École Polytechnique de Montréal
Département de Génie Informatique et Génie Logiciel

INF2990
Projet de logiciel graphique interactif
Tests logiciels

Remis par :

Matricule	Prénom & Nom
1733229	Olivier St-Amour
1736842	Philippe Marcotte
1739351	Simon-Pierre Desjardins
1740216	Ulric Villeneuve
1744468	Frédéric Grégoire
1745728	Camille Gendreau

Le 21 mars 2016

Introduction

Lors de la réalisation de ce projet, il nous a été possible d'écrire de nombreuses lignes de codes sans nécessairement les tester de manière formelle. Malgré le fait que les tests unitaires ne permettent pas de tester toutes les facettes d'une méthode, ils peuvent tout de même permettre d'éviter quelques problèmes et surtout de bien comprendre le fonctionnement de notre code dans certaines conditions définies. Dans ce projet, à la suite de notre apprentissage de ces connaissances dans le cours d'Ingénierie Logicielle, nous allons effectuer des tests unitaires sur une vingtaine de méthodes et ce dans un minimum de 6 classes différentes afin de s'assurer de couvrir un minimum de fonctionnalité. Lorsque l'implémentation de nos tests sera complétée, nous justifierons chacun des tests ainsi que leur pertinence afin de bien expliquer nos différents choix. Bref, il est important de bien intégrer les tests dans nos pratiques dès maintenant, car ils feront partie nécessairement de notre futur en tant qu'ingénieur en informatique/logiciel dans le marché du travail à un moment ou un autre.

Présentation des travaux

Suite de cas de test #1			
Classe testée	RectangleEnglobant (PhysiqueTest)	**Branche**	master
Justification			
Nous testons ici un rectangle englobant. Il est intéressant de tester cette classe puisque nous pouvons vérifier le bon fonctionnement des méthodes qui composent la physique de notre programme. Nous testons notamment certaines conditions booléennes et nous assurons qu'aucune forme englobante ne soit en contact l'une avec l'autre. S'ils sont en contact, ceux-ci retourne une normale de collision. Puisque rectangle englobant est toujours en contact avec des cercle englobant ou des points, les méthodes sont fortement en dépendance avec ces deux formes.			

Cas de test #1	
Méthode testée	`Bool RectangleEnglobant::calculerEstDansForme( const glm::dvec3& point) const`
Justification	
Comme un rectangle englobant est la majorité des formes englobantes (mur, ligne, robot, table) il faut être en mesure de savoir si un point est dans le rectangle englobant. Dès qu'un point est dans le rectangle on peut donc réagir avec une autre action par la suite.	
Explication du cas de test	
Ce test a été effectué avec un seul rectangle englobant et un point changeant de place à multiple reprises. Le point est détecté au pixel près dans la forme. Les cas de test sont les quatres coins du rectangle, les quatres coins milieux de chaque segment du rectangle, des points aléatoires à l'intérieur du rectangle et des points aléatoires à l'extérieur du rectangle.	

Cas de test #2	
Méthode testée	`Bool RectangleEnglobant::calculerIntersection(const RectangleEnglobant& rectangle) const`
Justification	
Comme un rectangle englobant est la majorité des formes englobantes (mur, ligne, robot, table) il faut être en mesure de savoir si deux rectangles sont en intersection. Dès qu'un rectangle est dans le rectangle on peut donc réagir avec une autre action par la suite.	
Explication du cas de test	
Ce test a été effectué avec deux rectangles englobants en changeant de place à multiple reprises. L'intersection est détectée à tous les contacts au pixel près. Les cas de test sont les quatres coins d'un rectangle avec les quatre coins de l'autre, lorsqu'un segment complet est par dessus un segment du second rectangle, quelques tests où le second rectangle est en angle et par dessus l'autre rectangle et des tests où les rectangles ne se touchent pas.	

Cas de test #3	
Méthode testée	`Bool RectangleEnglobant::calculerIntersection(const CercleEnglobant& cercle) const`
Justification	
Comme un rectangle englobant est la majorité des formes englobantes (mur, ligne, robot, table) il faut être en mesure de savoir si un rectangle est en intersection avec un cercle (poteau). Dès qu'un rectangle est dans le cercle on peut donc réagir avec une autre action par la suite.	
Explication du cas de test	
Ce test a été effectué avec un rectangle englobant et un cercle en le changeant de place à multiple reprises. L'intersection est détectée à tous les contacts au pixel près. Les cas de test sont les quatres coins d'un rectangle avec la limite du cercle, le point milieu des segments du rectangle avec l'extrémité du cercle et lorsque le cercle n'est pas en contact avec le cercle.	

Cas de test #4	
Méthode testée	`Bool RectangleEnglobant::assignerAngle(const double& angle)`
Justification	
Permet d'assigner un angle à un rectangle englobant. Cette méthode doit uniquement affecter le rectangle englobant et non l'objet qui possède le rectangle englobant. De plus, l'angle doit uniquement affecté un rectangle englobant.	
Explication du cas de test	
Assigner un angle de rotation quelconque à un rectangle englobant et vérifier la variable représantant cet attribut.	

Cas de test #5	
Méthode testée	`Glm::dvec3 RectangleEnglobant::calculerNormaleCollision(const RectangleEnglobant& rectangle) const`
Justification	
La normale d'un objet permet de déterminer la nouvelle position d'un objet après la collision. Comme le robot est représenté par un rectangle englobant et qu'il peut entrer en collision avec un mur ou la table, il faut être en mesure d'avoir la bonne normale pour ces objets.	
Explication du cas de test	
Ce test a été effectué avec deux rectangles englobants en changeant un rectangle de place à multiple reprises. Les cas de test utilisent la méthode pour savoir s'il y a une intersection entre deux rectangles, puis retourne la normale calculée. Pour ce test, nous vérifions la normale calculée par le programme avec la normale que nous expectons reçevoir sans aller dans le code.	

Cas de test #6	
Méthode testée	`Glm::dvec3 RectangleEnglobant::calculerNormaleCollision(const CercleEnglobant& rectangle) const`
Justification	
La normale d'un objet permet de déterminer la nouvelle position d'un objet après la collision. Comme le robot est représenté par un rectangle englobant et qu'il peut entrer en collision avec un poteau, il faut être en mesure d'avoir la bonne normale pour ces objets. La normale d'une collision avec un cercle (poteau) est toujours le vecteur entre les deux positions centres.	
Explication du cas de test	
Ce test a été effectué avec un rectangle et un cercle englobants en changeant le cercle de place à multiple reprises. Les cas de test utilisent la méthode pour savoir s'il y a une intersection entre un rectangle et un cercle, puis retourne la normale calculée. Pour ce test, nous nous vérifions la normale calculée par le programme avec la normale que nous expectons recevoir sans aller dans le code.	

Cas de test #7	
Méthode testée	`Glm::dvec3 RectangleEnglobant::calculerNormaleCollision(const glm::dvec3& point) const`
Justification	
La normale d'un objet permet de déterminer la nouvelle position d'un objet après la collision. Comme le robot est représenté par un rectangle englobant et qu'il peut entrer en collision avec la table, qui est plusieurs points, il faut être en mesure d'avoir la bonne normale pour ces objets. La normale d'une collision avec un cercle (poteau) est toujours le vecteur entre les deux positions centres.	
Explication du cas de test	
Ce test a été effectué avec un rectangle englobant et un point en changeant le point de place à multiple reprises. Les cas de test utilise la méthode pour savoir s'il y a une intersection entre un rectangle et un point, puis retourne la normale calculée. Pour ce test, nous nous vérifions la normale calculée par le programme avec la normale que nous expectons recevoir sans aller dans le code.	

Suite de cas de test #2			
Classe testée	CercleEnglobant (PhysiqueTest)	**Branche**	master
Justification			
Nous testons ici un cercle englobant. Il est intéressant de tester cette classe puisque nous pouvons vérifier le bon fonctionnement des méthodes qui composent la physique de notre programme. Nous testons notamment certaines conditions booléennes et nous assurons qu'aucune forme englobante ne soit en contact l'une avec l'autre ou ceux-ci retourne une normale de collision. Puisque le cercle englobant est toujours en contact avec des rectangles englobant ou des points, les méthodes sont fortement en dépendance avec ces deux formes.			

Cas de test #8	
Méthode testée	`Bool CercleEnglobant::calculerEstDansForme( const glm ::dvec3& point) const`
Justification	
Comme un cercle englobant est utilisé pour un poteau. il faut être en mesure de savoir si un point est dans un cercle englobant. Dès qu'un point est dans le cercle, on peut réagir avec une autre action par la suite.	
Explication du cas de test	
Ce test a été effectué avec un seul cercle englobant et un point changeant de place à multiple reprises. Le point est détecté au pixel près dans la forme. Les cas de test sont les extrémités du cercle, des points aléatoires dans le cercle et des points aléatoires à l'extérieur du cercle.	

Cas de test #9	
Méthode testée	`Bool CercleEnglobant::calculerIntersection(const CercleEnglobannt& cercle) const`
Justification	
Cette méthode est utile à tester puisque le capteur optique et le suiveur de ligne l'utilisent pour détecter une jonction dans une ligne. De cette façon, ils peuvent continuer le même comportement sur une ligne formée de plusieurs segments.	
Explication du cas de test	
Ce test a été effectué avec deux cercles englobants en changeant un cercle de place à multiple reprises. L'intersection est détectée à tous les contacts au pixel près. Les cas de test sont les extrémités d'un cercle avec l'autre cercle dans toutes les positions possibles et quand les deux cercles ne se touchent pas.	

Suite de cas de test #3			
Classe testée	SuiveurLigne (CapteurTest)	**Branche**	master
Justification			
Nous testons ici un capteur du robot, soit le suiveur de ligne. Il est intéressant de tester cette classe puisque c'est le capteur qui a le plus de chance de renvoyer une mauvaise donnée et qui est activé la majorité du temps. La classe SuiveurLigne est en fait une classe qui englobe 3 classes CapteurOptique. Ces capteurs optiques ont un cercle englobant servant à vérifier l'intersection avec une ligne et donc faire la vérification d'une ligne. Les capteurs optiques ne sont pas nécessairement très pertinents à tester étant donné que le cercle englobant est déjà testé et que la majorité de la logique et des algorithmes de détection sont effectués dans cette classe. Cependant, la classe SuiveurLigne nous semble pertinente à tester étant donné que celle-ci comporte une partie essentielle du projet. Nous pouvons vérifier le bon fonctionnement des méthodes en créant des segments de ligne et en regardant la valeur des capteurs. Nous testons notamment la valeur retournée par chaque capteur pour suivre la ligne.			

Cas de test #10	
Méthode testée	`Void SuiveurLigne::verifierDetection(NoeudLigne* ligne)`
Justification	
On teste la méthode permettant d'effectuer la vérification de la détection d'une ligne du suiveur de lignes. Celle-ci permet de gérer l'état de l'option active du profil utilisateur ainsi que de distribuer la ligne aux capteurs optiques afin de vérifier la détection. Cette méthode fait également appel à la méthode verifierDetection des capteurs optiques servant à faire la vérification des différentes composantes de la ligne. Cette méthode sert donc à tester le rôle de médiateur du suiveur de ligne, c'est à dire l'ensemble de ce qui constitue une détection de ligne.	
Explication du cas de test	
Dans le cas de test, on se sert de cette méthode afin d'initialiser la détection d'une ligne manuellement créée. On utilise la méthode obtenirEtatCapteurs() pour vérifier la validité de cette méthode.	

Cas de test #11	
Méthode testée	`uint8_t SuiveurLigne::obtenirEtatCapteurs() const`
Justification	
Permet d'obtenir l'état des trois capteurs qui sont utilisés pour faire le suivi d'une ligne. Cette méthode est essentielle à tester puisqu'elle assure les besoins de base pour que le comportement suivre ligne puisse fonctionner correctement. Cette méthode se charge de traduire l'état des 3 capteurs sous la forme d'un entier non signé où les 3 premiers bits indiquent la détection d'une ligne pour chaque capteur (de droite à gauche).	
Explication du cas de test	
Placer une ligne sous un capteur et vérifier la valeur de retour du capteur. Dans ce test, la ligne est déplacée pour activer plusieurs capteurs et on vérifie la validité de la détection avec le résultat sous la forme d'un entier positif sur 8 bits. Cette méthode est également utilisée pour vérifier la validité des autres méthodes testées pour cette classe.	

Cas de test #12	
Méthode testée	`Void SuiveurLigne::mettreAJourCapteurs() const`
Justification	
Cette méthode est utile à tester étant donné que le suiveur de ligne se charge également de lancer la mise à jour de la position des capteurs optiques ainsi que que de leur cercle englobant qui sont dépendants de la position et de l'angle actuel du robot étant donné que le cercleEnglobant a besoin d'une position courante absolue et non relative pour effectuer son algorithme de détection d'intersection. Tout comme la méthode veirifierDetection(), ce test sert à tester le rôle de médiateur du capteurOptique.	
Explication du cas de test	
La mise à jour des capteurs doit être effectué avant de vérifier la détection d'une ligne afin d'initialiser correctement la position des capteurs optiques. La validité de cette méthode est vérifiée grâce à un appel subséquent à la méthode obtenirEtatCapteurs().	

Suite de cas de test #4			
Classe testée	ControleRobot(ControleRobotTest)	**Branche**	master
Justification			
Cette suite test différentes fonctions du contrôleur du robot qui sont surtout en lien avec l'intelligence artificielle du robot. Il s'agit ici de tester le mode automatique et les comportements du robot et surtout de s'assurer que l'assignation et les changements de ceux-ci ne laissent pas le contrôleur dans un état incohérent au SRS ou à la simulation.			

Cas de test #13	
Méthode testée	`void ControleRobot::assignerVecteurComportements (std::vector<std::unique_ptr<ComportementAbstrait>>* vecteur)`
Justification	
Nous devons nous assurer que le vecteur de comportements est proprement assigné au controleur du robot car il est appelé à chaque changement de profil et assure aussi en partie le bon fonctionnement de la fonction d'assignation de comportement qui est elle appelée beaucoup plus souvent par les comportements en cours d'execution.	
Explication du cas de test	
On s'assure que le pointeur d'un vecteur créé dans le cadre du test arrive bien à destination dans le ControleRobot en vérifiant la concordance de la variable privée.	

Cas de test #14	
Méthode testée	`void ControleRobot::assignerComportement (TypeComportement nouveauComportement, std::wstring declencheur)`
Justification	
L'assignation de comportements dans le controleur du robot assure que le bon comportement sera adopté. De plus, c'est une fonction avec plusieurs composantes délicates dont il faut prendre compte, alors il est bien de pouvoir procéduralement assurer son fonctionnement avant d'assumer que c'est le comportement qui est une source d'erreur. La fonction s'occupe aussi de l'initialisation des comportements en appelant leur fonction virtuelle respective à ce sujet.	
Explication du cas de test	
On crée un vecteur avec chaque comportement tel que le robot possèderait au moment de l'exécution. Par la suite, on appelle un changement de comportement pour passer par tous les comportements possibles.	

Cas de test #15	
Méthode testée	`Void ControleRobot::passerAModeAutomatique()`
Justification	
Le passage au mode automatique est appelé à plusieurs reprises dans le code autre que lors de l'entrée utilisateur. En effet, après sa construction, le robot est placé en mode automatique. De plus, il y a passage à ce mode à chaque fois que l'utilisateur change de profil. Elle a pour but de démarrer un fil d'éxécution séparé ce qui peut mener à du code délicat dont il est bénifique de vérifier le bon fonctionnement par le biais de tests.	
Explication du cas de test	
On commence le test en plaçant le robot en mode manuel et en lui assignant un comportement qui n'est pas par defaut. Par la suite, on le fait passer au mode automatique. Nous vérifions qu'après ce passage, le fil d'éxécution servant à la logique du robot soit créé et présente. De plus, on s'assure que le booléen indiquant si le robot est en mode manuel est assigné à faux. Finalement, on vérifie si le comportement assigné après le passage est bel et bien le comportement par défaut.	

Cas de test #16	
Méthode testée	`void ControleRobot::passerAModeManuel()`
Justification	
Le passage au mode manuel semble simple en premier lieu, mais il appel aussi la terminaison du fil d'exécution de l'IA. Il faut donc s'assurer que le thread du robot est bel et bien terminé convenablement et que l'état du robot soit fonctionnel pour le passage au mode automatique.	
Explication du cas de test	
On commence le test en plaçant le robot en mode automatique pour s'assurer que le changement ait lieu. Ensuite, on procède au passage au mode manuel en vérifiant par après si le thread est désalloué et si le booléen indiquant le mode manuel est vrai.	

Cas de test #17	
Méthode testée	`void ControleRobot::inverserModeControle()`
Justification	
Cette fonction est appelée par la commande de l'utilisateur. Elle manipule les fonctions testées ci-haut, mais il s'agit de bien tester si l'alternance entre les deux fonctionne et s'il n'y a pas d'interactions malsaines	
Explication du cas de test	
On commence le test en plaçant le robot en mode manuel. On passe ensuite au mode automatique grace à l'inversion en testant les mêmes cas que le test précédent et on refait l'inversion pour voir si le passage au mode manuel s'est fait selon les attentes.	

Suite de cas de test #5			
Classe testée	ProfilUtilisateur (ProfilUtilisateurTest)	**Branche**	master
Justification			
Il est important de tester la classe ProfilUtilisateur, car celle-ci est centrale au fonctionnement de la simulation. De plus, celle-ci profite d'une grande indépendance des autres classes, ce qui rend facile de créer un simple profil test pour s'assurer que la sauvegarde, le chargement et la modification des profils s'effectuent correctement.			

Cas de test #18	
Méthode testée	bool ProfilUtilisateur::chargerProfil()
Justification	
Il important de s'assurer que le chargement de profil s'effectue correctement, puisqu'une erreur dans cette fonction aurait des répercussions graves sur le bon fonctionnement du programme. En plus de tester le chargement de profil, ce test vérifie que l'on sauvegarde correctement les attributs d'un profil, puisque si la sauvegarde ne s'effectue pas correctement, le chargement ne pourra pas non plus.	
Explication du cas de test	
On commence par créer un profil dont les attributs sont directement instantiés par le programme à la place de les charger d'un fichier. Ainsi, on s'assure que ce profil est dans un état correct. Par la suite, à partir de ce profil, on crée un fichier contenant toutes les informations pertinentes en format JSON. Ensuite, on charge ce fichier dans un deuxième profil qui est le profil test. Enfin, on compare chaque attributs des deux profils pour voir si ceux-ci sont identiques. Dans le cas où deux attributs diffèrent, cela indique qu'il y a eu un problème lors du chargement du profil.	

Cas de test #19	
Méthode testée	Void ProfilUtilisateur::modifierToucheCommande(uint8_t touche, TypeCommande& commande)
Justification	
Cette fonction sert à modifier les touches associées au commande que l'on envoie pour contrôler le robot. Ainsi, une simple erreur dans cette fonction pourrait causer de flagrants problèmes, puisque l'utilisateur ne pourrait pas contrôler le robot correctement lors d'une simulation. De plus, la modification d'une touche implique la suppression et l'insertion d'un élément dans une map en plus de la modification d'un vecteur. Cela nous semble à risque de causer une erreur, donc nous voulons nous assurer qu'elle fonctionne correctement.	
Explication du cas de test	
Pour tester la fonction, on assigne la touche Y à la commande AVANCER. Ensuite, on vérifie directement, dans le vecteur de touches, si la touche Y s'y trouve. Puis, on vérifie si la touche Y correspond bel et bien à la clé qui renvoie la commande AVANCER dans	

la map de commande.

Suite de cas de test #6			
Classe testée	CommandeRobot (ControleRobotTest)	**Branche**	master
Justification			
Cette classe sert à contrôler le robot. Tester cette classe se révèle pertinent, car une commande n'étant pas construite correctement peut facilement causer plusieurs problèmes. Une commande peut changer la vitesse des moteurs du robot et donc sa direction, ainsi que passer du mode automatique à manuel et vice versa. Il faut donc être certain que la construction d'une commande ne peut pas permettre d'envoyer de mauvaises vitesses ou de changer le mode alors qu'il ne le faut pas.			

Méthode testée	CommandeRobot ::CommandeRobot(TypeCommande commande)
Justification	
Une commande robot est un objet assez simple dans son utilisation. Il suffit de la construire en précisant le type de commande que l'on veut. Ainsi, nous n'avons qu'à tester le constructeur par paramètre pour vérifier que la construction d'une commande s'exécute correctement.	
Explication du cas de test	
Les tests sont limités aux cinq principales commandes: AVANCER, RECULER, ARRETER, ROTATION_DROITE, ROTATION_GAUCHE. AVANCER : Une fois la commande construite, on vérifie que les vitesses des deux moteurs sont belles et bien positives et égales. RECULER : On vérifie que les vitesses des deux moteurs sont négatives et égales. ARRETER : On vérifie que les vitesses sont nulles. ROTATION_DROITE : On vérifie que les vitesses absolues des deux moteurs sont égales. Puis, on s'assure que la vitesse du moteur droite est plus petite ou égale à 0 et que la vitesse du moteur gauche est plus grande que 0. ROTATION_GAUCHE : On vérifie que les vitesses absolues des deux moteurs sont égales. Puis, on s'assure que la vitesse du moteur gauche est plus petite ou égale à 0 et que la vitesse du moteur de droite est plus grande que 0.	

Grille de correction

L'évaluation des tests logiciels vaut 5% du cours. Le correcteur appliquera la grille de correction détaillée ci-dessous. Cette grille tient compte des deux éléments suivants :

- la **pertinence** et la **justification** des cas de test choisis parmi les différentes suites de cas de test, ainsi que les tableaux remplis;
- la **qualité** de l'implémentation des cas de test.

Choix des cas de test

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

Implémentation des cas de test

2,5 pts	La qualité de l'implémentation des cas de test est **excellente**.
2 pts	La qualité de l'implémentation des cas de test est **très bonne**.

1,5 pt	La qualité de l'implémentation des cas de test est **bonne**.
1 pt	La qualité de l'implémentation des cas de test est **médiocre**.
0 pt	La qualité de l'implémentation des cas de test est **nettement insuffisante**.

Conclusion

En conclusion, comme l'implémentation de tests unitaires est un outil critique dans le maintien du bon fonctionnement de tout logiciel, nous en avons réaliser une vingtaine au travers de notre code. En effet, on en retrouve parmi chaque partie importante du programme. Il y a des tests qui se rattachent à la physique des collisions ainsi que les boîtes englobantes des objets, d'autres qui assurent le fonctionnement des capteurs, certains traitent le contrôle du robot, quelques uns vérifient que les changement effectués sur les profils sont satisfaits, etc. Bref, il est important d'avoir de tels tests notamment pour éviter que des changements dans le code ne modifient pas le comportement du logiciel. Il s'agit d'une bonne habitude de penser aux cas de tests potentiels en pleine implémentation d'une méthode.