INFO-F-204 - Analyse et méthode - C. Hernalsteen
Résumé du cours

Rodrigue Van Brande

28 décembre 2014

Table des matières

1 Le software engineering

1.1 Introduction

Dans les années 70 on s'est rendu compte qu'on avait besoin de méthode de développement. Mais plusieurs problèmes se posent : Le client peut vouloir changer quelque chose au cours du projet, on doit pouvoir estimer le temps que ça va prendre, etc... Deux plus grosses causes d'échec : Compréhension du client et le travail en équipe. Les diagrammes UML vont aider, et clarifier les choses avec le client et son équipe. Pourquoi ne peut on pas s'inspirer de l'ingénierie civile (ex :construction d'un pont) ? Car un pont on le voit, pas le logiciel et ce dernier évolue continuellement contrairement au pont. Problème : Évolution du logiciel dégradante ⇒ Chaque modification devient de plus en plus compliquée.

1.2 Méthode Waterfall

On procède par phase :

Requirements Collection Rencontre avec le client et note de tous les besoins (Risques : Documentations incomplètes, inexactes et ambiguës)

Analysis Les analystes définissent les besoins, les écrans, ... (Risques : Fournir une spécification qui ne correspond pas aux besoins du client.)

Design Architectes conçoivent l'architecture de l'application. Conception de diagrammes et choix des librairies.

Implementation Les codeurs développent leurs modules.

Testing Assemblage et livraison.

Cela a donné de bon résultats mais problèmes de communication car elle est faite par documents plutôt que par la parole. Les codeurs n'ont pas de recul et ne peuvent détecter des problèmes potentiels. Si il y a une erreur il faut corriger dans chacune des étapes. Un logiciel est long à développer et les besoins du client peuvent changer dur le développement.

Avantages : Très contrôlé, panifiable, des documents décrivant l'entièreté de l'application.

1.3 Méthodes incrémentales et itératives

On procède par incréments. On livres des morceaux de logiciel au client petit à petit. C'est un enchaînement de mini waterfall. Cette méthode permet d'avoir plus de retour du client. On ne fait pas une grosse analyse, on développe juste complètement un module qu'on va montrer au client. Il vérifie et apporte ses corrections qui seront facilement faisables. On fait cela à chaque itération. Une itération dure en moyenne entre 2 et 4 semaines.

Avantages : Le client s'implique et le projet à de grosses chances de réussites.

1.4 Bibliothèques

Avant lorsqu'on codait, on écrivait quasiment tout, mais maintenant on utilise beaucoup de frameworks et de composant. On doit trouver tout ces composants lors de l'analyse.

2 L'orienté objet

2.1 Introduction

2.2 Types de données abstraits

On sépare l'implémentation de la spécification, ce qui permet de développer plein de petits modules plutôt qu'un énorme bloc de code. On peut modifier des bouts de code à un endroit sans devoir modifier toute l'application du coup la maintenance est facile.

Code Client : Code qui dépend d'un autre

2.3 Les objets

On écrivait le code qui traitait des données. Les objets réunissent les deux dans "une boîte".

2.3.1 Ce qu'est un objet

Un objet a une réalité et est unique (!= d'une classe). Il a un état et est modifiable. Objet = Identité + état + comportement Un objet reçoit des messages (se déplacer, changer d'état, ...) et va chercher le code correspondant. L'objet cache ses données, on ne peut passer que par les messages.

2.3.2 Interaction entre objets

Si plusieurs objets, il peuvent interagir (Exemple : En passant un objet en paramètre d'un message).
Message $\neq$ Méthode $\Rightarrow$ On envoie un message à un objet, la méthode c'est le code que l'objet exécute lorsqu'il reçoit le message.
Method lookup : Ce qui trouve la méthode correspondant au message.

2.3.3 Polymorphisme

Message $\neq$ d'un appel de fonction. Lors d'un message, on dit l'objet sur lequel ça va intervenir, le a fonction peut porter sur n'importe quoi.
Polymorphisme : On envoie un même message à des objets de forme différente et la réponse pourra être différente.

2.3.4 Stockage des méthodes et classes

Classe à une réalité en mémoire, elle contient le code mais pas les données.
Un objet à aussi une réalité en mémoire, elle contient les données mais pas le code.
Un objet est une instance d'une classe.
Un objet connaît toujours sa classe, lorsqu'il reçoit un message il lui demande la méthode qui convient.

2.4 L'héritage

Héritage On décrit un objet abstrait, puis des objets qui en hérite et qui récupère tout ça.

Généraliser Opération de créer un objet plus abstrait dont on va hériter.

2.4.1 Principe de l'héritage

On dérive d'une classe parente et les sous-classes héritent de tous ses attributs et méthode.
Les sous-classes peuvent surcharger les méthode pour les adapter à leurs besoins $\Rightarrow$ Overriding.
But : Mettre de la structure qui amènera au polymorphisme et un code facilement maintenable.
Pour savoir si B doit hériter de A, il faut pouvoir se dire : B est une sorte de A.

2.4.2 Types d'héritage

Héritage simple Une classe hérite d'une seule et une seule classe. Cet héritage à une structure d'arbre, chaque classe a un parent et des enfants.

Héritage multiple Permet d'hériter directement de plusieurs classe. Les langages récents ne le permettent plus car plus de problème que de solution.

2.4.3 Method lookup

Lorsqu'un objet reçoit un message, il connaît sa classe et de laquelle elle hérite. Il va chercher si la méthode est dans sa classe, sinon il regarde dans celle héritée et ainsi de suite.
Surcharge de méthodes : Une classe peut redéfinir une méthode dont elle hérite.

2.4.4 self/this et super

Mots clés spécials :

super représente la classe parente.

this représente la classe qui a reçu le message qu'on traite.

super est statique, on sait immédiatement ce que c'est.

shis est dynamique, lors de la compilation on ne sait pas quel objet il représente. Car il peut représenter la classe elle même ou une classe fille.

2.5 Polymorphisme

2.5.1 Références

Référence (ou pointeur) : permet de nommer un objet et est différente de l'objet lui-même.
Si on a une référence vers un objet de type A, on peut la faire pointer sur un objet de type B.
L'objet de type B étant une sorte de A, on pourra utiliser les messages connus de A (mais pas ceux de B).
C'est grâce à ce principe qu'on va pouvoir utiliser le polymorphisme.

2.5.2 Exemple

Imaginons qu'on veuille réaliser une application pour faire des dessins. On a des boutons permettant de choisir un carré, un triangle ou un rond. On veut programmer ça, avec la pensée objet.

On a une classe FORME qui va contenir tout ce qui est commun à une forme (taille, position, couleur). On a les classes ROND, CARRE et TRIANGLE qui héritent de FORME. On a une classe DESSIN qui contient des FORMEs et qui n'hérite de personne.

On crée une méthode dessiner qui ne fait rien à la classe FORME (elle est juste la pour dire qu'elle existe). Et on la surcharge dans les classes TRIANGLE, CARRE, Rond. Comme ça on peut utiliser (FORME f).dessiner(), ce qui va appelé la méthode de la classe correspondante (soit de TRIANGLE, soit de CARRE, soit de ROND).
Délégation : Le fait qu'un objet en utilise un autre.

2.5.3 Le C++

Le C++ laisse beaucoup de liberté, et il y a plein de piège dans son orienté objet. Les fonctions ne sont pas virtuelle par défaut (si on surcharge une méthode de A dans B, et qu'on envoie le message à une référence A qui pointe vers un objet B c'est la méthode de A qui sera appelé.) Pour résoudre le problème, il faut rajouter **virtual** devant la méthode de A.

Modificateur de visibilité : Lors de l'héritage on peut préciser si il est public privé ou protégé. Les autres types d'héritage que public masquent les attributs et méthodes de A. ⇒ Cela rompt l'orienté objet.

C++ permet de créer des objets sur la pile et pas seulement par référence. ⇒ Empêche tout un tas de bonnes choses de l'orienté objet.

this est implicite et il n'existe pas de mot clé **super**, on nomme explicitement la classe.

3 Héritage avancé

3.1 Introduction

3.2 Overriding de méthodes trouvées en Framework

Le super ne doit être appelé que dans des méthode redéfinies. Et ne doit servir que à appeler la méthode de la classe parent.

3.3 Classes abstraites

Une classe abstraite est une classe qu'on ne peut instancier directement. Cela permet d'implémenter des interfaces. Elle fournit une abstraction qui permettra de mettre des choses en commun entre ses filles. Une classe abstraite l'est si elle a au moins une méthode abstraite. (**virtual** foo() = 0 ;).

Si la classe fille ne veut pas être abstraite elle doit redéfinir toutes les méthodes virtuelles pures (=abstraite) de la mère. Si elle le fait, on appelle ça une concrétisation.

3.4 Où placer les méthodes

Quand on ajoute une méthode à une classe, on le met le plus haut possible dans la hiérarchie des classes. L'implémentation doit aussi se baser le plus possible sur les méthodes déjà existantes.

4 Aperçu d'UML

4.1 Introduction

UML = **U**NIFIED **M**ODELLING **L**ANGUAGE
Il s'agit d'une boîte à outils, le processus est propre à celui qui l'utilise.
C'est un langage de modélisation.
Modéliser : Avant d'implémenter on va faire des plans de construction.
Cela aide à réfléchir et à communiquer.

4.2 Bits de l'UML

Il a été fait pour être automatisable et lisible par un humain.
Il est générique, il peut s'appliquer à tout type d'application.

4.3 Concepts

4.3.1 Les vues

Il existe 5 vues dont chacune est définie par un certain nombre de diagrammes :

Use case montre les fonctionnalités du système tel qu'elles sont perçues par un acteur externe, qui peuvent être des utilisateurs ou d'autres systèmes (Diagrammes : Use case, Activity).

Logical view définit les fonctionnalités du système, les informations manipulés, ... (Diagrammes : Class Diagram, State, Sequence, Collaboration, Activity)

Component view indique comment le code est mis en boite. Les classes, les bibliothèques, fichiers de configurations, BDD, ... (Diagrammes : Diagramme de composants)

Deployment view indique là où les composants vont s'exécuter. Le déploiement du système dans l'architecture physique avec les ordinateurs et les appareils.

Concurrency view décrit comment les composants interagissent entre eux. Point de vue dynamique du système. (Diagramme de séquence)

4.3.2 Les diagrammes

Il y a au total 9 diagrammes.

Use case Basique. Ce sont des dessins. On définit ce que les acteurs peuvent faire.

Class On représente ici les classes à un haut niveau. On n'utilise que le nom des classes et les liens qui les unissent.

State Représente les états d'un programme. On a un état initial (un rond noir) Puis des états reliés par des flèches.

Sequence Représente des séquences de communication entre objets. Le temps se lit du haut vers le bas.

Collaboration Correspond au sequence diagram, met on met en évidence la structure plutôt que le temps.

Object Donne un exemple pour aider à la compréhension.

Activity Représente le coté comportemental de l'application. On y montre des étapes.

Component Identifie les composants et les relient.

Deployment Constitué de gros cube. Un gros cube est une unité de traitement ou de stockage, un serveur ou une bdd.

5 Meta Model UML

5.1 Introduction

Tout les composants qu'on peut utiliser en UML ont été écrits en UML.
Un méta-model est un modèle qui en décrit un autre.
Chaque diagramme est une instance du méta-model.

5.2 L'élément

Tout ce qu'on manipule en UML est un élément.
Chaque élément peut être contenu dans un paquet.

5.3 Mécanismes

Stéréotype permet d'étendre UML. (format : « stéréotype »). On veut par exemple montrer que certaines classe servent à la gestions des freins, on leur ajoute le stéréotype «gestion des freins».
Tagged values est une association entre un nom et une valeur. (Exemple : créateur : machin)
Notes donne des bouts de texte, des commentaires.
Contraintes entouré par des accolades. Permet de représenter des contraintes sur les attributs, valeurs, associations, ...
Dépendances dit qu'un élément dépend d'un autre (flèche en pointillé)
Types prédéfinis sont les types qu'on connaît (bool, string, int, ...)
Multiplicité représente un nombre lors des associations.
Package rassemble des éléments.

5.4 Diagrammes des classes

On y représente les classes utilisées dans le projet. Représente les différentes informations que l'application va devoir manipuler.

5.4.1 Représentation d'une classe

Représenté par un grand rectangle contenant en haut le nom de la classe, puis les attributs, puis les méthodes.

5.4.2 Règles de représentation

Classe commence par une majuscule.
Méthode doit être un nom, non une action ou un verbe.
Attribut commence par une minuscule et ont un type ou une visibilité et peut avoir une valeur par défaut.
Méthodes ont une signature (type de retour, le nom, 0 ou plusieurs paramètres) et une visibilité.

5.4.3 Relations de base

4 types de relations/associations entre les classes.

Usage Utilisation

Inheritance L'héritage

Refinement La même classe avec plus de détails.

Réalisation Les classes abstraites qu'on réalise.

Pour la relation d'usage on peut (pas obligatoire) avoir une direction et une multiplicité.
Relations ternaires : Relations entre trois classes. **Role name** : Le nom du rôle joué par la classe dans la relation (Exemple : maître ou esclave.)

5.4.4 Relations étendues

Association Une association représente une relation quelconque entre deux classes ; en général, les objets d'une classe se servant de ceux d'une autre classe. (Exemple : une personne utilise un ordinateur.) Un simple trait entre les deux classes, peut être composé d'une flèche pour indiquer une direction.

Agrégation : Une agrégation permet de définir une entité comme étant liée à plusieurs entités de classe différente ; décrit une association de type « fait partie de », « a ». (Exemple : Une flotte constituée de plusieurs bateau. Une flotte n'est plus une flotte sans bateau.) Représenté par un diamant vide du coté de la classe qui agrège (ici la flotte).

Composition : Comme l'agrégation mais en plus fort. Elle fait partie entière de l'objet. Si on détruit le tout, on détruit tous les éléments. (Exemple : Si on détruit le livre, les pages n'ont plus de raison d'être, contrairement aux bateaux sans la flotte.) Représenté par un diamant noir du coté de la classe principale (ici le livre).

Généralisation : Héritage, flèche pointant sur la classe parente.

Raffinement : Relation d'une classe vers elle même. On représente ça avec « refine»

Réalisation : Dit qu'une classe en réalise une autre.