Concept de l'ACOO (Analyse et Conception Orientées Objet)

Récapitulatif des informations issues du livre L'ACOO de la collection Tête la Première des éditions O'Reilly. Il est basé sur le langage Java.

Qu'est-ce qu'un bon logiciel ?

- Assurez-vous que le logiciel fait ce que le client veut qu'il fasse (fonctionnalités).
- Appliquez les principes OO de base pour ajouter de la souplesse (l'assemblage du logiciel a un sens (Encapsultion).
- Oeuvrer pour une conception facile à maintenir et à réutiliser (Encapsulation, délégation pour que les objets soient moins dépendants les uns des autres).

Analyse et Conception UML

Cas d'utilisation

Pour concevoir une application, il faut commencer par dresser **le cas d'utilisation** (il peut y en avoir plusieurs si l'application fait plusieurs choses). Il contient un chemin principal et peut contenir un ou plusieurs chemins alternatifs. C'est la transcription ordonnée et en français des exigences du client.

Diagramme des cas d'utilisation

Dans le cas d'une grosse application, dégager la vision d'ensemble avec un **diagramme des cas d'utilisation** (le plan directeur de notre système) et ensuite dresser les cas d'utilisation. On va du général au particulier.

Diagramme de classe

Ensuite seulement, il faut dresser le **diagramme de classe UML**. Il est plus facile à trouver puisqu'avec l'analyse textuelle du cas d'utilisation on se rend compte qu'en général les sujets donnent des classes et les verbes donnent les méthodes.

Composition

La composition vous permet d'utiliser le comportement d'une famille d'autres classes et de changer ce comportement pendant le fonctionnement de l'application. Avec la composition, l'objet composé des autres comportements **possède** ces comportements. Quand l'objet est détruit, **tous ses comportements le sont aussi**.

Agrégation

L'agrégation désigne le fait qu'une classe est utilisée comme une partie d'une autre classe, mais continue à exister en dehors de cette autre classe. La crème glacée, la banane et les cerises existent en dehors du banana split. Enlevez la jolie coupe en verre et vous avez toujours les composants séparés.

Composition ou agrégation ?

Le moyen le plus simple : **Est-ce que l'objet dont je veux utiliser le comportement existe en dehors de l'objet qui utilise son comportement ?**

Éléments de la programmation objet

Les types enumérés

Les types enumérés permettent de définir des constantes à utiliser à la place de valeurs un peu aléatoires. Cependant, il faut que les données ne soient pas trop importantes. Par exemple, faire correspondre des majuscules et des minuscules de façon à éviter les fautes de frappes. On met un switch dans la méthode toString() en java.

L'encapsulation

- Permet de protéger les classes des modifications inutiles (getter/setter).
- Permet de subdiviser l'application en parties cohérentes.
- Doit être utilisé pour ce qui risque de changer.

La délégation

La **délégation**, c'est quand un objet a besoin de réaliser une certaine tâche, et plutôt que de faire cette tâche directement, il demande à un autre objet de s'en occuper. La délégation protège vos objets des changements de mise en oeuvre des autres objets de votre logiciel.

Elle rend le code plus facile à réutiliser. Elle permet d'avoir des objets qui s'occupent de leur propre fonctionnalité plutôt que de disséminer le code qui gère le comportement d'un objet unique un peu

partout dans l'application. L'exemple le plus courant en Java est la méthode equals(). Au lieu de faire une méthode qui compare deux objets, elle appelle equals() et on obtient simplement une réponse vrai ou faux.

Une application lâchement couplée

La délégation aide l'application à rester **lâchement couplée**. Cela signifie que les **objets sont indépendants les uns des autres**. Les changements d'un objet n'obligent pas à faire tout un tas de changement dans d'autres objets.

L'héritage

On met en place l'**héritage** afin de factoriser des champs et des méthodes pour plusieurs sous-classes. On créé une sous-classe parce que le **comportement** de la sous-classe est différent de la super-classe. Si un Instrument est différent d'une guitare, il faut sous-classer Guitare. Par contre, si le **comportement d'un Instrument est identique par rapport à une Guitare**, il faut penser à **ne pas mettre en place l'héritage** et ne faire qu'une seule classe.

Classe abstraite

La **classe abstraite** définit un comportement et les sous-classes implémentent ce comportement.

L'interface

- Coder avec une interface à la place d'une implémentation rendra votre logiciel plus facile à étendre.
- En codant avec une interface, votre code fonctionnera avec toutes les sous-classes de l'interface, même celles qui ne sont pas encore créées.

La collection (List, Vector, Map..)

Si vous avez un ensemble de propriétés qui varient selon vos objets, utilisez **une collection, une Map** par exemple, pour conserver ces propriétés de façon dynamique. Vous enlèverez beaucoup de méthodes de vos classes et vous n'aurez plus à modifier votre code quand les nouvelles propriétés seront ajoutées à votre application. Ceci peut même enlever des notions d'héritages qui semblaient pourtant intéressantes mais au final qui ne le sont pas (Instrument–>Guitare ou Instrument–>Mandoline donne juste Instrument par exemple).

Classe cohésive

Une **classe cohésive** fait une chose vraiment bien et n'essaie par de faire ou d'être quelque chose d'autre. La cohésion est une raison unique pour qu'une classe change. Un logiciel fortement cohésif est lâchement couplé.

Les méthodes des classes doivent être liées au nom de la classe. Si ce n'est pas le cas, elle est peut-être à déplacer dans une autre classe.

Cas de tests

- Chaque tests doit avoir un identifiant et un nom. Pas test1, test2 mais TestPropriete ou TestCreation.
- Chaque cas de test doit tester une chose spécifique. Une fonctionnalité peut impliquer une méthode, deux méthodes ou même plusieurs classes...mais pour commencer, occupez-vous de fonctionnalités très simples, une à la fois. Si on fait plusieurs méthodes de tests bien spécifiques, après on peut de toute façon les exécuter à la suite pour effectuer un test global.
- Chaque cas de test doit avoir des données entrantes que vous fournissez. ex : initialiser la masse à 15. C'est écrit en dur dans le test.
- Chaque cas de test doit avoir des données sortantes que vous vérifiez. Par exemple initialiser la masse à 15 et vérifier que la masse=15. Si ce n'est pas le cas le test échoue.
- La plupart des cas de test ont un état de départ. Par exemple, dans le cas d'une base de données, il faut initialiser la connexion avant de pouvoir effectuer des tests dessus.

Conseils importants

- Ne jamais dupliquer du code. Si c'est le cas, on peut faire autrement.
- L'une des meilleures façons de vérifier qu'un logiciel est bien conçu est d'essayer de le modifier.
- La plupart des bonnes conceptions viennent de l'analyse de mauvaises conceptions.
- Ne pas avoir peur de faire des erreurs et de tout restructurer.
- On résout les gros problèmes de la même façon que les petits. Vous pouvez résoudre un gros problème en le fractionnant en de nombreuses petites parties fonctionnelles puis en travaillant sur chaque partie séparément.
- Parfois le meilleur moyen d'écrire du bon code est de vous retenir d'écrire du code aussi longtemps que vous le pouvez.
- Si vous préférez la délégation, la composition et l'agrégation plutôt que l'héritage, votre logiciel sera généralement plus souple et plus facile à maintenir, étendre et réutiliser.

Pour un gros projet

- Il faut commencer par regarder à quoi il ressemble (c'est la conformité) et à quoi il ne ressemble pas (c'est la variabilité). En règle général, au moment de la conception, la conformité est une super-classe et la variabilité constitue les sous-classes.
- Commencer par les caractéristiques d'un système est très utile quand on a pas trop de détails

et qu'on ne sait pas trop par où commencer.
- Obtenez les caractéristiques du client, puis déduisez les exigences dont vous avez besoin pour implémenter ces caractéristiques.
- Retarder toujours les détails le plus longtemps possible. Il n'est donc pas très intéressant de faire un cas d'utilisation tout de suite mais plutôt dans un second temps, après avoir travaillé sur les choses générales.
- Créer le diagramme UML des cas d'utilisations pour dégager l'ensemble du projet. Attention les acteurs peuvent ne pas être que des humains (ex : le jeu).
- L'analyse de domaine permet de représenter un système dans un langage que le client comprend. Il permet de vérifier vos conceptions.
- Découper l'application en grand domaine dans des modules (Unités, Plateau, Jeu, Contrôleur, Outils) plus petits.
- Appliquer des designs patterns.
- Dans l'application, les choses qui sont vraiment importantes sont significatives dans l'architecture et vous devez les aborder en PREMIER. Trois questions pour savoir si quelque chose est significatif et qu'il va falloir le traiter en premier :
    - Cette caractéristique est-elle vraiment le noyau ? Pouvons-nous imaginer le système sans cette caractéristique ? Si non, cette caractéristique appartient à l'essence du système. L'essence d'un système est ce qu'est le système à son niveau le plus basique.
    - Si on n'est pas certain de la signification réelle d'une caractéristique. Il vaut mieux consacrer du temps au début du développement.
    - Les caractéristiques qui ont l'air difficiles à implémenter car impliquent des tâches de de programmation complètement nouvelles, sont à traiter en priorités pour éviter quelles créées des problèmes plus tard.
- Une fois les caractéristiques clés décelées, il faut choisir celles qui présentent le moins de RISQUES (des retards sur les délais, des problèmes autres...).
- Construire un scénario d'une simple exécution. Il s'apparente un peu à écrire sur papier ce qu'on ferait dans un test unitaire.

Principes de conception

Le principe d'ouverture-fermeture ou OCP (Open-Closed Principle)

L'objectif est de **permettre le changement**, mais **sans avoir à modifier du code existant**. Les classes doivent être ouvertes à l'extension et fermées à la modification. Par exemple, la super-classe a une **méthode qui n'est pas changeable**, par contre les sous-classes peuvent **récrire cette même méthode** pour l'adapter à un besoin particulier. Autre exemple, si on a plusieurs méthodes privées dans une classe, elle sont fermés à la modification. On peut ajouter des méthodes publiques qui invoquent ces méthodes privées. **On étend** le comportement des méthodes privées **sans les changer**.

Le principe de non duplication ou DRY (Don't Repeat

Yourself)

- Éviter le code dupliqué en rendant les **éléments communs abstraits** et en les plaçant tous au même endroit.
- La non duplication : **UNE exigence à UN endroit**.
- L'intérêt du principe de non duplication est d'attribuer une **place unique** et **appropriée** pour chaque information ou comportement de votre système.

Le principe de responsabilité unique ou SRP (Single Responsability Principle)

Chaque objet de votre système ne doit avoir qu'une seule responsabilité et tous les services de cet objet doivent s'efforcer d'assumer cette unique responsabilité. Synonyme : **la cohésion**. Vous avez implémenté le principe de responsabilité unique correctement si chacun de vos objets n'a **qu'une seule raison de changer**. Pour analyser le principe de responsabilité unique d'une classe, il faut réaliser le test suivant : **Le/la/l'** NOMCLASSE NOMMETHODE **lui/elle-même**.

Cette méthode mémo-technique permet de déceler si une classe respecte le principe de responsabilité unique. Par exemple avec un classe Automobile et les méthodes suivantes :

demarrer(), arreter(), changerPneus(), conduire(), getHuile()

Si on applique le test : L' AUTOMOBILE demarre elle-même. –>oui L' AUTOMOBILE change ses pneus elle-même. –>non …

Attention cette méthode n'est pas infaillible. L'expérience et le bon-sens est de mise.

Le principe de substitution de Liskov ou LSP (Liskov Substitution Principle)

Les sous-types doivent pouvoir être substitués à leurs types de base. Tout le principe de substitution de Liskov tourne autour d'une **bonne conception de l'héritage**. Quand vous héritez d'une classe de base (Plateau), vous devez pourvoir lui substituer les sous-classes (Plateau3D) de cette classe de base (Plateau) sans que cela tourne à la catastrophe. Sinon, l'utilisation de l'héritage n'est pas bonne. Ici la classe Plateau3D ne peut pas être substituée à Plateau parce qu'aucune des méthodes de Plateau ne fonctionne correctement dans un environnement 3D.

Exemple avec une classe Plateau et une autre Plateau3D qui hérite de Plateau. Plateau avec : getCase(), addUnite(), removeUnite()

Plateau3D avec les mêmes méthodes appliquées à la 3D : getCase(), addUnite(), removeUnite()

Comme elle hérite, en réalité voici ce qu'on obtiens : Plateau3D avec toutes les méthodes : getCase(), getCase(), addUnite(), addUnite(), removeUnite(), removeUnite()

L'héritage et le principe de substitution de Liskov indiquent que toute méthode de Plateau doit pouvoir être utilisée dans Plateau3D ou d'une autre manière que Plateau3D peut remplacer Plateau

sans problème. Plateau plateau=new Plateau3D ne pose pas de problème à la compilation mais si on utilise cette instance Plateau3D comme un Plateau c'est la panade.

Si vous voulez utiliser la fonctionnalité d'une autre classe, mais sans changer cette fonctionnalité, pensez à utiliser la délégation plutôt que l'héritage.

Boîte à outils

Exigences

- De bonnes exigences permettent à votre système de fonctionner comme les clients le souhaitent.
- Vérifiez que vos exigences recouvrent bien toutes les étapes du cas d'utilisation de votre système.
- Utilisez vos cas d'utilisation pour découvrir ce que vos clients ont oublié de vous dire.
- Vos cas d'utilisation révèleront toute exigence manquante ou incomplète que vous devez ajouter à votre système.
- Vos exigences vont toujours changer (et grandir) avec le temps.

Analyse et conception

- Un logiciel bien conçu est facile à modifier et à étendre.
- Utilisez des principes de base OO comme l'encapsulation et l'héritage pour accomplir votre logiciel.
- Si une conception n'est pas souple, alors **CHANGEZ-LA!** Ne restez jamais sur une mauvaise conception, même si c'est votre mauvaise conception.
- Chacune de vos classes doit être cohésive : chacune devrait se concentrer pour bien faire **UNE SEULE CHOSE**.
- Travaillez toujours à une cohésion plus forte au fur et à mesure que vous avancez dans le cycle de vie de votre conception.

Résoudre de gros problèmes

- Écoutez le client et comprenez ce qu'il veut que vous construisiez.
- Faites une liste des caractéristiques dans un langage compréhensible pour le client.
- Vérifiez que vos caractéristiques correspondent à ce que le client veut réellement.
- Faites les plans directeurs du système en utilisant des diagrammes de cas d'utilisation (et des cas d'utilisation).
- Divisez le gros système en plusieurs sections plus petites.
- Appliquez des design patterns aux petites parties du système.
- Utilisez des principes de base ACOO pour concevoir et coder chaque petite partie.

Principes OO

- Encapsuler ce qui risque de changer
- Coder avec une interface plutôt qu'avec une implémentation.
- Chaque classe dans votre application ne doit avoir qu'une seule raison de changer.
- Les classes doivent se concentrer sur le comportement et la fonctionnalité.
- Les classes doivent être ouvertes à l'extension mais fermées à la modification (principe d'ouverture-fermeture).
- Évitez le code dupliqué en enlevant les choses communes, en les rendant abstraites et en les plaçant dans un endroit unique (principe de non duplication).
- Tout objet de votre système ne doit avoir qu'une seule responsabilité et tous les services de l'objet doivent être orientés vers l'accomplissement de cette responsabilité unique (principe de responsabilité unique).
- Les sous-classes doivent pouvoir être substituées à leurs classes de base (principe de substitution de Liskov).

Pratique de la programmation

- La **programmation par contrat** établit un accord sur le comportement de votre logiciel que vous et les utilisateurs de votre logiciel acceptez de respecter.
- La **programmation défensive** ne fait pas confiance aux autres logiciels et fait beaucoup de vérification de données et d'erreurs pour éviter que les autres logiciels ne vous donnent des informations dangereuses ou erronées.

Approche de développement

- Le **développement orienté cas d'utilisation** s'occupe d'un cas d'utilisation isolé dans votre système et a pour but de terminer le code pour implémenter le cas d'utilisation entier, incluant tous ses scénarios, avant de passer à quoi que ce soit d'autre dans l'application.
- Le **développement orienté fonctionnalités** se concentre sur une caractéristique isolée et code tout le comportement de cette caractéristique, avant de passer à quoi que ce soit d'autre dans l'application.
- Le **développement orienté tests** écrit des scénarios de test pour une fonctionnalité avant d'écrire le code pour cette fonctionnalité. Ensuite vous écrivez le logiciel pour réussir tous les tests.
- Un bon développement de logiciel **incorpore généralement tous ces modèles** de développement à **différentes étapes du cycle de développement**.

Récapitulatif des étapes d'un projet

- On nous fournit un descriptif du projet.
- Liste des caractéristiques.
- Diagrammes UML des cas d'utilisation.
- Diviser le problème en packages logiciels.
- Dresser les cas d'utilisation de chaque partie du diagramme UML des cas d'utilisation. Parfois, il

faut comprendre le problème avant de les rédiger.

- **Sur un seul cas d'utilisation**.
  - Analyse textuelle des cas d'utilisation pour dégager les classes et les méthodes en fonction des noms (sujet des phrases) et des verbes. Cela fait partie de l'analyse de domaine .
  - Dresser le premier diagramme de classes UML. On est à ce stade dans la conception préliminaire.
  - Passer à l'implémentation. Le diagramme des classes peut évoluer. L'objectif quand même de ne pas avoir à tout remettre à plat, mais l'ajout d'une classe ou le déplacement de méthode est tout à fait possible en cours de développement.
  - Après le premier cas d'utilisation codé, itérer sur le deuxième cas d'utilisation et réaliser de nouveau cette sous-partie.