LOG210 Analyse et conception de logiciels

Christopher Fuhrman Yvan Ross

January 20, 2020

Contents

1 Analyse et conception de logiciels

Voici le descriptif du cours, selon le plan de cours:

> À la suite de ce cours, l'étudiant sera en mesure :
>
> - de maîtriser et appliquer des patrons de conception logicielle;
> - de concevoir un logiciel orienté objet en appliquant un ensemble de principes et des méthodes heuristiques de génie logiciel;
> - de réaliser un logiciel en suivant un processus itératif et évolutif incluant les activités d'analyse et de conception par objets.
>
> Méthodes et techniques de modélisation orientés objet, langage de modélisation, cas d'utilisation, analyse orientée objet, modèle du domaine, conception et programmation orientées objet, principes GRASP, patrons de conception, processus itératif et évolutif.
> Séances de laboratoire axées sur l'application des notions d'analyse, de conception et de programmation orientées objet vues en classe. Mise en œuvre d'un modèle d'objet à partir d'une spécification de logiciel et à l'aide d'un langage orienté objet contemporain. Conception d'applications utilisant les outils UML ainsi que des techniques et des outils utiles au génie logiciel, tels qu'un environnement de développement intégré, la compilation automatique et les tests automatiques.

Figure 1.1: Nuage de mots importants du plan de cours de LOG210

1.1 Analyse vs Conception

Ce sujet est abordé en détail dans le chapitre 1 du livre du cours.

L'**analyse** met l'accent sur une investigation du problème et des besoins plutôt que sur la recherche d'une solution.

La **conception** sous-entend l'élaboration d'une solution conceptuelle répondant aux besoins plutôt que la mise en œuvre de cette solution.

Dans LOG210, c'est une modélisation objet qui est utilisée et pour l'analyse (classes conceptuelles décrivant le problème et les besoins) et pour la conception (classes logicielles proposant une la solution dont sa représentation est proche de la modélisation du problème).

1.2 Décalage des représentations

Plus une solution (conception) ressemble à une description du problème, plus elle est facile à comprendre. La distance entre la représentation d'un problème et la représentation de sa solution s'appelle le *décalage des représentations*. Pour des explications de Larman, lisez la section 9.3 du livre du cours.

Imaginez un jeu qui est joué dans la vraie vie avec un dé à six faces. Ensuite, on veut construire un logiciel pour ce jeu et donc on peut spécifier un besoin de générer un nombre aléatoire entre 1 et 6 (comme un dé à six faces). On peut aussi modéliser ce besoin (un élément du problème) par une classe conceptuelle `Dé` ayant un attribut `face` dont sa valeur est un type `int`. Les personnes travaillant sur un projet vont facilement comprendre ce modèle, car les gens comprennent les objets qui représentent des aspects de la vraie vie.

Ensuite, imaginez des solutions à ce problème suivantes:

1. On peut définir un programme en langage assembleur pour générer un nombre réel entre 0.00000 et 1.00000. Le programme sera assez complexe, car les métaphores en assembleur sont des registres, des adresses, peut-être des modules, etc.
2. On peut utiliser un langage orienté objet (comme Java ou TypeScript) pour définir une classe `GénérateurNombreAléatoire` qui a une fonction `générer()` qui retourne une valeur réelle aléatoire entre 0.000000 et 1.000000. Il faudra travailler un peu avec le code pour obtenir un nombre entier entre 1 et 6, mais c'est possible. Lorsqu'on lit le nom de la classe, il est possible de deviner à quelle partie du problème ça correspond, mais le lien n'est pas aussi évident.
3. On peut utiliser encore un langage orienté objet, mais cette fois on définit une classe `Dé` ayant une fonction `brasser` qui retourne une valeur aléatoire entière entre 1 et 6.

Parmi toutes les solutions au problème, laquelle a le plus faible décalage de représentation? C'est la troisième, car elle utilise la même notion de classe `Dé` qui a été utilisée pour modéliser le problème. Cet exemple est trivial, mais le principe est encore plus important lorsque le problème est complexe.

Un défi dans la programmation est d'éviter d'augmenter trop l'écart des représentations. Sinon, la solution devient moins évidente par rapport à son problème. Cette notion est aussi reliée à la facilité de la traçabilité. C'est-à-dire que chaque élément de la solution devrait être facilement traçable au problème. La méthodologie enseignée dans LOG210 cherche à réduire le décalage des représentations, car c'est un bénéfice des langages orientés objet si on fait attention.

2 Principes GRASP

GRASP est un acronyme de l'expression anglaise "General Responsibility Assignment Software Patterns" c'est-à-dire les principes pour affecter les responsabilités logicielles dans les classes.

Une approche GRASP devrait amener un design vers la modularité et la maintenabilité.

L'acronyme d'une expression vulgarisée pourrait être POMM: "Principes pour déterminer Où Mettre une Méthode".

En tant qu'ingénieur logiciel, vous devez décider souvent où placer une méthode (dans quelle classe) et cette décision ne devrait pas être prise de manière arbitraire, mais plutôt en suivant les directives d'ingénierie favorisant la modularité.

Alors, les GRASP sont les directives qui vous aident à prendre des décisions de conception, menant à un design avec moins de couplage inutile et des classes plus cohésives. Les classes cohésives sont plus faciles à comprendre, à maintenir et à réutiliser.

⚠ Avez-vous déjà une bonne expérience en programmation? Avez-vous l'habitude de coder rapidement des solutions qui fonctionnent? Si la réponse est oui, alors travailler avec les principes GRASP peut être un défi pour vous. Vous devez être en mesure de justifier vos choix de conception et cela va vous ralentir au début. Le but avec GRASP (et le cours LOG210) est d'apprendre à faire du code facile à maintenir. C'est normal au début que ça prenne plus de temps. Mais une fois que vous avez l'habitude à l'utiliser, vous serez aussi rapide avec votre développement, mais en plus votre design sera meilleur sur le plan de la maintenabilité.

2.1 Spectre de la conception

Neal Ford a proposé la notion d'effort pour la conception qu'il a nommée le "Spectre de la conception". La figure 2.1 illustre le principe.

À une extrémité il y a la notion de mettre presque zéro effort pour une conception, que l'on nomme "Hacking cowboy". C'est le cas lors d'un hackathon (un marathon de programmation durant 24 ou 48 heures où il faut produire une solution rapidement). Vous ne feriez pas un logiciel avec 10 patrons GoF ou les diagrammes UML pour réfléchir à votre architecture. Mais vous savez aussi que le code qui est produit lors d'un hackathon ne sera pas facile à maintenir. Le seul but est de faire du code qui marche.

Au fait, dans certains contextes d'entreprise (par exemple une entreprise en démarrage qui a seulement six mois de financement), c'est une une attitude similaire. Si une solution de "produit minimum viable" (MVP en anglais)W n'existe pas à la fin de la période de financement, l'entreprise n'existera plus, car il n'y aura pas une deuxième période de financement. Si la compagnie est financée pour une deuxième période, la conception du code aura besoin de beaucoup de soins, car elle a été négligée. Cette négligence à la conception est aussi nommée la "dette technique"W.

À l'autre extrémité du spectre, c'est beaucoup d'effort dépensé sur la conception, que l'on nomme "Cascade pure". Dans le cycle de vie en cascade, on met un temps fixe, par exemple plusieurs mois, à étudier

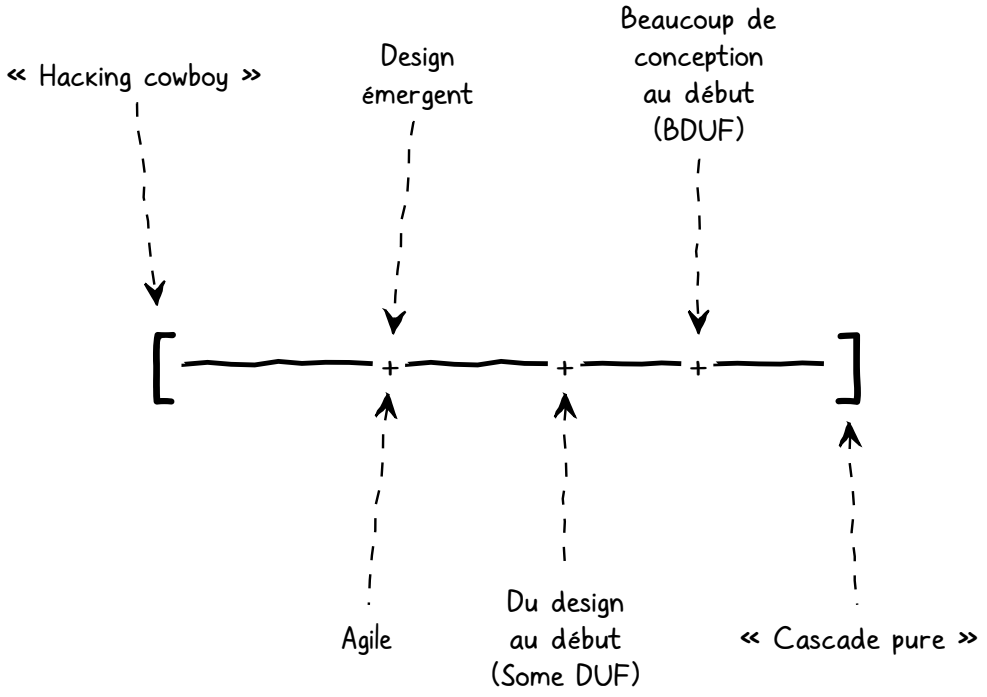

Figure 2.1: Spectre de la conception, adapté de Neal Ford

la conception. Comme toute chose poussée à l'extrême, ce n'est pas idéal non plus. Dans le livre du cours, Larman explique en détail des problèmes posés par une approche en cascade. Dans certains domaines, par exemple les logiciels pour le contrôle d'avion ou des appareils médicaux, une approche cascade est toujours utilisée, en dépit des problèmes dus à l'approche. La sécurité des logiciels est très importante, alors on passe beaucoup de temps à vérifier et valider la conception. Puisque les exigences sont plus stables, l'approche en cascade n'est pas si mal. Pourtant le coût pour produire des logiciels certifiés est énorme.

Le spectre de la conception est très important pour LOG210, parce que c'est le contexte de l'entreprise pour laquelle vous travaillez qui déterminera combien d'effort à mettre sur la conception. Si vous négligez complètement la conception, vous pouvez peut-être produire du code qui fonctionne plus vite à court terme. Mais il faudra repayer la dette technique un jour. Un moyen de gérer cette dette technique est avec le réusinage (anglais refactoring).

2.2 Tableau des principes GRASP

Voici un extrait du livre du cours, **UML 2 et les design patterns** de Craig Larman.

Table 2.1: Patterns (principes) GRASP

Pattern	Description
Expert en information	Un principe général de conception d'objets et d'affectation des responsabilités. Affecter une responsabilité à l'expert – la classe qui possède les informations nécessaires pour s'en acquitter.

Pattern	Description
Créateur	Qui crée ? (Notez que Fabrique Concrète est une solution de rechange courante.) Affectez à la classe B la responsabilité de créer une instance de la classe A si l'une des assertions suivantes est vraie: 1. B contient A 2. B agrège A 3. B a les données pour initialiser A 4. B enregistre A 5. B utilise étroitement A
Contrôleur	Quel est le premier objet en dehors de la couche présentation qui reçoit et coordonne (« contrôle ») les opérations système ? Affectez une responsabilité à la classe qui correspond à l'une de ces définitions : 1. Elle représente le système global, un « objet racine », un équipement ou un sous-système (*contrôleur de façade*). 2. Elle représente un scénario de cas d'utilisation dans lequel l'opération système se produit (*contrôleur de session* ou *contrôleur de cas d'utilisation*). On la nomme *GestionnaireX* où *X* est le nom du cas d'utilisation.
Faible Couplage (évaluation)	Comment minimiser les dépendances? Affectez les responsabilités de sorte que le couplage (inutile) demeure faible. Employez ce principe pour évaluer les alternatives.
Forte Cohésion (évaluation)	Comment conserver les objets cohésifs, compréhensibles, gérables et, en conséquence, obtenir un Faible Couplage ? Affectez les responsabilités de sorte que les classes demeurent cohésives. Employez ce principe pour évaluer les différentes solutions.
Polymorphisme	Qui est responsable quand le comportement varie selon le type ? Lorsqu'un comportement varie selon le type (classe), affectez la responsabilité de ce comportement - avec des opérations polymorphes - aux types pour lesquels le comportement varie.
Fabrication Pure	En cas de situation désespérée, que faire quand vous ne voulez pas transgresser les principes de faible couplage et de forte cohésion ? Affectez un ensemble très cohésif de responsabilités à une classe « comportementale » artificielle qui ne représente pas un concept du domaine - une entité fabriquée pour augmenter la cohésion, diminuer le couplage et faciliter la réutilisation.

Pattern	Description
Indirection	Comment affecter les responsabilités pour éviter le couplage direct ? Affectez la responsabilité à un objet qui sert d'intermédiaire avec les autres composants ou services.
Protection des variations	Comment affecter les responsabilités aux objets, sous-systèmes et systèmes de sorte que les variations ou l'instabilité de ces éléments n'aient pas d'impact négatif sur les autres ? Identifiez les points de variation ou d'instabilité prévisibles et affectez les responsabilités afin de créer une « interface » stable autour d'eux.

3 Besoins (exigences)

Un exemple d'une exigence d'un système pourrait être qu'il doit afficher le nombre d'utilisateurs dans un forum de discussion en ligne. Il s'agit d'une exigence de fonctionnalité. Si cette information doit être actualisée toutes les 2 secondes, alors il s'agit d'une exigence sur *la qualité de la performance* du système. Pour les qualités d'un système comme la performance, on peut les appeler *exigences non fonctionnelles*, car elles ne sont pas les fonctionnalités. Il y a beaucoup d'exemples d'exigences non fonctionnelles, par exemple sur Wikipedia W.

Figure 3.1: Besoins non fonctionnels?

3.1 FURPS+

Le **chapitre 5** du livre du cours traite le sujet des besoins et leur évolution. FURPS+ est un modèle (avec un acronyme) pour classer les exigences (besoins) d'un logiciel. Voici un résumé, mais la **section 5.4** du livre explique en détail:

- **Fonctionnalité** (*Functionality*). Ce sont les exigences exprimées souvent par les cas d'utilisation, par exemple, *Traiter une vente*. La sécurité est aussi considérée dans ce volet.
- **Aptitude à l'utilisation** (*Usability*). Convivialité - les facteurs humains du logiciel, par exemple le nombre de clics que ça prend pour réaliser une fonctionnalité, combien une interface est facile à comprendre par un utilisateur, etc.
- **Fiabilité** (*Reliability*). Comment le logiciel doit se comporter lorsqu'il y a des problèmes ou des pannes, par exemple, un traitement texte produit un fichier de sauvegarde de secours, ou une application continue à fonctionner même si le réseau est coupé.
- **Performance** (*Performance*) - comment un logiciel doit se comporter lors d'une charge importante sur le système, par exemple, lors de la période d'inscription, Cheminot doit avoir un temps de réponse de moins de 2 secondes.
- **Possibilités de prise en charge** (*Supportability*) Adaptabilité ou maintenabilité - combien le logiciel sera facile à modifier face aux changements prévus, par exemple, lors d'un changement de lois fiscales, quelles sont les caractéristiques de la conception qui vont faciliter le développement d'une nouvelle version du logiciel.
- + : Comprend toutes les autres choses:
  - **Implémentation**. par exemple le projet doit être réalisé avec des langages et des bibliothèques qui ne sont pas payants (logiciel libre).
  - **Interface**. par exemple contraintes d'interfaçage avec un système externe.
  - **Exploitation**. par exemple utilisation de système d'intégration continue.
  - **Aspects juridiques**. par exemple la licence du logiciel, les politiques de confidentialité, etc.