

**INF2990**

**Projet de logiciel graphique interactif**

**Tests logiciels**

**Version 14.1**

Auteur original : Julien Gascon-Samson

Éditeur : Olivier Gendreau

# Tâche à réaliser

Le cours INF2990 est un projet se voulant « intégrateur ». Dans cette optique, il intègre plusieurs cours que vous avez suivis précédemment et en parallèle, notamment le cours LOG1000 – Ingénierie logicielle.

Dans le cadre du cours LOG1000, vous avez notamment vu le fonctionnement des tests unitaires (en cours théorique et en laboratoire). Plus précisément, en laboratoire, vous avez travaillé avec l’outil *cppunit* pour créer vos tests unitaires, les maintenir et les exécuter. L’équipe académique a retenu cet outil pour le projet intégrateur INF2990. La création et l’exécution de cas de tests constituent une très bonne pratique du génie logiciel. Ainsi, pour intégrer le concept des tests unitaires à votre projet, nous vous demanderons de créer des suites de tests que vous maintiendrez à jour et exécuterez tout au long de la session. 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 jeux 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 jeux de tests pour tester 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 tests 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 la classe Vecteur ou l’arborescence des nœuds, par exemple.

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 jeux de tests lors de l’évaluation du livrable.

Vous trouverez, aux pages 5 et 6, un tableau informatif indiquant comment remplir les tableaux de tests, ainsi qu’un tableau d’exemples (pages 7 et 8), rempli pour les tests unitaires de la classe *NoeudAbstrait*. Notez que les tests unitaires pour cette classe sont fournis en exemple avec le cadriciel (classe *NoeudAbstraitTest*).

# 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 :** | BoiteEnglobanteOrientee  (BoiteEnglobanteOrienteeTest) |  |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | Cette classe gère les collisions de boite englobante qui encadre les objets. Il est important de voir si les collisions sont bien effectués |  |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | testCalculerCollision: Ce test vérifie que la collision entre deux boites englobantes fonctionne bien. On joue avec les rotations aussi, puisque c’est le gros avantage de cette classe. | // Deux boites l'une dans l'autre  math::BoiteEnglobanteOrientee obb1(Vecteur3f(0.0f, 0.0f, 0.0f), Vecteur3f(10.0f, 10.0f, 0.0f));  math::BoiteEnglobanteOrientee obb2(Vecteur3f(0.0f, 0.0f, 0.0f), Vecteur3f(10.0f, 10.0f, 0.0f));  CPPUNIT\_ASSERT(obb1.calculerCollision(obb2));  // On les sépare  obb2.update(aiVector3D(15.0f, 0.0f, 0.0f), 0.0f, aiVector3D(1.0f, 1.0f, 1.0f));  CPPUNIT\_ASSERT(!obb1.calculerCollision(obb2));  // On en oriente un à -90 degré  obb2.update(aiVector3D(15.0f, 0.0f, 0.0f), -90.0f, aiVector3D(1.0f, 1.0f, 1.0f));  // Il ne devrait pas encore collisioner car ils sont séparés et que la rotation l'a  // plus déplacé vers la droite  CPPUNIT\_ASSERT(!obb1.calculerCollision(obb2));  // On redéplace la boite  obb2.update(aiVector3D(5.0f, 0.0f, 0.0f), 90.0f, aiVector3D(1.0f, 1.0f, 1.0f));  CPPUNIT\_ASSERT(obb1.calculerCollision(obb2)); |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #2*** |  |  |
| **Classe :** | Vaisseau (TestVaisseau) |  |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | La classe Vaisseau est une dérivée de ElementJeuAbstrait, et donc, il est intéressant de tester que tous ses attributs, lorsque modifiés, sont correctement changés. Étant donné que le vaisseau est un élément de base du jeu, ainsi que tous les autres éléments dérivés de ElementJeuAbstrait, il est important de vérifier leur bon fonctionnement. |  |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | testAngle : Suite à son initialisation à 0, nous créons un float de 90.0 et nous modifions l’attribut angle\_ de element\_ (ici notre vaisseau). Puis, nous vérifions que l’angle a bien été modifié. | void TestVaisseau::testAngle()  {  float angle = 90.0;  element\_->setAngle(angle);  CPPUNIT\_ASSERT(element\_->getAngle() == angle);  } |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testPosition : Ici, nous testons en même temps les setter/getter de positionActuelle\_ et positionDepart\_ en vérifiant chacune des composantes des vecteurs position. | void TestVaisseau::testPosition()  {  aiVector3D position = (1.9f,2.0f,2.1f);  element\_->setPositionActuelle(position);  element\_->setPositionDepart(position);  // Test pour la position actuelle  CPPUNIT\_ASSERT(element\_->getPositionActuelle().x == position.x);  CPPUNIT\_ASSERT(element\_->getPositionActuelle().y == position.y);  CPPUNIT\_ASSERT(element\_->getPositionActuelle().z == position.z);  // Test pour la position de dÈpart  CPPUNIT\_ASSERT(element\_->getPositionDepart().x == position.x);  CPPUNIT\_ASSERT(element\_->getPositionDepart().y == position.y);  CPPUNIT\_ASSERT(element\_->getPositionDepart().z == position.z);  } |
| **Cas test #3 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testAxe : Nous vérifions les get/set de l’axe de rotation afin de s’assurer que lorsque l’axe sera appelé pour les rotations, ce sera le bon. | void TestVaisseau::testAxe()  {  Vecteur3f axe = (0.0f, 1.0f, 0.0f);  element\_->setAxe( axe );  //Test sur chaque composante du vecteur  CPPUNIT\_ASSERT(element\_->getAxe().x == axe[0]);  CPPUNIT\_ASSERT(element\_->getAxe().y == axe[1]);  CPPUNIT\_ASSERT(element\_->getAxe().z == axe[2]);  } |
| **Cas test #4 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | testVitesse : Nous vérifions, ici, l’exactitude des set/get de la vitesse de l’élément. | void TestVaisseau::testVitesse()  {  aiVector3D vitesse = ( 1.0f, 2.0f, 3.0f);  element\_->setVitesse( vitesse );  //Test sur chaque composante du vecteur  CPPUNIT\_ASSERT(element\_->getVitesse().x == vitesse.x);  CPPUNIT\_ASSERT(element\_->getVitesse().y == vitesse.y);  CPPUNIT\_ASSERT(element\_->getVitesse().z == vitesse.z);  } |
| **Cas test #5 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | testId : Nous vérifions que lors de la modification de l’identificateur de l’élément, celui-ci donne le résultat attendu. | void TestVaisseau::testId()  {  unsigned int id = 22;  element\_->setId( id );  CPPUNIT\_ASSERT( element\_->getId() == id);  } |
| **Cas test #6 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | testPoids : Cette vérification permet une exactitude lors de la modification de l’attribut poids de l’élément de part ses fonctions set/get. | void TestVaisseau::testPoids()  {  unsigned int poids = 22;  element\_->setPoids( poids);  CPPUNIT\_ASSERT( element\_->getPoids() == poids);  } |
| **Cas test #7 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | testTaille : Tout d’abord, nous créons un vecteur pour la taille que nous désirons et ensuite, avec la fonction setTaille(), nous modifions l’attribut taille de l’élément. Finalement, nous vérifions l’exactitude du résultat. | void TestVaisseau::testTaille()  {  aiVector3D taille = ( 1.0f, 2.0f, 3.0f);  element\_->setTaille( taille );  //Test sur chaque composante du vecteur  CPPUNIT\_ASSERT(element\_->getTaille().x == taille.x);  CPPUNIT\_ASSERT(element\_->getTaille().y == taille.y);  CPPUNIT\_ASSERT(element\_->getTaille().z == taille.z);  } |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #3*** |  |  |
| **Classe :** | GestionnaireXML et ConfigScene (ConfigSceneTest) |  |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | Puisque le chargement de carte ainsi que l’enregistrement de cartes est un élément important du jeu, il est important de considérer les tests pour cette classe. Son bon fonctionnement est important, car tout au long du jeu, l’utilisateur doit être capable de charger et enregistrer ses cartes dans le mode édition. |  |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | testSauvergardeChargement : Ici, nous changeons la difficulté de la carte, nous l’enregistrons, nous la modifions de nouveaux, puis nous chargeons la carte enregistrée précédemment et vérifions l’exactitude de l’enregistrement. | void ConfigSceneTest::testSauvegardeChargement()  {  // On ajoute une station pour pouvoir sauvegarder le fichier XML  Station \*stationTest = new Station(FacadeModele::obtenirInstance()->obtenirArbreRenduINF2990());  FacadeModele::obtenirInstance()->getCarte()->ajouterElement(stationTest);  // On modifie la difficulte  FacadeModele::obtenirInstance()->getCarte()->setDifficulte(2);  // On sauvegarde le XML...  FacadeModele::obtenirInstance()->enregistrerConfiguration(nom\_);  // On remet les valeurs par défaut  FacadeModele::obtenirInstance()->getCarte()->setDifficulte(1);  // On charge le XML...  FacadeModele::obtenirInstance()->chargerConfiguration(nom\_);    // On vérifie si les valeurs de test sont celles qui ont bien été chargées  CPPUNIT\_ASSERT( FacadeModele::obtenirInstance()->getCarte()->getDifficulte() == 2 );  // On réaffecte les valeurs par défaut  FacadeModele::obtenirInstance()->getCarte()->setDifficulte(1);  // On resauvegarde le XML  FacadeModele::obtenirInstance()->enregistrerConfiguration(nom\_);  } |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #4*** |  |  |
| **Classe :** | ProjectionOrtho (ProjectionOrthoTest) |  |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | Étant donnée l’importance de la vue dans le jeu, il va de soit que tester la classe ProjectionOrtho est une priorité afin de s’assurer de l’exactitude de ses méthodes et ainsi, une vue bien fonctionnelle pour le jeu. |  |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | ajusterRapportAspect : Ce test permet de s’assurer que lorsqu’on appelle la fonction ajusterRapportAspect, le rapport d’aspect de la fenêtre est bien respecté. | void ProjectionOrthoTest::ajusterRapportAspect()  {  // Dans l'instanciation d'une projection orthogonale, ajusterRapportAspect()  // sera appelé, donc il ne reste qu'à vérifier les rapports.  Vecteur4 fenetreVirtuelle;  Vecteur2i cloture( projOrtho\_->obtenirDimensionCloture() );  projOrtho\_->obtenirCoordonneesFenetreVirtuelle(fenetreVirtuelle[0], fenetreVirtuelle[1],  fenetreVirtuelle[2], fenetreVirtuelle[3]);    double rapportFenetre( (fenetreVirtuelle[1]-fenetreVirtuelle[0])/(fenetreVirtuelle[3]-fenetreVirtuelle[2]) );  double rapportCloture( (double)cloture[0]/(double)cloture[1] );    // Si le ratio de la fenetre et le ratio de la cloture sont equivalentes,  // alors le ratio entre ces deux devraient etre le meme, donc equivalent a 1.  CPPUNIT\_ASSERT( abs(1.0f-(rapportFenetre/rapportCloture) < utilitaire::EPSILON) );  } |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testRedimensionnementFenetre : Ce test permet de valider l’exactitude de la méthode redimensionnerFenetre en lui fournissant de nouvelles largeurs et hauteurs. | void ProjectionOrthoTest::testRedimensionFenetre()  {  // Puisque Java nous envoie toujours les dimensions de la cloture dans  // le bon rapport d'aspect, on suppose que l'on obtient des parametres  // avec le bon rapport d'aspect, il ne reste qu'a verifier si le rapport  // entre la fenetre et la cloture reste pareil.  for(int largeurCloture = 200; largeurCloture < 1080; ++largeurCloture)  {  int hauteurCloture = largeurCloture/RATIO;  projOrtho\_->redimensionnerFenetre( Vecteur2i(0, 0), Vecteur2i(largeurCloture, hauteurCloture) );  // Meme code que le test d'ajustement de rapport d'aspect.  Vecteur4 fenetreVirtuelle;  Vecteur2 cloture( projOrtho\_->obtenirDimensionCloture() );  projOrtho\_->obtenirCoordonneesFenetreVirtuelle(fenetreVirtuelle[0], fenetreVirtuelle[1],  fenetreVirtuelle[2], fenetreVirtuelle[3]);    double rapportFenetre( (fenetreVirtuelle[1]-fenetreVirtuelle[0])/(fenetreVirtuelle[3]-fenetreVirtuelle[2]) );  double rapportCloture( (double)cloture[0]/(double)cloture[1] );  // On verifie si la difference est inferieure a 1%, qui est une limite acceptable.  CPPUNIT\_ASSERT( (abs(1.0f-rapportFenetre/rapportCloture) < (1.0f/100.0f)) );  }  } |
| **Cas test #3 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testZoomerInOut : Ce test permet de vérifier qu’après un certain nombre de zoomIn et un même nombre de zoomOut, la fenêtre virtuelle se retrouve à la même position qu’elle se trouvait initialement. | void ProjectionOrthoTest::testZoomerInOut()  {  Vecteur4 fenetreAvantZoom;  projOrtho\_->obtenirCoordonneesFenetreVirtuelle(fenetreAvantZoom[0], fenetreAvantZoom[1],  fenetreAvantZoom[2], fenetreAvantZoom[3]);  Vecteur2 pointCentralAvantZoom( (fenetreAvantZoom[0]+fenetreAvantZoom[1])/2.0f,  (fenetreAvantZoom[2]+fenetreAvantZoom[3])/2.0f );  for(int i=0; i<10; ++i){  projOrtho\_->zoomerOut();  }  for(int i=0; i<10; ++i){  projOrtho\_->zoomerIn();  }  Vecteur4 fenetreApresZoom;  projOrtho\_->obtenirCoordonneesFenetreVirtuelle(fenetreApresZoom[0], fenetreApresZoom[1],  fenetreApresZoom[2], fenetreApresZoom[3]);  Vecteur2 pointCentralApresZoom( (fenetreApresZoom[0]+fenetreApresZoom[1])/2.0f,  (fenetreApresZoom[2]+fenetreApresZoom[3])/2.0f );  // Verifier que l'on zoom toujours dans le centre  CPPUNIT\_ASSERT(pointCentralAvantZoom==pointCentralApresZoom);  // 4 car taille du vecteur est de 4 et on veut verifier les 4 bounds.  for(int i = 0; i < 4; ++i){  CPPUNIT\_ASSERT( abs(fenetreAvantZoom[i]-fenetreApresZoom[i]) < utilitaire::EPSILON );  }  } |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #5*** |  |  |
| **Classe :** | Vecteur3f |  |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | La classe représentant un vecteur mathématique en 3 dimensions est utilisé pour les transformations des objets, la sélection et, plus tard, pour les collisions. Il est vitale de s’assurer qu’elle fonctionne bien. |  |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | testProduitScalaire : Ce test vérifie que les multiplications du produit scalaire se font correctement. On assigne des valeurs un peu au hasard pour les vecteurs u et v et on vérifie que l’opération s’est bien effectué | Vecteur3f u(1.0f, 2.0f, 3.0f);  Vecteur3f v(4.0f, 5.0f, 6.0f);  float dot = produitScalaire(u, v);  CPPUNIT\_ASSERT(dot == 32.0f); |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testProduitVectoriel: Ce test vérifie que le produit vectoriel est bien implémenté. On vérifie par ailleurs qu’un produit vectoriel avec deux mêmes vecteurs ne donne pas le même résultat lorsqu’effectué dans l’autre sens. | Vecteur3f u(1.0f, 0.0f, 0.0f);  Vecteur3f v(0.0f, 1.0f, 0.0f);  Vecteur3f w = produitVectoriel(u, v);  CPPUNIT\_ASSERT(w == Vecteur3f(0.0f, 0.0f, 1.0f));    w = produitVectoriel(v, u);  CPPUNIT\_ASSERT(w == Vecteur3f(0.0f, 0.0f, -1.0f));  u[0] = -1.0f;  u[1] = 7.0f;  u[2] = 4.0f;  v[0] = -5.0f;  v[1] = 8.0f;  v[2] = 4.0f;  w = produitVectoriel(u, v);  CPPUNIT\_ASSERT(w == Vecteur3f(-4.0f, -16.0f, 27.0f)); |
| **Cas test #3 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testReflexion : Ce test vérifie que la réflexion d’un vecteur par rapport à une normale est correctement implémenté. | Vecteur3f u(1.0f, -1.0f, 0.0f);  Vecteur3f n(0.0f, 1.0f, 0.0f);  Vecteur3f r = calculerReflexion(u, n);  CPPUNIT\_ASSERT(r == Vecteur3f(1.0f, 1.0f, 0.0f)); |

|  |  |  |
| --- | --- | --- |
| ***Jeu de tests #6*** |  |  |
| **Classe :** | NoeudComposite (NoeudCompositeTest) |  |
| **Justification :**  *(pertinence de tester cette classe)*  *5 lignes max.* | La classe représentant un noeud composite qui permet d’avoir plusieurs noeuds interreliés. Il est vitale de s’assurer que cette classe fonctionne bien. |  |
| **Cas test #1 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *10 lignes max* | testAjouter : Ce test vérifie que l’ajout d’un noeud abstrait au conteneur d’enfants se comporte de la façon attendue, puis on fait la recherche du noeud et compare son adresse avec le noeud inséré. | void NoeudCompositeTest::testAjouter(){  NoeudAbstrait\* noeudAbstrait = new NoeudAbstrait("Noeud Abstrait");  noeud\_->ajouter(noeudAbstrait);  CPPUNIT\_ASSERT( noeud\_->getEnfants()[0] == noeudAbstrait );  } |
| **Cas test #2 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testChercher: Ce test vérifie que, une fois un noeud ajouté, il est possible de faire une recherche dans le conteneur avec son nom pour pouvoir vérifier son existence. | void NoeudCompositeTest::testChercher(){  NoeudAbstrait\* noeudAbstrait = new NoeudAbstrait("Noeud Abstrait");  noeud\_->ajouter(noeudAbstrait);  CPPUNIT\_ASSERT( noeud\_->chercher("Noeud Abstrait") == noeudAbstrait );  } |
| **Cas test #3 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max* | testCalculerProfondeur : Ce test vérifie que la profondeur de l’arbre créé est égale à 1 suite à l’ajout d’un seul noeud. | void NoeudCompositeTest::testCalculerProfondeur(){  NoeudAbstrait\* noeudAbstrait = new NoeudAbstrait("Noeud Abstrait");  noeud\_->ajouter(noeudAbstrait);  CPPUNIT\_ASSERT( noeud\_->calculerProfondeur() == 1 );  } |
| **Cas test #4 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | testEffacer: Ce test vérifie que à la suite d’une suppression dans le conteneur de noeud, le noeud ne s’y retrouve plus. | void NoeudCompositeTest::testEffacer(){  NoeudAbstrait\* noeudAbstrait = new NoeudAbstrait("Noeud Abstrait");  noeud\_->ajouter(noeudAbstrait);  noeud\_->effacer(noeudAbstrait);  CPPUNIT\_ASSERT( noeud\_->chercher("Noeud Abstrait") == 0);  } |
| **Cas test #5 :**  *(description, choix valeurs, autres infos et*  *code source du cas de test)*  *(si nécessaire)*  *10 lignes max.* | testObtenirNombreEnfant: Ce test vérifie que suite à un ajout d’un enfant, la méthode obtenirNombreEnfants() retourne bien le nombre d’enfants réel. | void NoeudCompositeTest::testObtenirNombreEnfant(){  NoeudAbstrait\* noeudAbstrait = new NoeudAbstrait("Noeud Abstrait");  noeud\_->ajouter(noeudAbstrait);  CPPUNIT\_ASSERT( noeud\_->obtenirNombreEnfants() == 1 );  } |