMinClot && yMaxClot == setYMaxClot);  //On s'assure que le facteur de zoom ne soit pas modifié  CPPUNIT\_ASSERT( (xMaxClot - xMinClot) / (xMaxClotureInitiale\_ - xMinClotureInitiale\_) - (xMaxFenetreInitiale\_ - xMinFenetreInitiale\_) / (xMaxFenetre - xMinFenetre) );  CPPUNIT\_ASSERT( (yMaxClot - yMinClot) / (yMaxClotureInitiale\_ - yMinClotureInitiale\_) - (yMaxFenetreInitiale\_ - yMinFenetreInitiale\_) / (yMaxFenetre - yMinFenetre) );  //Test de rapetissement  setXMinClot = 0, setXMaxClot = 327, setYMinClot = 0, setYMaxClot = 569;  projection\_->redimensionnerFenetre(Vecteur2i (setXMinClot,setYMinClot ), Vecteur2i (setXMaxClot, setYMaxClot));  projection\_->obtenirCoordonneesCloture( xMinClot,xMaxClot ,yMinClot, yMaxClot);  projection\_->obtenirCoordonneesFenetreVirtuelle(xMinFenetre, xMaxFenetre, yMinFenetre, yMaxFenetre);  //On s'assure que la clôture soit modifiée  CPPUNIT\_ASSERT(xMinClot == setXMinClot && xMaxClot == setXMaxClot && yMinClot == setYMinClot && yMaxClot == setYMaxClot);  //On s'assure que le facteur de zoom ne soit pas modifié  CPPUNIT\_ASSERT( (xMaxClot - xMinClot) / (xMaxClotureInitiale\_ - xMinClotureInitiale\_) - (xMaxFenetreInitiale\_ - xMinFenetreInitiale\_) / (xMaxFenetre - xMinFenetre) );  CPPUNIT\_ASSERT( (yMaxClot - yMinClot) / (yMaxClotureInitiale\_ - yMinClotureInitiale\_) - (yMaxFenetreInitiale\_ - yMinFenetreInitiale\_) / (yMaxFenetre - yMinFenetre) ); |