Gestionnaire d'un inventaire de guitares

Objectifs

Apprendre, comprendre et appliquer l'analyse et la conception orientées objet sur un cas d'étude simple.

Copyrights

Le cas d'étude et les codes sources sont tirés du livre « Tête la première – Analyse et conception orientées objet » (Ed. O'Reilly).

Le cas d'étude

Créer une application de gestion d'inventaire pour le magasin « Les guitares d'Eric ». Cette application doit permettre d'aider Eric à rechercher les guitares de son inventaire qui correspondent aux besoins de ses clients.

Conseils *:*

- pour chaque version de l'application travaillez dans un même projet Eclipse mais dans des paquetages différents (*version1*, *version2,* etc.) ;

- faites des diagrammes de classes pour chaque version à l'aide d'ObjectAID.

Première version

TO DO :
1 – Récupérez le code (lien dépôt GIT sur l'espace-cours – projet avec plusieurs paquetages par version ; veillez à faire une copie et à travailler dans un autre projet que celui venant du dépôt).
2 – Compléter le code de la première version.
3 – Exécutez et trouvez pourquoi aucune guitare ne semble correspondre…
4 – Identifiez des problèmes à cette application. Par quoi commenceriez-vous ?

UN PEU DE COURS…

Un bon logiciel en 3 étapes :
1 - s'assurer que le logiciel fait ce que le client veut qu'il fasse
2 – appliquer les principes OO de base pour ajouter de la souplesse
3 – oeuvrer pour une conception facile à maintenir et à réutiliser

Deuxième version : satisfaire le client

TO DO :
1 – Améliorez les comparaisons fastidieuses de chaines en utilisant des types énumérés (cf. code à trous de la version 2) afin d'améliorer le contrôle des types et valeurs.
2 – Modifiez les classes existantes pour prendre en compte ces types énumérés.
3 – N'oubliez pas de modifiez la comparaison du String restant (modèle).
4 – Testez (ça doit fonctionner mais pour une seule guitare !)
5 – Modifiez la classe d'inventaire pour que la recherche renvoie la liste de toutes les guitares correspondantes aux préférences du client et non seulement la première.
6 – Modifiez aussi la classe de test et exécutez (2 guitares doivent correspondre)
7 – Assurez-vous que l'ensemble des tests d'intégration fournis passent au vert.

UN PEU DE COURS…

Les **types énumérés** sont disponibles en C, C++, Java version 5.0 et +, Perl 6…
Ils sont utiles car ils permettent aux classes et méthodes les utilisant d'être protégés de toute valeur non définie dans l'enum. Plus de fautes de frappe ou d'orthographe sans avoir une erreur de compilation.

Troisième version : appliquer les principes de base OO

TO DO :
1 – Recherchez d'éventuels problèmes dans l'outil de recherche de la version 2.
2 – Analysez plus précisément la méthode *search*(). Qu'est censée faire cette méthode ? (établissez une description textuelle)
3 – Trouvez l'objet « bancal ».
4 – Créez une nouvelle classe PrefGuitare (*GuitarSpec* en anglais) et mettez-y toutes les propriétés et méthodes de *Guitar* que vous pensez nécessaires. Encapsulez un objet de cette classe pour chacun de vos objets *Guitar*.
5 – Modifiez les classes *Guitar* et *Inventory* (le constructeur de *Guitar* reste **inchangé** dans cette version, idem pour la signature de la méthode *addGuitar*() de *Inventory*).
6 – Modifiez aussi la classe de test et exécutez.
7 – Assurez-vous que l'ensemble des tests d'intégration fournis passent au vert.

UN PEU DE COURS…

Une **description textuelle** du problème que vous essayez de résoudre vous garantit que votre conception correspondra à la fonctionnalité voulue de votre application.

UN PEU DE COURS…

Conseils pour repérer l'**objet bancal** :
1 – les objets sont censés faire ce qu'indique leur nom (prendreBillet() n'est pas une méthode pour un objet Avion)
2 – chaque objet doit représenter un concept unique (et ne doit pas servir à plusieurs utilisations) : exemple d'un Canard représentant l'animal, comme le journal, ou le canard en plastique !
3 – les propriétés inutilisées nous trahissent toujours (si des propriétés prennent souvent les valeurs null alors peut-être que votre objet fait plus que ce qu'il devrait faire…)

UN PEU DE COURS…

L'**encapsulation** permet de subdiviser votre application en parties cohérentes. En séparant les différentes parties, on peut alors changer l'une sans modifier toutes les autres. En général on encapsule les parties de l'application qui peuvent varier indépendamment des parties qui vont rester stables. L'information d'une partie de l'application est ainsi protégée des autres parties.
Vous connaissiez déjà l'encapsulation des données de vos classes en les rendant privées, maintenant vous pouvez aussi encapsuler un **ensemble complet de propriétés** (comme les préférences de guitare) ou même des **comportements** (comme la façon de voler d'une espèce de canard).
Chaque fois que vous voyez du code dupliqué, cherchez où l'encapsuler.

L'**encapsulation** est l'un des principes OO. Les autres sont l'héritage, le polymorphisme et l'abstraction.

Quatrième version : vers une conception facile à maintenir et réutiliser

TO DO :
1 – Regardez de plus près le code de la méthode *search*(). Que changeriez-vous ?
2 – Imaginez que l'on souhaite ajouter une nouvelle caractéristique « nbcordes » pour une guitare. Que devriez-vous changer dans votre code ? Combien de classes faut-il modifier ?
3 – Nous avons encapsuler les paramètres de guitare, il faut aussi les isoler du reste de l'outil de recherche !
3.1 – ajoutez une propriété *numStrings* (nbCordes) et une méthode *get* correspondante à *GuitarSpec*
3.2 – modifiez *Guitar* pour que les propriétés de *GuitarSpec* soient encapsulées à l'extérieur du constructeur
3.3 – changez la méthode *search*() dans *Inventory* pour déléguer la comparaison de 2 objets *GuitarSpec* à la classe *GuitarSpec*
4 – Modifiez aussi la classe de test et vérifiez.
5 – Assurez-vous que l'ensemble des tests d'intégration fournis passent au vert.

UN PEU DE COURS…

Délégation : action d'un objet renvoyant une opération vers un autre objet pour qu'elle soit réalisée pour le compte du premier objet. La délégation permet au code d'être plus facilement réutilisable. Vos objets s'occupent ainsi de leur propre fonctionnalité plutôt que de disséminer le code gérant leur comportement un peu partout dans le code de l'application.

La méthode « equals() » est l'exemple le plus courant de délégation.

Avec la délégation les objets sont plus indépendants les uns des autres : ils sont lâchement couplés, chaque objet a une responsabilité unique.

Cinquième version : parce que le changement est inévitable…

On souhaite désormais gérer aussi des mandolines de manière identique aux guitares !

TO DO :
1 – Mandoline et Guitare ont-elles quelque chose en commun ? Que pensez-vous d'utiliser l'héritage ? Une classe abstraite ? Une interface ?
2 – Modélisez à l'aide d'un diagramme de classe UML votre nouvelle application.
3 – Codez cette version :
2.1 – Créez une nouvelle classe abstraite *Instrument*. Mettez-y les propriétés communes à tous les instruments.
2.2 – Reprenez *Guitar* pour qu'elle étende *Instrument* et utilise *GuitarSpec* dans son constructeur.
2.3 – Créez *Mandolin* de manière identique (elle étend *Instrument* et utilise *MandolineSpec* dans son constructeur).
2.4 – Créez une classe abstraite *InstrumentSpec* contenant les spécifications communes.
2.5 – Codez *GuitarSpec* et *MandolinSpec* (les méthodes *matches*() utiliseront la méthode *matches*() de la super-classe et effectueront des vérifications supplémentaires pour s'assurer que la pref reçue en paramètre est du bon type et correspond aux propriétés spécifiques aux guitares/mandolines).
2.6 – Mettez à jour la classe *Inventory* pour travailler avec plusieurs types d'instruments au lieu de seulement les guitares (vous allez avoir besoin d'une autre méthode de recherche qui s'occupera des mandolines).
2.7 – Ne modifiez pas la classe de test mais regardez s'il est facile d'ajouter encore d'autres instruments : Que faut-il modifier/créer à chaque nouvel ajout d'instrument ?
4 – Assurez-vous que l'ensemble des tests d'intégration fournis passent au vert.

UN PEU DE COURS…

Une **conception** se construit par cycles. Il faut **être capable de changer** ses propres conceptions, de la même façon que celles que vous héritez d'autres programmeurs.

La fierté tue une bonne conception ! N'ayez pas peur d'examiner vos propres décisions de conception et de les améliorer, même si cela implique de revenir en arrière.

La plupart des **bonnes conceptions** viennent de l'analyse des **mauvaises conceptions** ! N'ayez donc pas peur de faire des erreurs puis de tout restructurer.

UN PEU DE COURS…

La **cohésion** mesure le degré de **connectivité** entre les éléments d'un module, d'une classe ou d'un objet donné. Plus la cohésion de votre logiciel est forte, plus les **responsabilités** de chaque classe individuelle sont **bien définies et reliées** dans votre application. Chaque classe a un ensemble très spécifique d'actions de **proximité** qu'elle accomplit.

Une classe **cohésive** fait une chose vraiment bien et n'essaie pas de faire ou d'être quelque chose d'autre.