LOG1000 - TP5

Samuel Rondeau & Pacôme Bondet de la Bernardie

E1.2)

L'avantage d'utiliser l'héritage pour la maintenabilité du code est que si il y a une méthodes (ou des méthodes) des objects dérivés en commun, on n'a qu'à changer les implémentations dans la classe mère plutôt que de les changer dans plusieurs classes filles. C'est logique et pratique.

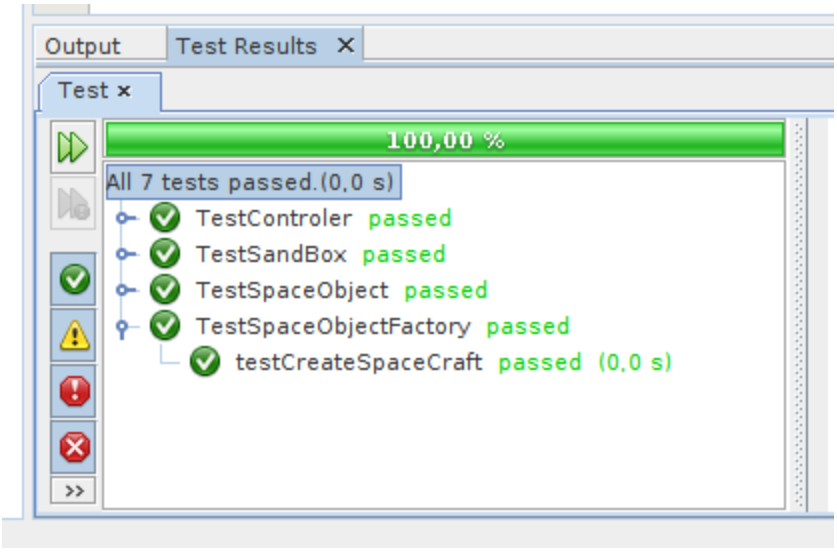
E1.3)

Oui. Dans notre restructuration du programme, la superclasse SpaceObject est une classe asbstraite de laquelle SpaceCraft et SpacePlanet héritent. Ainsi, on voudra forcer l'implémentation de la méthode getTemperature() dans chaque classe fille, et laisser tomber le switch/case dans la classe mère (et la déclarer virtuelle pure).
Également, on déclare que la fonction getTemperature() ne prend aucun paramètres, puisqu'elle peut y accéder directement : ce sont ceux de l'objet appellant.

E1.4)

Oui. On voudra les attributs *protected*, c'est à dire accessibles aux classes dérivées, mais privées aux classes externes. Cela peut nécessiter l'implémentation de getters/setters. On respecte le principe d'encapsulation.

E1.6)



E2.1.a)

SpaceObject::getTemperature() pue le problème de cohésion logique, à cause de son switch/case qui n'Est pas la meilleure façon de gérer le polymorphisme.

E2.1.b)

On la rend polymorphique et on retire son switch/case pour rendre la fonction propre à une classe fille. Pour ce faire, on la déclare virtuelle pure dans SpaceObject pour forcer son implémentation dans chaque classe fille. Puisque la fonction est définie pour chaque classe fille, on retire le switch/case et le fonction est alors propre à une classe fille.

E2.1.c)

Voir code.

E2.2.a)

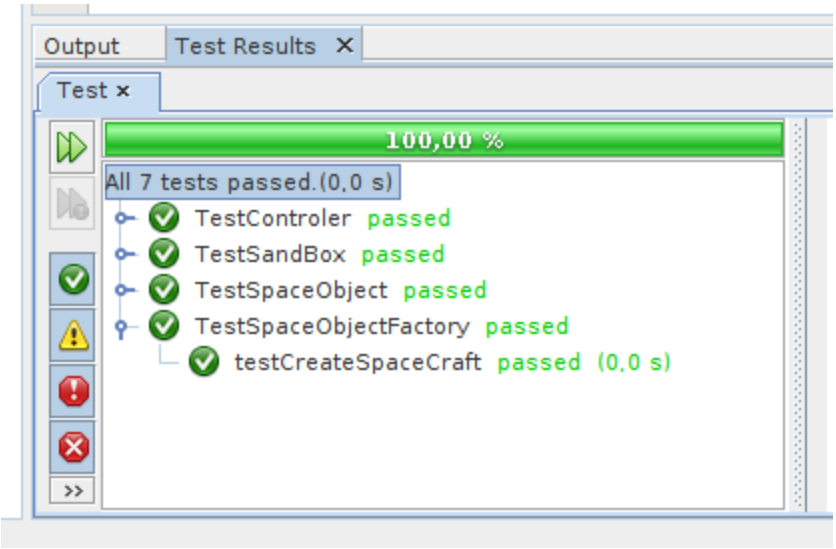
SandBox::wantSpaceCraftIds() et SandBox::wantSpacePlanetIds(), pour la même raison qu'en E2.1.a). On sait que SpaceCraft et SpacePlanet héritent de SpaceObject, et SpaceObject possède le string id. Du coup, SandBox::addSpaceCraft() et SandBox::addSpacePlanet() devront aussi être modifiées, et les attributs vecteurs aussi. Cependant, on ne sait pas exactement ce que l'agence spatiale canadienne veut. Doit-on garder les vecteurs séparés, pour clairement avoir un vecteur de SpaceCraft et un vecteur de SpacePlanet, puisque ce sont des objets différents avec des propriétés différentes, comme par exemple leur mouvement dans la sandbox (un vaisseau et une planète ont certainement des propriétés intrinsèques différentes). Ou doit-on les remplacer par un vecteur de SpaceObject pour répondre à la restructuration suite à la création de la superclasse? Il faudra demander à l'agence leur préférence: une bonne séparation du code, ou une généralisation du code. Il y a donc ici une ambiguïté. On choisit le deuxième cas, puisque le premier cas demande une grande restructuration, incluant des restructurations de code qu'on ne demande pas de modifier explicitement, tel que spécifié à la condition 1).

E2.2.b)

Dans le premier cas, on va remplacer chaque paire de méthodes par une seule méthode générique, qui ira obtenir les id de SpaceObject, sachant qu'on s'intéresse au id des classes filles, et que SpaceObject ne peut lui-même être instancé; qui ajoutera dans le vecteur générique un pointeur d'une classe dérivée de SpaceObject. Suite à l'Ambiguïtée décrite en E2.2.a), nous choisissons le deuxième cas: des méthodes et attributs séparés pour des objets différents.

E2.3)

Puisque les tests en E1.6) ont été adaptés au code restructuré au E2.1), et à la décision de la non restructuration du E2.2) suite à l'ambiguïté décrite à E2.2.a), il n'y a aucun changement par rapport aux tests en E1.6).



E3.1)

Non. Par défaut, elle ne possède aucun attribut. Supposons qu'on demande cette question après l'ajout des attributs. Alors les nouveaux attributs ne sont pas bien gérés. Ils doivent être mis *protected*, et ne possèdent pas de *getters/setters*. On devra donc les créer. Ainsi, si une classe externe désire obtenir un attribut, par exemple si la sandbox désire obtenir id, elle passera par un getter; et si un attribut doit être modifié, ce sera fait par l'entremise d'un setter.

Pourquoi metre les attributs protected? Parce qu'ils sont directement hérités par les classes dérivées, en font intégralement partie, et qu'ils ne font pas de sens dans SpaceObject, qui ne peut être instancé directement.

E3.2.a)

Si on compte les lignes ne comptant qu'une accolade à l'intérieur de la méthode, mais pas celle définissant l'extérieur de la méthode, le scope de i est 42 - 29 + 1 = 14.

E3.2.b)

(2 + 1 + 0 + 1 + 1 + 0 + 0 + 1) / 8 = 6 / 8 = 0.75

E3.2.C)

Non. i est déclarée trop tôt. On pourrait la déclarer deux lignes plus tard. On pourrait également remplacer les deux boucles while par des boucles for, puisque la structure le permet, en l'occurence la valeur maximale de i est connue au début de la boucle et ne change pas. Ainsi, on peut également supprimer la ligne inutile i = 0 entre les deux boucles et réinitialiser i à 0 uniquement dans la déclaration de la seconde boucle for.

Ceci suppose qu'on utilise toujours le code original, qui suit la logique du deuxième cas proposé au E2.2.a).

E3.3.a)

Oui.

std::string id est justifié, car chaque objet, qu'il soit un vaisseau ou une planète, possède un nom, et il est logique qu'un nom soit une chaîne de caractères.

Data3D position est justifié, car il est important de connaître la position d'un objet. Sa position est donnée en 3 dimensions, et Data3D est une structure donnant la position en x, y et z de l'objet.

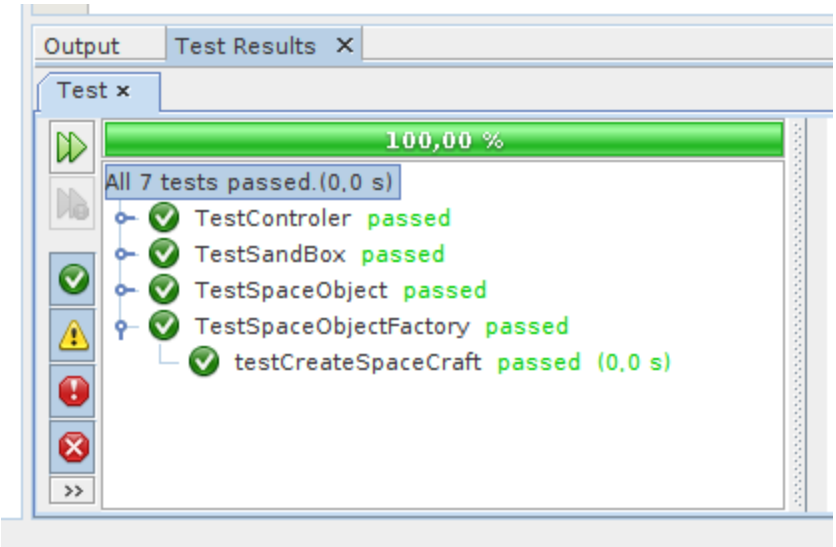
Data3D velocity est justifié, car on veut savoir la vitesse d'un objet dans la sandbox. Sa vitesse doit être donnée elle aussi en trois dimensions. La vitesse n'est pas totalement dépendante de l'accélération, car on peut vouloir lui donner une vitesse initiale et connaître facilement la vitesse sans devoir calculer la la vitesse selon le temps et l'accélération de l'objet.

Data3D acceleration est justifié, car afin de déterminer la vitesse d'un objet à un temps, on a besoin de connaître son accélération, encore une fois selon trois dimensions. On pourrait cependant remplacer l'accélération par la force appliquée sur l'objet, toujours un Data3D, mais cela nécessiterait un nouvel attribut: la masse.

Les quatre attributs présents sont donc justifiés, mais il s'agit du strict minimum. On pourrait ajouter plusieurs autres attributs importants, comme la masse de l'objet, son momentum, ses dimensions, sa date de création, etc.

E3.4)

En supposant qu'on ait choisit le deuxième cas en E2.2.a):


Output
Test Results
Test
100,00 %
All 7 tests passed.(0,0 s)
TestControler passed
TestSandBox passed
TestSpaceObject passed
TestSpaceObjectFactory passed
testCreateSpaceCraft passed (0,0 s)