



LOG210 Analyse et conception de logiciels

Notes de cours

Christopher Fuhrman

Yvan Ross

31 juillet 2020

Table des matières

1 Analyse et conception de logiciels	1
1.1 Livre obligatoire	2
1.2 Analyse vs Conception	2
1.3 Décalage des représentations	2
1.4 Survol de la méthodologie	3
1.5 Développement itératif, évolutif et agile	4
2 Principes GRASP	7
2.1 Spectre de la conception	7
2.2 Tableau des principes GRASP	8
2.3 GRASP et RDCU	10
2.4 GRASP et Patterns GoF	10
3 Besoins (exigences)	11
3.1 FURPS+	12
4 Cas d'utilisation	13
4.1 Exemple : jeu de Risk	13
5 Modèle du domaine (MDD, modèle conceptuel)	15
5.1 Classes conceptuelles	15
5.2 Attributs	17
5.3 Associations	17
5.4 Exemple de MDD pour le jeu de Risk	19
5.5 Attributs dérivés	20
5.6 Classes d'association	20
5.7 Affinement du MDD	21
5.8 FAQ MDD	21
6 Diagrammes de séquence système (DSS)	23
6.1 Exemple : DSS pour Attaquer un pays	23
6.2 Les DSS font abstraction de la couche présentation	23
6.3 FAQ DSS	25
7 Contrats d'opération	27
7.1 Qu'est-ce qu'un contrat d'opération	27
7.2 Exemple : Contrats d'opération pour Attaquer un pays	28
8 Réalisations de cas d'utilisation (RDCU)	31
8.1 Spécifier le contrôleur	33
8.2 Satisfaire les postconditions	33

Table des matières

8.3	Visibilité	34
8.4	Transformer identifiants en objets	35
8.5	Utilisation de tableau associatif (Map<clé, objet>)	35
8.6	RDCU pour l'initialisation, le scénario Démarrer	36
9	Développement piloté par les tests	39
9.1	Kata TDD	41
10	Réusinage (Refactorisation)	45
10.1	Introduction	45
10.2	Symptômes de la mauvaise conception - Code smells	46
10.3	Automatisation du réusinage par les IDE	48
10.4	Impropriété	48
11	Développement de logiciel en équipe	49
11.1	Humilité, Respect, Confiance	50
11.2	Redondance des compétences dans l'équipe (Bus Factor)	51
11.3	Mentorat	52
11.4	Scénarios	52
12	Outils pour la modélisation UML	55
12.1	Exemples de diagramme avec PlantUML pour LOG210	57
12.2	Astuces PlantUML	57
13	Décortiquer les patterns GoF avec GRASP	61
13.1	Exemple avec Adaptateur	61
13.2	Imaginer le code sans le pattern GoF	61
13.3	Identifier les GRASP dans les GoF	63
13.4	GRASP et réusinage	64
14	Fiabilité	65
15	Diagrammes d'état	67
16	Diagrammes d'activités	69
17	Conception de packages	71
18	Dette technique	73
19	Diagrammes de déploiement et de composants	77
19.1	Diagrammes de déploiement	77
20	Laboratoires	81
20.1	JavaScript/TypeScript	81
20.2	Contributions de l'équipe	81
20.3	TODO	84
21	Bibliographie	85

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.

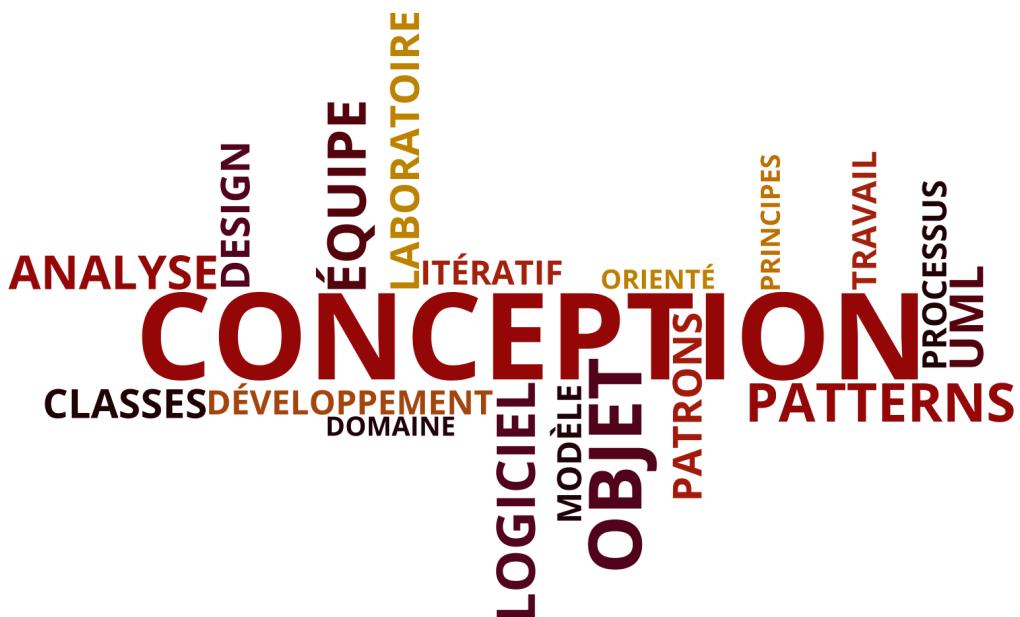


FIGURE 1.1 – Nuage de mots importants du plan de cours de LOG210.

1.1 Livre obligatoire

Le livre obligatoire (Larman, 2005) pour ce cours est indiqué dans le plan de cours. **Le présent document n'est pas un substitut pour le livre obligatoire.**

Les références au livre obligatoire seront indiquées par l'icône du livre .

1.2 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).

1.3 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.

1.4 Survol de la méthodologie

La figure 1.2 présente la méthode d'analyse et de conception enseignée dans le cours. C'est une adaptation de plusieurs figures présentées dans le livre du cours.

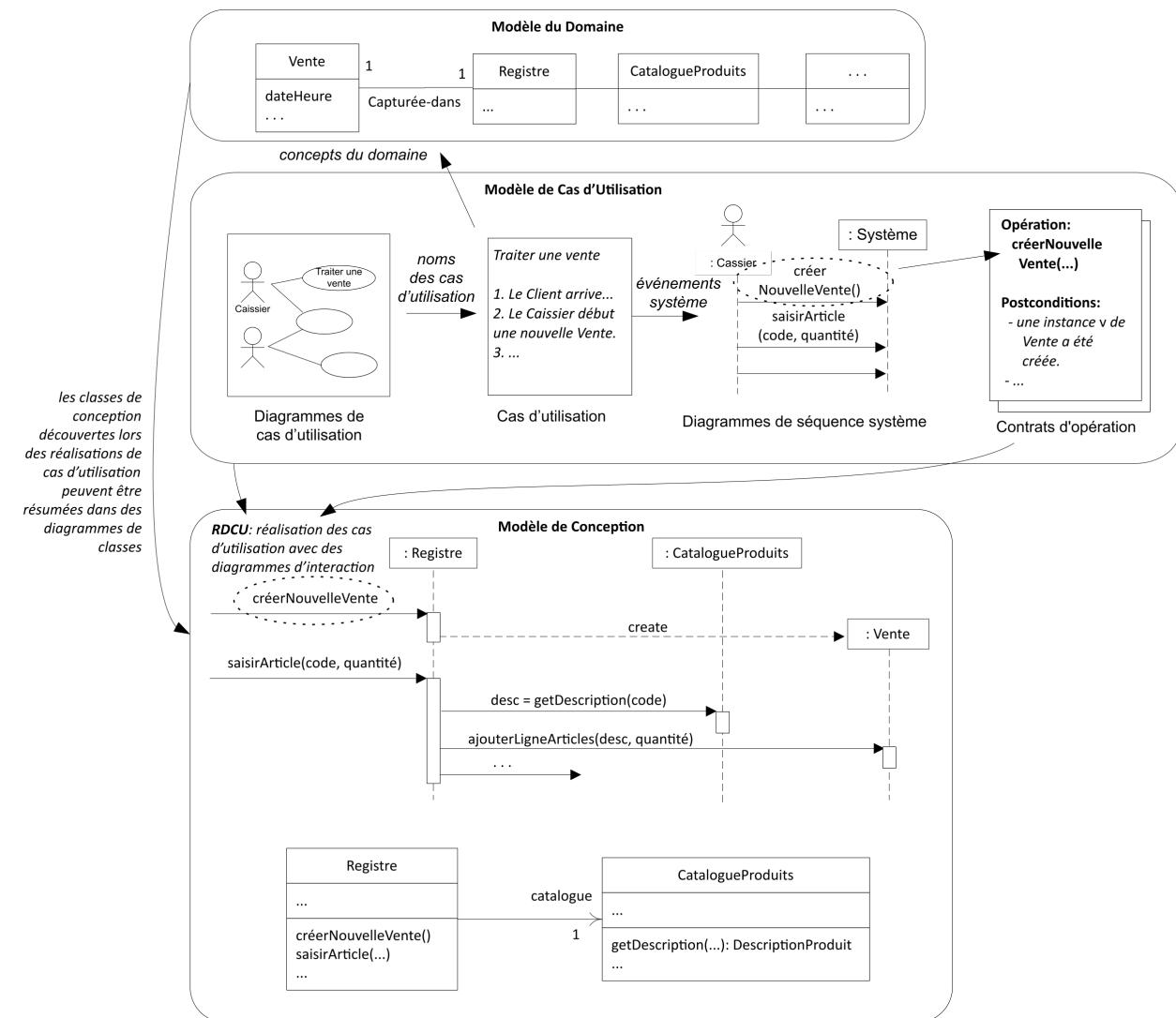


FIGURE 1.2 – Survol de la méthodologie.

1.5 Développement itératif, évolutif et agile

Le chapitre 2 du livre **UML** définit un processus itératif et adaptatif ainsi que les concepts fondamentaux du Processus Unifié.

Les points importants sont les suivants :

- Le développement itératif et évolutif implique de programmer et de tester précocement un système partiel selon des cycles répétitifs.
- Un cycle est nommé une itération et dure un temps fixe (par exemple, 3 semaines) comprenant les activités d'analyse, de conception, de programmation et de test, ainsi qu'une démonstration pour solliciter du feedback du client (voir la figure 1.3).
- La durée d'une itération est limitée dans le temps (*timeboxed* en anglais), de 2 à 6 semaines. Il n'est pas permis d'ajouter du temps à la durée d'une itération si le projet avance plus lentement que prévu, car cela impliquerait un retard de la rétroaction du client. Si le respect des délais semble compromis, on supprime plutôt des tâches ou des spécifications et on les inclut dans l'itération suivante.
- Les premières itérations peuvent sembler chaotiques, car elles sont loin de la « bonne voie ». Avec la rétroaction du client et l'adaptation, le système à développer converge vers une solution appropriée (voir la figure 1.4).
- Il y a plusieurs avantages du développement itératif et incrémental :
 - moins d'échecs, amélioration de la productivité et de la qualité;
 - gestion précoce des risques élevés (risques techniques, exigences, objectifs, convivialité, etc.)
 - progrès immédiatement visibles;
 - rétroaction, implication des utilisateurs et adaptation précoces;
 - complexité gérée (par itération);
 - possibilité d'exploiter méthodiquement les leçons tirées d'une itération.

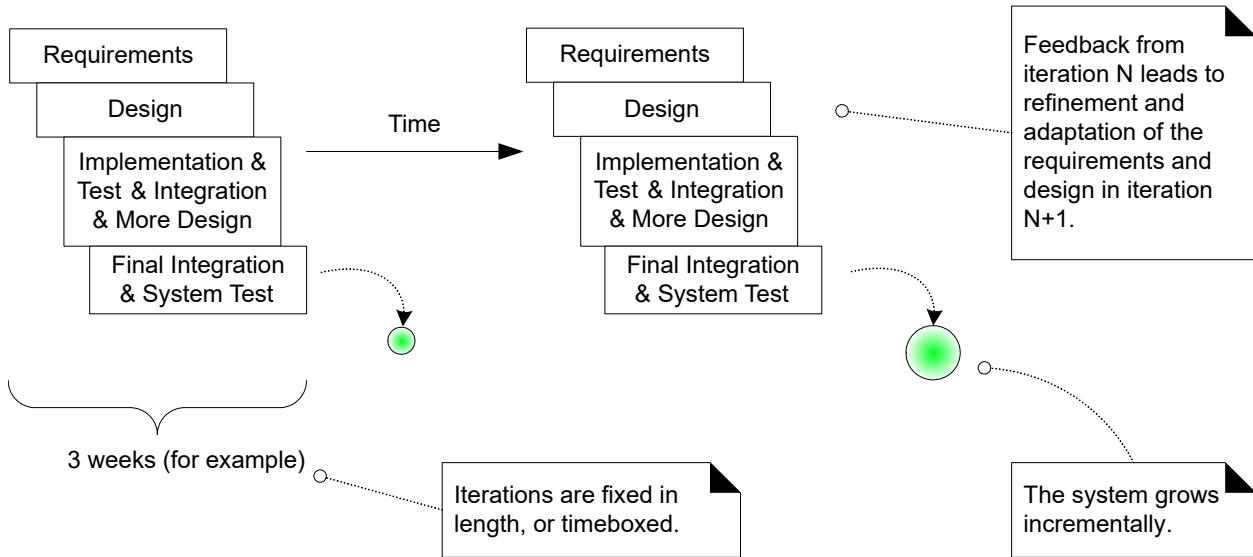


FIGURE 1.3 – Le développement itératif et incrémental (Figure 2.1 du livre).

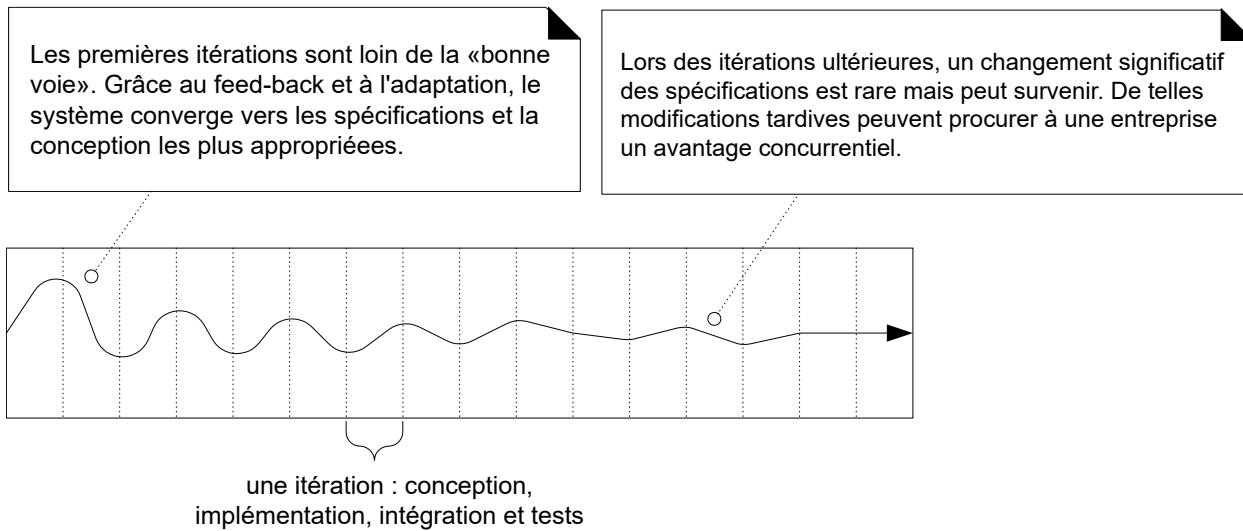


FIGURE 1.4 – Rétroaction et adaptation itératives convergent vers le système souhaité (Figure 2.2 du livre).

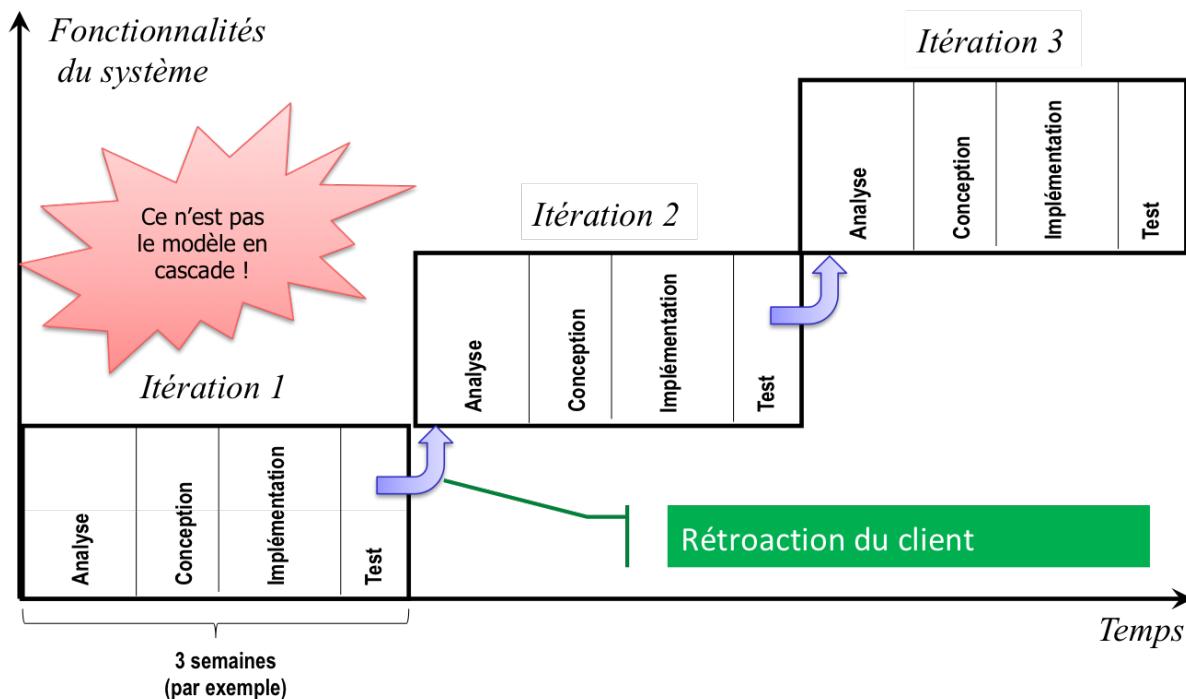


FIGURE 1.5 – Processus itératif et évolutif.

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 ([2009](#)) 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 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,

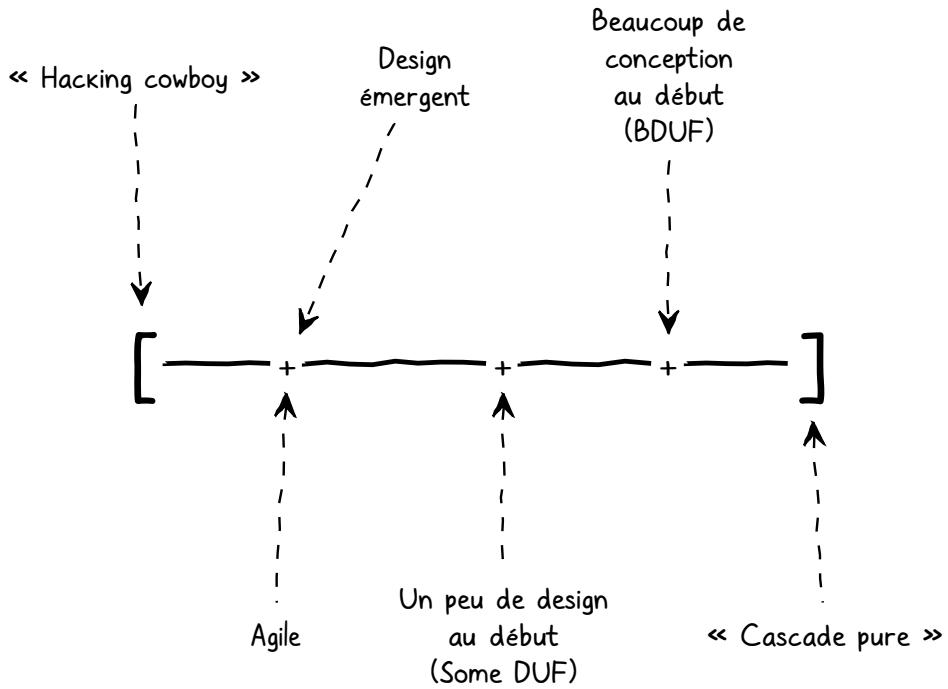


FIGURE 2.1 – Spectre de la conception, adapté de Neal Ford. ([PlantUML](#))

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](#).

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 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 en 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 (et les développeurs ont *a priori* une meilleure compréhension du domaine), 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 de *récusiner* (anglais refactor).

2.2 Tableau des principes GRASP

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

Tableau 2.1 – Patterns (principes) GRASP

Pattern	Description
Expert en information <i>F16.11/A17.11</i>	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.
Créateur <i>F16.10/A17.10</i>	Qui crée ? (Notez que Fabrique Concète 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 : <ol style="list-style-type: none"> 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 <i>F16.13/A17.13</i>	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 : <ol style="list-style-type: none"> 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 (<i>contrôleur de session</i> ou contrôleur de cas d'utilisation). On la nomme GestionnaireX, où X est le nom du cas d'utilisation
Faible Couplage (évaluation) <i>F16.12/A17.12</i>	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) <i>F16.14/A17.14</i>	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 <i>F22.1/A25.1</i>	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.

2 Principes GRASP

Pattern	Description
Fabrication Pure <i>F22.2/A25.2</i>	<p>En cas de situation désespérée, que faire quand vous ne voulez pas transgresser les principes de faible couplage et de forte cohésion ?</p> <p>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.</p>
Indirection <i>F22.3/A25.3</i>	<p>Comment affecter les responsabilités pour éviter le couplage direct ?</p> <p>Affectez la responsabilité à un objet qui sert d'intermédiaire avec les autres composants ou services.</p>
Protection des variations <i>F22.4/A25.4</i>	<p>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 ?</p> <p>Identifiez les points de variation ou d'instabilité prévisibles et affectez les responsabilités afin de créer une « interface » stable autour d'eux.</p>

2.3 GRASP et RDCU

Les principes GRASP sont utilisés dans les réalisations de cas d'utilisation (RDCU). On s'en sert pour annoter des décisions de conception, pour rendre explicite (documenter) les choix. Voir la section [Réalisations de cas d'utilisation \(RDCU\)](#) pour plus d'informations.

2.4 GRASP et Patterns GoF

On peut voir les principes GRASP comme des généralisations (principes de base) des patterns GoF. Voir la section [Décortiquer les patterns GoF avec GRASP](#) pour plus d'informations.

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](#).

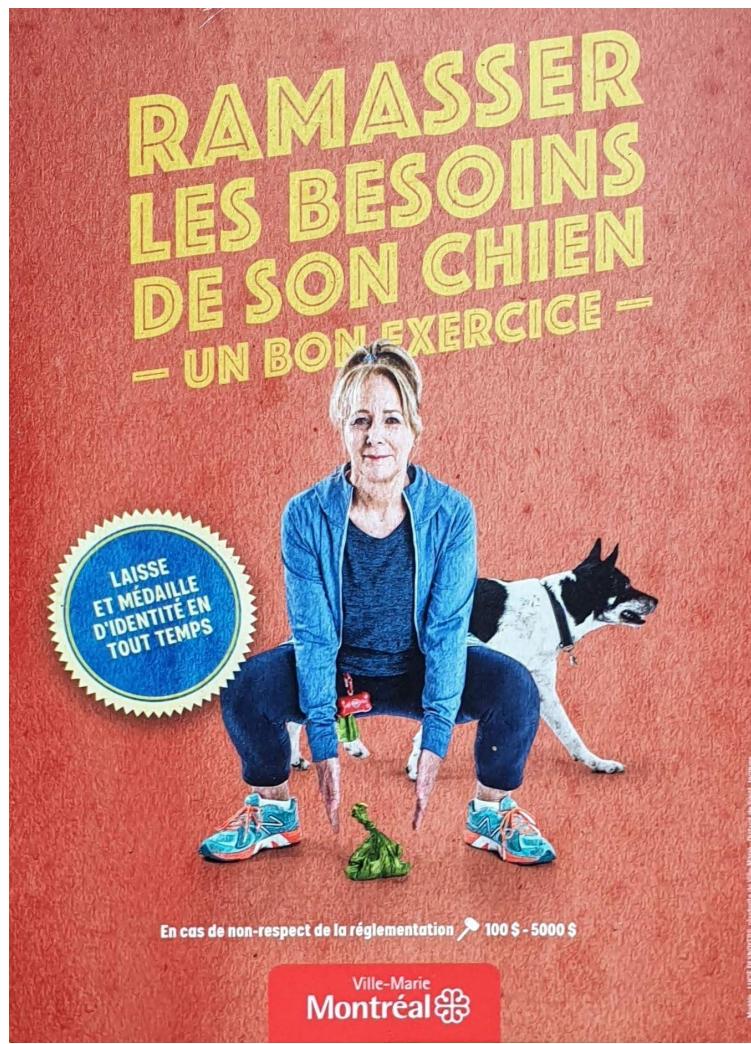


FIGURE 3.1 – Besoins non fonctionnels ?

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, à quel point 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 - à quel point 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.

4 Cas d'utilisation

Le chapitre 6 **█** du livre du cours présente les cas d'utilisation. Il s'agit des documents textuels décrivant l'interaction entre un système (logiciel à développer) et un ou plusieurs acteurs (les utilisateurs ou systèmes externes). Le cas d'utilisation décrit plusieurs scénarios, mais en général il y a un scénario principal (« *Happy Path* ») représentant ce qui se passe lorsqu'il n'y a pas d'anomalie.

La notation UML inclut les diagrammes de cas d'utilisation, qui sont comme une table des matières pour les fonctionnalités d'un système.

Dans LOG210, la théorie sur comment écrire les cas d'utilisation ne fait pas partie du cours ; c'est un élément expliqué dans un autre cours (*LOG410 Analyse de besoins et spécifications*).

4.1 Exemple : jeu de Risk

L'exemple suivant concerne le jeu de Risk.



FIGURE 4.1 – Cinq dés utilisés dans le jeu de Risk¹.

Selon « Risk ». 2019. Wikipédia. <https://fr.wikipedia.org/w/index.php?title=Risk&oldid=164264587> (9 décembre 2019) :

1. By Val42 - <https://en.wikipedia.org/wiki/Image:Risk-dice-example.jpg>, CC By-SA 3.0 Link

L'attaquant jette un à trois dés suivant le nombre de régiments qu'il désire engager (avec un maximum de trois régiments engagés, et en considérant qu'un régiment au moins doit rester libre d'engagements sur le territoire attaquant) et le défenseur deux dés (un s'il n'a plus qu'un régiment). On compare le dé le plus fort de l'attaquant au dé le plus fort du défenseur et le deuxième dé le plus fort de l'attaquant au deuxième dé du défenseur. Chaque fois que le dé du défenseur est supérieur ou égal à celui de l'attaquant, l'attaquant perd un régiment ; dans le cas contraire, c'est le défenseur qui en perd un.

4.1.1 Scénario : Attaquer un pays

1. Le Joueur attaquant choisit d'attaquer un pays voisin du Joueur défenseur.
2. Le Joueur attaquant annonce combien de régiments il va utiliser pour son attaque.
3. Le Joueur défenseur annonce combien de régiments il va utiliser pour sa défense.
4. Les deux Joueurs jettent le nombre de dés selon leur stratégie choisie aux étapes précédentes.
5. Le Système compare les dés et élimine les régiments de l'attaquant ou du défenseur selon les règles et affiche le résultat.

Les Joueurs répètent l'étape 2 jusqu'à ce que l'attaquant ne puisse plus attaquer ou ne veuille plus attaquer.

4.1.2 Diagramme de cas d'utilisation

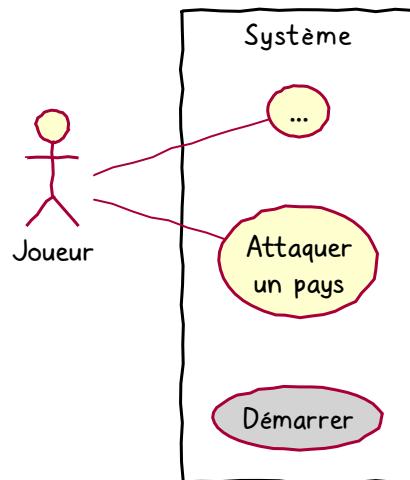


FIGURE 4.2 – Diagramme de cas d'utilisation. ([PlantUML](#))

Notes :

- Le cas d'utilisation « ... » signifie tous les autres cas d'utilisation du jeu, par exemple pour distribuer les régiments à chaque tour, etc.
- Le cas d'utilisation *Démarrer* n'est pas normalement indiqué dans un diagramme. C'est une astuce pédagogique pour la méthodologie du cours, car il faudra concevoir et coder ce scénario, bien qu'il ne soit pas une fonctionnalité connue par l'utilisateur.

5 Modèle du domaine (MDD, modèle conceptuel)

Les MDD sont expliqués en détail dans le chapitre 9  du livre du cours, mais voici des points importants pour LOG210 :

- Les classes conceptuelles ne sont pas des classes logicielles. Ainsi, selon la méthodologie de Larman, *elles n'ont pas de méthodes*.
- Les classes ont des noms commençant avec une lettre majuscule, par exemple `Joueur` et elles ne sont jamais au pluriel, par exemple `Joueurs`.

5.1 Classes conceptuelles

Il y a trois stratégies pour identifier les classes conceptuelles :

1. Réutiliser ou modifier des modèles existants.
2. Utiliser une liste de catégories.
3. Identifier des groupes nominaux.

5.1.1 Catégories pour identifier des classes conceptuelles

Tableau 5.1 – Extrait du tableau 9.1  du livre du cours.

Catégorie	Exemples
Transactions métier	<i>Vente, Attaque, Réservation, Inscription, EmpruntVélo</i>
Elles sont essentielles, commencez l'analyse par les transactions.	
Lignes d'une transaction	<i>LigneArticles, ExemplaireLivre, GroupeCours</i>
Éléments compris dans une transaction.	
Produit ou service lié à une transaction ou une ligne de transaction	<i>Article, Vélo, Vol, Livre, Cours</i>
Pour quel concept sont faites des transactions ?	
Où la transaction est-elle enregistré ?	<i>Caisse, GrandLivre, ManifesteDeVol</i>

5 Modèle du domaine (MDD, modèle conceptuel)

Catégorie	Exemples
Rôle des personnes liées à la transaction	<i>Caissier, Client, JoueurDeMonopoly, Passager</i>
Qui sont les parties impliquées dans une transaction ?	
Organisations liées à la transaction	<i>Magasin, CompagnieAérienne, Bibliothèque, Université</i>
Quelles sont les organisations impliquées dans une transaction ?	
Lieu de la transaction ; lieu du service	<i>Magasin, Aéroport, Avion, Siège, LocalCours</i>
Événements notables, à mémoriser	<i>Vente, Paiement, JeuDeMonopoly, Vol</i>
Objets physiques	<i>Article, Caisse, Plateau, Pion, Dé, Vélo</i>
Important surtout lorsqu'il s'agit d'un logiciel de contrôle d'équipements ou de simulation.	
Description d'entités	<i>DescriptionProduit, DescriptionVol, Livre</i> (en opposition avec <i>Exemplaire</i>), <i>Cours</i> (en opposition avec <i>CoursGroupe</i>)
Voir section 9.13 pour plus d'informations.	
Catalogues	<i>CatalogueProduits, CatalogueVols, CatalogueLivres, CatalogueCours</i>
Les descriptions se trouvent souvent dans des catalogues	
Conteneurs	<i>Magasin, Rayonnage, Plateau, Avion, Bibliothèque</i>
Un conteneur peut contenir des objets physiques ou des informations.	
Contenu d'un conteneur	<i>Article, Case (sur un Plateau de jeu), Passager, Exemplaire</i>
Autres systèmes externes	<i>SystèmeAutorisationPaiementsACrédit, SystèmeGestionBorderaux</i>
Documents financiers, contrats, documents légaux	<i>Reçus, GrandLivre, JournalDeMaintenance</i>

Catégorie	Exemples
Instruments financiers	<i>Espèces, Chèque, LigneDeCrédit</i>
Plannings, manuels, documents régulièrement consultés pour effectuer un travail	<i>MiseAJourTarifs, PlanningRéparations</i>

5.2 Attributs

Les attributs sont le sujet de la section 9.16 du livre. Comme c'est le cas pour les classes et les associations, on fait figurer les attributs *quand les cas d'utilisation suggèrent la nécessité de mémoriser des informations*.

Pour l'UML, la syntaxe complète d'un attribut est :

visibilité nom : type multiplicité = défaut {propriété}

Voici des points importants :

- *Le type d'un attribut est important et il faut les spécifier dans un MDD*, même si dans le livre du cours il y a plusieurs exemples sans type.
- On ne se soucie pas de la visibilité des attributs dans un MDD.
- Faites attention à l'attribut qui devrait être une classe. Si on ne pense pas un attribut *X* en termes alphanumériques dans le monde réel, alors il s'agit probablement d'une classe conceptuelle. Voir la section 9.12 du livre.
- De la même manière, faites attention aux informations qui sont mieux modélisées par des associations, par exemple dans la figure 5.1 la classe *Pays* n'a pas un *attribut joueur:Joueur* (qui contrôle le *Pays*), mais elle a plutôt une *association* avec la classe *Joueur* et un verbe *contrôle*.

⚠ Il est vrai que dans un langage de programmation, les associations doivent être les attributs dans les classes. Cependant, dans un modèle du domaine on cherche à éviter des attributs si une association peut mieux décrire la relation. La relation relie visuellement les deux classes et elle est décrite avec un verbe.

5.3 Associations

Les associations dans le MDD sont le sujet de la section 9.14 du livre du cours. Il faut se référer au contenu du livre pour les détails. Une association est une relation entre des classes (ou des instances de classes). Elle indique une connexion significative ou intéressante. Voici des points importants :

- Il est facile de trouver beaucoup d'associations, mais il faut se limiter à celles qui doivent être conservées un certain temps. Pensez à la **mémorabilité** d'une association dans le contexte du logiciel à développer. Par exemple, considérez les associations de la figure 5.1 :

5 Modèle du domaine (MDD, modèle conceptuel)

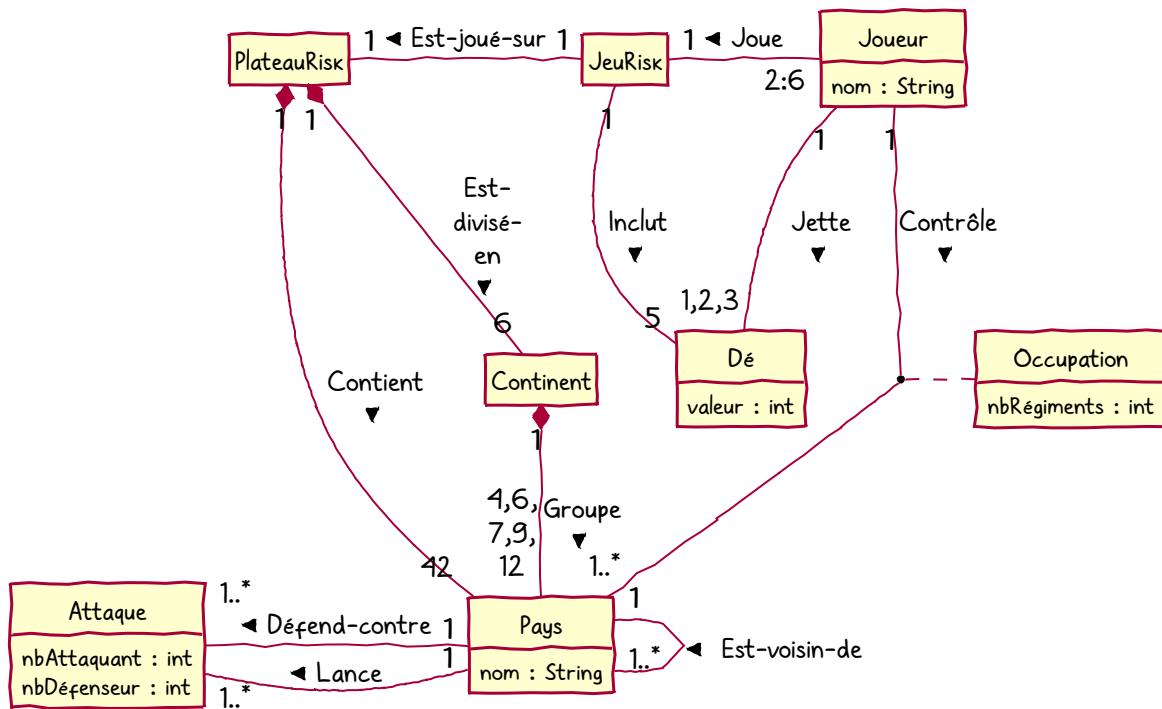
- Il existe une association entre **Joueur** et **Pays**, car il est important de savoir quel joueur contrôle quel pays dans le jeu de Risk.
- Il n'y a pas d'association entre **JeuRisk** et **Attaque**, même si les attaques font partie du jeu. Il n'est pas essentiel de mémoriser l'historique de toutes les attaques réalisées dans le jeu.
- Il y a des associations dérivées de la liste des associations courantes. Voir le tableau 5.2.
- En UML les associations sont représentées par des lignes entre classes.
 - Elles sont nommées (avec un verbe commençant par une lettre majuscule).
 - Des noms simples comme « A », « Utilise », « Possède », « Contient », etc. sont généralement des choix médiocres, car ils n'aident pas notre compréhension du domaine. Essayez de trouver des noms plus riches, si possible.
 - Une flèche (triangle) de « sens de lecture » optionnelle indique la direction dans laquelle lire l'association. Si la flèche est absente, on lit l'association de gauche à droite ou de haut en bas.
 - Les extrémités des associations ont une expression de la multiplicité indiquant une relation numérique entre les instances des classes. Vous pouvez en trouver plusieurs exemples dans la figure 5.1.

Tableau 5.2 – Extrait du tableau 9.2  du livre du cours.

Catégorie	Exemple
A est une transaction liée à une transaction B	<i>PaiementEnEspèces – Vente</i> <i>Réservation – Annulation</i>
A est un élément d'une transaction B	<i>LigneArticles – Vente</i>
A est un produit pour une transaction (ou un élément de transaction) B	<i>Article – LigneArticles (ou Vente)</i> <i>Vol – Réservation</i>
A est un rôle lié à une transaction B	<i>Client – Paiement</i> <i>Passager – Billet</i>
A est une partie logique ou physique de B	<i>Tiroir – Registre</i> <i>Case – Plateau</i> <i>Siège – Avion</i>
A est physiquement ou logiquement contenu dans B	<i>Registre – Magasin</i> <i>Joueur – Monopoly</i> <i>Passager – Avion</i>
A est une description de B	<i>DescriptionProduit – Article</i> <i>DescriptionVol – Vol</i>
A est connu/consigné/enregistré/saisi dans B	<i>Vente – Registre</i> <i>Pion – Case</i> <i>Réservation – ManifesteDeVol</i>

Catégorie	Exemple
A est un membre de B	<i>Caissier – Magasin</i> <i>Joueur – Monopoly</i> <i>Pilote – CompagnieAérienne</i>
A est une sous-unité organisationnelle de B	<i>Rayon – Magasin</i> <i>Maintenance – CompagnieAérienne</i>
A utilise, gère ou possède B	<i>Caissier – Registre</i> <i>Joueur – Pion Pilote – Avion</i>
A est voisin de B	<i>Article – Article</i> <i>Case – Case</i> <i>Ville – Ville</i>

5.4 Exemple de MDD pour le jeu de Risk

FIGURE 5.1 – Modèle du domaine du jeu de Risk. ([PlantUML](#))

5.5 Attributs dérivés

Les attributs dérivés sont expliqués en détail dans la section 9.16 [9.16](#) du livre du cours. Il s'agit des attributs qui sont calculés à partir d'autres informations reliées à la classe. Ils sont indiqués par le symbole / devant leur nom. L'exemple à la figure 5.2 s'applique à la règle du jeu de Risk spécifiant qu'un joueur reçoit un certain nombre de renforts selon le nombre de pays occupés. La classe Joueur pourrait avoir un attribut dérivé /nbPaysOccupés qui est calculé selon le nombre de Pays contrôlés par le joueur.

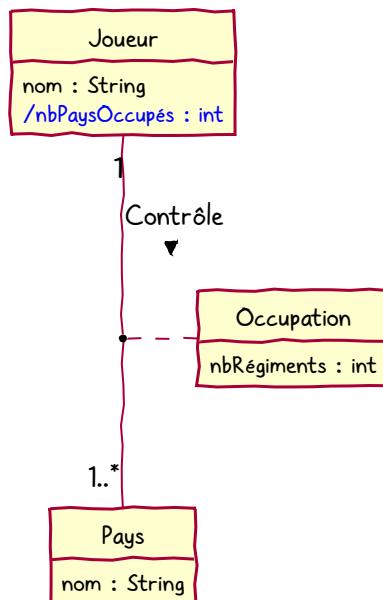


FIGURE 5.2 – Classe d’association dans le MDD Jeu de Risk. ([PlantUML](#))

5.6 Classes d’association

Les classes d’association dans le MDD sont le sujet de la section A32.10/F26.10 [9.16](#) du livre du cours.

Une classe d’association permet de traiter une association comme une classe, et de la modéliser avec des attributs...

Il pourrait être utile d’avoir une classe d’association dans un MDD si :

- un attribut est lié à une association ;
- la durée de vie des instances de la classe d’association dépend de l’association ;
- il y a une association $N-N$ entre deux concepts et des informations liées à l’association elle-même.

Dans l’exemple à la figure 5.3, voici pourquoi il y a une classe d’association Occupation. Lorsqu’un Joueur contrôle un Pays, il doit y avoir des armées dans ce dernier. Le MDD pourrait avoir un attribut nbRégiments dans la classe Pays. Cependant, l’attribut nbRégiments est lié à l’association entre le Joueur et le Pays qu’il contrôle, alors on décide d’utiliser une classe d’association.

Si un Joueur envahit un Pays, la nouvelle instance de la classe d'association Occupation sera créée (avec la nouvelle association). Pourtant, cette instance d'Occupation sera détruite si un autre Joueur arrive à prendre le contrôle du Pays. Alors, la durée de vie de cette instance dépend de l'association.

Voir le livre obligatoire pour plus d'exemples.

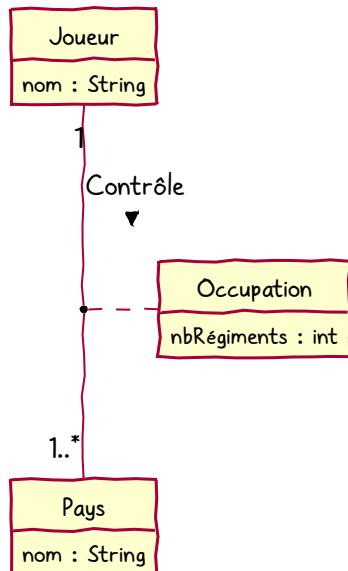


FIGURE 5.3 – Classe d'association dans le MDD Jeu de Risk. ([PlantUML](#))

5.7 Affinement du MDD

Lorsqu'on modélise un domaine, il est normal de commencer avec un modèle simple (à partir d'un ou deux cas d'utilisation) et ensuite on l'affine dans les itérations suivantes, où on y intègre d'autres éléments plus subtils ou complexes du problème qu'on étudie. Les détails de cette approche sont présentés dans le chapitre F26/A32 du livre du cours. Bien que la matière soit présentée plus tard dans le livre, ce sont des choses à savoir pour la modélisation d'un domaine, même dans une première itération.

Voici un résumé des points importants traités dans ce chapitre, dont quelques-uns ont déjà été présentés plus haut :

- Composition/Agrégation
- Généralisation/spécialisation
- Attribut dérivé
- Hiérarchies dans un MDD et héritage dans l'implémentation
- Noms de rôles
- Organisation des classes conceptuelles en Packages

5.8 FAQ MDD

Question : Est-ce qu'il y a un MDD pour chaque cas d'utilisation ?

5 Modèle du domaine (MDD, modèle conceptuel)

Réponse : Selon la méthodologie du livre du cours, il n'y a qu'un seul MDD pour un domaine d'application. Une application peut avoir beaucoup de fonctionnalités (cas d'utilisation), mais il y a un seul MDD.

Cela dit, la notion d'une *version* de MDD existe (par exemple, une version pour chaque itération). Le MDD évoluera normalement après chaque nouvelle analyse. Le MDD au début d'un projet est plus simple, puisqu'il porte sur seulement les cas d'utilisation ciblés à la première itération. Le MDD devient plus riche au fur et à mesure qu'on avance dans les itérations, parce qu'il a de plus en plus de fonctionnalités réalisées.

6 Diagrammes de séquence système (DSS)

Un diagramme de séquence système (DSS) est un diagramme UML (diagramme de séquence) limité à un acteur (provenant du scénario d'un cas d'utilisation) et le Système. Les DSS sont expliqués en détails dans le chapitre 10 [10](#) du livre du cours, mais voici des points importants pour LOG210 :

- Le DSS a toujours un titre.
- L'acteur est indiqué dans la notation par un bonhomme et est représenté comme une *instance* de la classe du bonhomme, comme `Joueur` : dans la figure [6.1](#) (le « `:` » signifie une instance).
- Le Système est un objet (une instance `Syst` : ème) et n'est jamais détaillé plus.
- Le but du DSS est de définir des opérations système (Application Programming Interface) du système ; il s'agit d'une conception de haut niveau.
- Le côté acteur du DSS n'est pas un acteur tout seul, mais une couche logicielle de présentation, comme une interface graphique ou un logiciel qui peut reconnaître la parole. Cette couche reconnaît des gestes de l'acteur (par exemple un clic sur un bouton dans l'interface, une demande « Hé Siri », etc.) et envoie une opération système.
- Puisque la couche présentation reçoit des informations des êtres humains, *les opérations système ont des arguments de type primitif*. Il est difficile pour un utilisateur de spécifier une référence (pointeur en mémoire) à un objet. Alors, on peut donner le nom (de type `String`) d'un morceau de musique à jouer, ou spécifier une quantité (de type `Integer`).
- Puisque les types des arguments sont importants, on les spécifie dans les opérations système du DSS.
- Un message de retour (ligne pointillée avec flèche ouverte) vers l'acteur représente la communication des informations précises, par exemple les valeurs des dés dans l'attaque. Puisque la couche présentation a beaucoup de moyens pour afficher ces informations, *on ne va pas spécifier les messages de retour comme des méthodes*.

6.1 Exemple : DSS pour Attaquer un pays

La figure [6.1](#) est un exemple de DSS pour le cas d'utilisation *Attaquer un pays*. Vous pouvez noter tous les détails (titre, arguments, types).

6.2 Les DSS font abstraction de la couche présentation

Le but du DSS est de se concentrer sur l'API (les opérations système) de la solution. Dans ce sens, c'est une conception de haut niveau. Le « Système » est modélisé comme une boîte noire. Par exemple, dans la figure [6.2](#) il y a l'acteur, le Système et une opération système. On ne rentre pas dans les détails, bien qu'ils existent et sont importants.

Plus tard, lorsque c'est le moment d'implémenter le code, les détails importants seront à respecter. Il faut faire attention aux principes de la séparation des couches présentation et domaine. Par exemple, la figure [6.3](#)

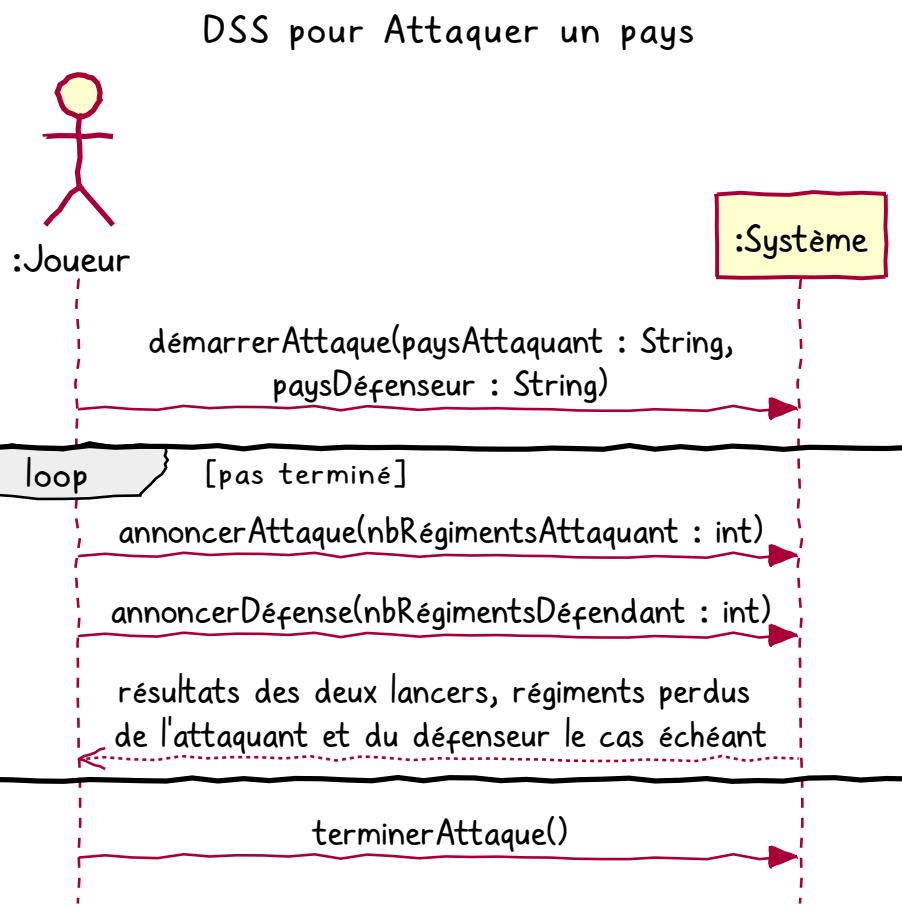


FIGURE 6.1 – Diagramme de séquence système pour *Attaquer un pays*. ([PlantUML](#))

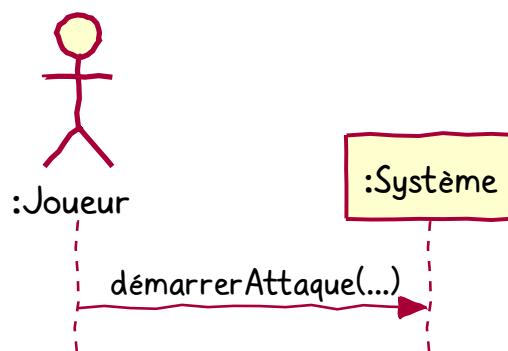


FIGURE 6.2 – Une opération système dans un DSS. C'est une abstraction. ([PlantUML](#))

rentre dans les détails de ce qui se passe réellement dans une opération système quand la solution fonctionne avec un service web :

- D'abord, l'acteur clique sur un bouton ;
- Ce clic se transforme en service REST ;
- Un routeur transforme l'appel REST en une opération système envoyée à un contrôleur GRASP.
Notez que c'est un **objet du domaine qui reçoit l'opération système** – c'est l'essence du principe GRASP Contrôleur ;
- Le contrôleur GRASP dirige la suite, selon la solution proposée dans la réalisation de cas d'utilisation (RDCU).

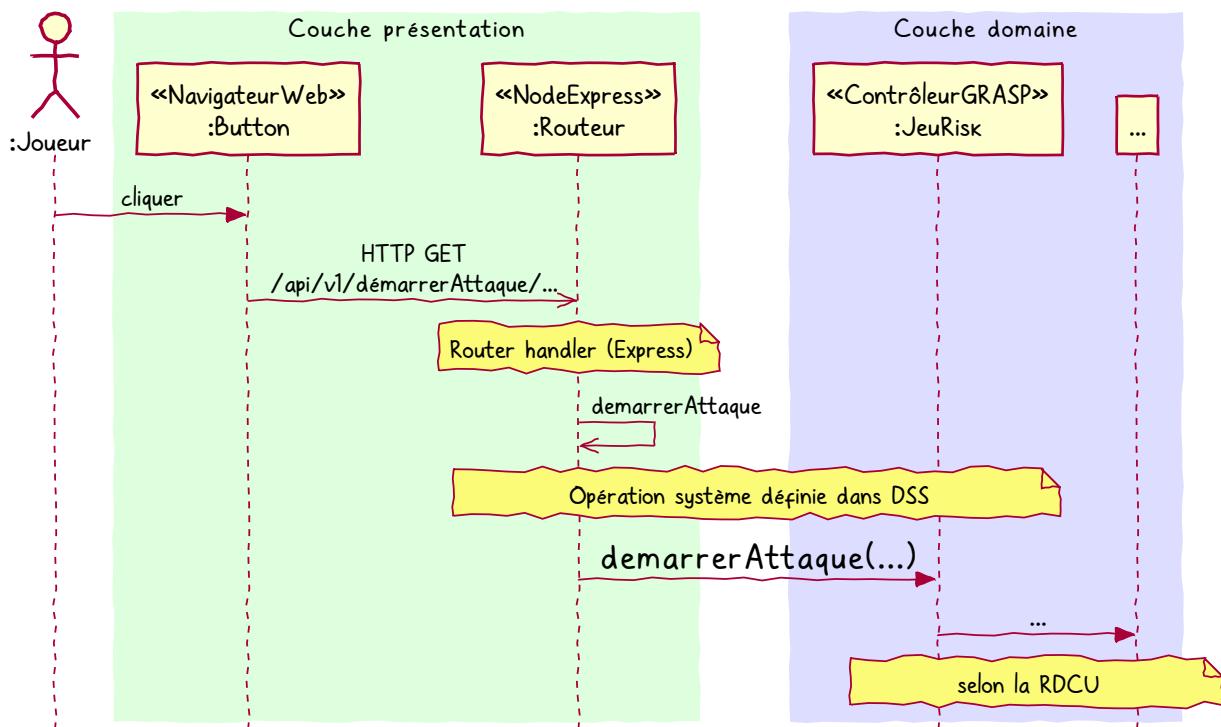


FIGURE 6.3 – Une opération système est envoyée par la couche présentation et elle est reçue dans la couche domaine par son contrôleur GRASP. Ceci est un exemple avec un navigateur web, mais d'autres possibilités existent pour la couche présentation. ([PlantUML](#))

⚠ La figure 6.3 est à titre d'information seulement. Les DSS sont censés être simples (sans rentrer dans les détails). C'est de la conception à haut niveau.

6.3 FAQ DSS

Question : Comment faire si un cas d'utilisation a des *scénarios alternatifs*? Fait-on plusieurs DSS (un pour chaque scénario) ou utilise-t-on la notation UML (des blocs opt et alt) pour montrer des flots différents dans le même DSS?

6 Diagrammes de séquence système (DSS)

Réponse : Un objectif de faire un DSS est de **définir les opérations système**. Donc, on peut se poser la question suivante : les scénarios alternatifs impliquent-ils une ou plusieurs opérations système n'ayant pas encore été définies ? Si la réponse est non, on peut ignorer les scénarios alternatifs dans le DSS. Par contre, si la réponse est oui, il est essentiel de définir ces opérations système dans un DSS. Quant au choix de faire des DSS séparés ou d'utiliser la notation UML pour montrer les flots différents sur le même DSS, ça dépend de la complexité de la logique des flots. Un DSS devrait être *facile à comprendre*. C'est à vous de juger si votre DSS avec des `opt` ou `alt` est assez simple ou fait du spaghetti. Utilisez un autre DSS (ou plusieurs) ayant le nom des scénarios alternatifs si cela vous semble plus clair.

Question : Puisqu'une opération système doit avoir seulement des arguments de type primitif, j'ai plusieurs opérations système avec de nombreux (plus que 5) arguments. Pourquoi il n'est pas permis de passer des objets comme argument ?

Réponse : Il n'est pas conseillé de passer des *objets du domaine* comme argument, puisque c'est la couche présentation qui invoque l'opération système. Si cette couche manipule les objets du domaine, cela ne respecte pas la séparation des couches. Une solution est d'appliquer un réusinage ([Réusinage \(Refactorisation\)](#)) pour le *smell* nommé *Long Parameter List*, par exemple [Introduce Parameter Object](#). Notez que l'objet que vous introduisez n'est pas un objet (classe) du domaine ! La distinction est importante, car la logique d'affaires est toujours en dehors de la couche de présentation.

Question : Décortiquer toutes les informations dans un formulaire web est compliqué, puis on doit passer tout ça à un contrôleur GRASP comme des arguments de type primitif. Ne serait-il pas plus simple de passer l'objet `body` de la page web au contrôleur GRASP et le laisser faire le décorticage ?

Réponse : Dans un sens ça serait plus simple (pour le code de la couche de présentation). Cependant, le but de séparer les couches est de favoriser l'utilisation d'autres couches présentation, par exemple à travers une application iOS ou Android. Si vous mettez la logique de la couche présentation (décortiquer un formulaire web) dans la couche domaine (le contrôleur GRASP), ça ne respecte pas les responsabilités des couches. Imaginez un tel contrôleur GRASP si vous avez 3 types d'application frontale (navigateur web, application iOS et application Android). Le contrôleur GRASP recevra des représentations de « formulaire » de chaque couche présentation différente (`body` n'a rien à voir avec une interface Android). Ce pauvre contrôleur serait obligé de connaître alors toutes les trois formes (web, iOS, Android) et ainsi sa cohésion sera beaucoup plus faible. Laisser la couche présentation faire le décorticage et construire une opération système selon l'API définie dans le DSS simplifie le contrôleur GRASP et respecte les responsabilités des couches.

7 Contrats d'opération

Les contrats d'opération sont le sujet du chapitre 11  du livre du cours. Voici les points importants pour le cours :

- On ne spécifie pas les préconditions dans les contrats.
- Un contrat d'opération correspond à une opération système provenant d'un DSS.
- Ne pas confondre les postconditions d'un contrat d'opération et d'un cas d'utilisation. Ce sont deux choses différentes.
- Une postcondition décrit les modifications de l'état des objets dans le modèle du domaine après une opération système.
- Le vocabulaire pour les postconditions provient du modèle du domaine. Il s'agit des noms de classes, d'attributs et d'associations qu'on trouve dans le MDD.
- Chaque postcondition doit avoir la bonne forme :
 - création (ou suppression) d'instances ;
 - modification des valeurs des attributs ;
 - formation (ou rupture) d'associations.
- En rédigeant les contrats, il est normal de découvrir dans le modèle du domaine des incohérences ou des éléments manquants. Il faut les corriger (il faut changer le MDD), car cela fait partie d'un processus itératif et évolutif.

7.1 Qu'est-ce qu'un contrat d'opération

Un contrat d'opération est un document décrivant ce qui est arrivé après l'exécution d'une opération système. Cette description est présentée sous forme de postconditions utilisant le vocabulaire du modèle du domaine.

Le MDD décrit la vraie vie. Il y a des classes conceptuelles (comme Vente) mais aussi des *instances* de ces classes. Dans un magasin, pour chaque nouvelle vente, on imagine une nouvelle instance de la classe Vente. S'il y a eu 72 clients qui ont acheté des choses dans un magasin dans une journée, on imagine 72 instances de Vente, une pour chaque client.

Dans la figure 7.1, l'opération système `créerNouvelleVente()` provient d'un diagramme de séquence système lié au cas d'utilisation *Traiter Vente*. Elle correspond au moment où le caissier démarre une nouvelle vente pour un client. Avant l'exécution de cette opération, l'instance de la classe Vente indiquée dans le modèle du domaine n'existe pas. Cependant, après l'exécution de l'opération système, l'instance de Vente devrait exister. Le contrat d'opération spécifie ce fait dans une postcondition (avec le passé composé en français) : « une instance *v* de Vente a été créée ».

Un contrat d'opération permet de spécifier tous les changements dans le MDD qui doivent avoir lieu lors de l'opération système. Les postconditions du contrat saisissent l'évolution du MDD.

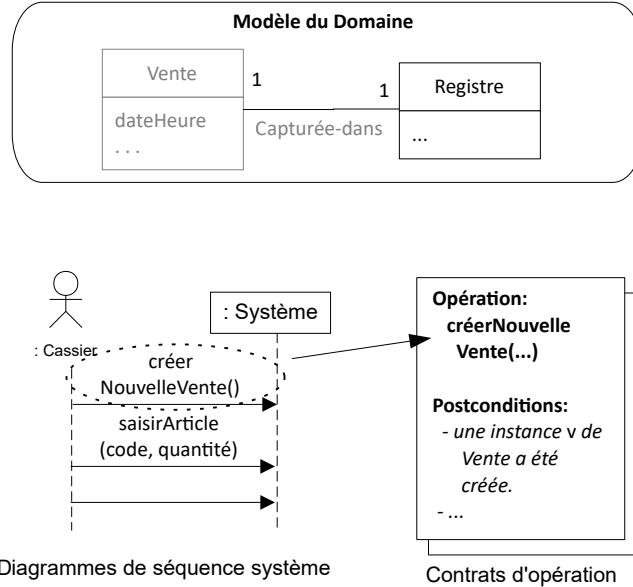


FIGURE 7.1 – Pendant l'opération système `créerNouvelleVente`, une instance de `Vente` doit être créée.
Le contrat d'opération le spécifie dans une postcondition.

7.2 Exemple : Contrats d'opération pour Attaquer un pays

7.2.1 Attaquer un pays

7.2.1.1 Opération : `démarrerAttaque(paysAttaquant:String, paysDéfenseur:String)`

7.2.1.1.1 Postconditions

- une nouvelle instance `a` de `Attaque` a été créée
- `a` a été associée au `Pays` sur une base de correspondance avec `paysAttaquant`
- `a` a été associée au `Pays` sur une base de correspondance avec `paysDéfenseur`

7.2.1.2 Opération : `annoncerAttaque(nbRégimentsAttaquant:int)`

7.2.1.2.1 Postconditions

- `a.nbAttaquant` est devenu `nbRégimentsAttaquant`

7.2.1.3 Opération : `annoncerDéfense(nbRégimentsDéfendant:int)`

7.2.1.3.1 Postconditions

- `a.nbDéfendant` est devenu `nbRégimentsDéfendant`
- L'attribut valeur des `d1` à `d5` est devenue un nombre entier aléatoire entre 1 et 6
- `Occupation.nbRégiments` est ajusté selon le résultat des valeurs sur une base de correspondance avec `paysAttaquant`.

7.2 Exemple : Contrats d'opération pour Attaquer un pays

- Occupation.nbRégiments est ajusté selon le résultat des valeurs sur une base de correspondance avec paysDéfendant.

7.2.1.4 Opération : terminerAttaque ()

7.2.1.4.1 Postconditions

- TODO : Handle the change of Occupation ?

8 Réalisations de cas d'utilisation (RDCU)

Les réalisations de cas d'utilisation (RDCU) sont le sujet du chapitre F17/A18  du livre du cours. Voici les points importants pour le cours :

- Une RDCU est une synthèse des informations spécifiées dans le MDD, le DSS et les contrats d'opération. Si vous n'avez pas bien compris ces autres éléments, il est difficile de réussir les RDCU. Mais, d'une autre part, c'est aussi une étape pour valider votre compréhension. Il est normal de ne pas tout comprendre au début.
- De manière générale, toute bonne RDCU doit faire les choses suivantes :
 - spécifier un contrôleur (pour la première opération système dans un DSS, qui sera le même pour le reste des opérations dans le DSS);
 - satisfaire les postconditions du contrat d'opération correspondant;
 - rechercher les informations qui sont éventuellement rendues à l'acteur dans le DSS.
- Il s'agit d'un diagramme de séquence en UML. Il faut alors maîtriser la notation UML pour ces diagrammes, mais on applique la notation de manière agile :
 - Il n'est pas nécessaire de faire les boîtes d'activation, car ça prend du temps à les faire correctement lorsqu'on dessine à la main un diagramme.
 - On doit se servir des annotations pour documenter les choix (GRASP).
 - On dessine à la main des diagrammes puisqu'on peut faire ça en équipe à un tableau blanc. Mais aussi, à l'examen vous devez faire des diagrammes à la main.
 - Au lieu d'un message pointillé indiquant le retour d'une valeur à la fin de l'exécution d'une méthode, on utilise l'affectation sur le message (comme dans la programmation), par exemple `c = getClient(...)` à la figure 8.4
- Le livre présente quelques RDCU qui sont des *diagrammes de communication*. Cette notation n'est pas utilisée dans le cours, car elle est plus complexe à utiliser et elle est comparable à la notation des diagrammes de séquence.
- Faire des RDCU est plus agile que coder, car dans un diagramme on peut voir le flux de plusieurs messages à travers plusieurs classes. Dans le code source, il serait nécessaire d'ouvrir plusieurs fichiers en même temps et on ne peut pas voir toute la dynamique de la même manière. Faire des changements à un diagramme est aussi plus facile que changer tout le code source. On peut également se servir des structures (List, Array, Map, etc.) avant que celles-ci ne soient créées.
- Faire des RDCU est une activité créative, mais *le codage dans un langage de programmation est la seule manière de valider une RDCU*. Évidemment, la programmation prend beaucoup plus de temps et n'est pas triviale. Faire une RDCU est comme faire un *plan* pour un bâtiment tandis que faire de la programmation est comme la *construction* du bâtiment. Si un plan contient des erreurs de conception, on va les savoir lors de la construction.

Tout le processus de proposer une solution (RDCU) peut être visualisé comme un diagramme d'activités, comme dans la figure 8.1.

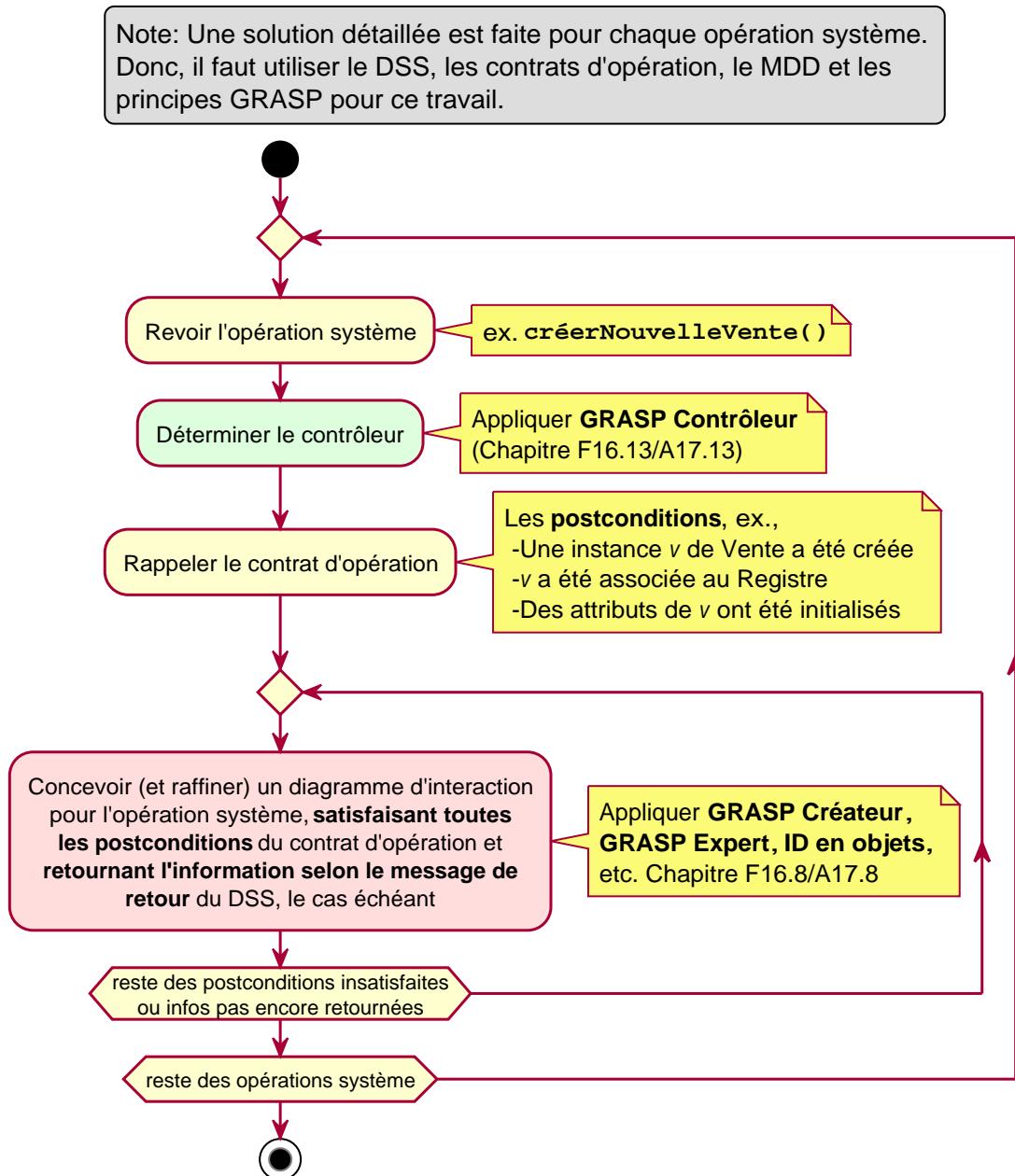


FIGURE 8.1 – Aide mémoire pour faire une RDCU. L'étape en rouge nécessite beaucoup de pratique, selon la complexité des postconditions. Vous pouvez vous attendre à ne pas la réussir du premier coup.
(PlantUML)

8.1 Spécifier le contrôleur

Pour commencer une RDCU, on spécifie le contrôleur selon GRASP. Dans les travaux pour le cours, vous devez indiquer *pourquoi vous avez choisi telle classe pour être le contrôleur*. Ce n'est pas un choix arbitraire. Référez-vous à la définition dans [Tableau des principes GRASP](#).

Pour initialiser les liens entre la couche présentation et les contrôleurs GRASP, Larman vous propose de le faire dans la [RDCU pour l'initialisation, le scénario Démarrer](#).

8.2 Satisfaire les postconditions

8.2.1 Créer une instance

Certaines postconditions concernent la création d'une instance. Dans votre RDCU, vous devez respecter le GRASP Créeur. Référez-vous à la définition dans le [Tableau des principes GRASP](#).

⚠ Une erreur potentielle est de donner la responsabilité de créer à un contrôleur, puisqu'*il a les données pour initialiser* l'objet. Bien que ce soit justifiable par le principe GRASP Créeur, il vaut mieux favoriser une classe qui *agrège* l'objet à créer, le cas échéant.

8.2.2 Former une association

Pour les postconditions où il faut former une association entre un objet *a* et *b*, il y a plusieurs façons de faire.

- S'il y a une agrégation entre les objets, il s'agit probablement d'une méthode `add()` sur l'objet qui agrège.
- S'il y a une association simple, il faut considérer la navigabilité de l'association. Est-ce qu'il faut pouvoir retrouver l'objet *a* à partir de l'objet *b* ou vice-versa ? Il s'agira d'une méthode `setB(b)` sur l'objet *a* (pour trouver *b* à partir de *a*), etc.
- S'il faut former une association entre un objet et un autre « sur une base de correspondance avec » un identifiant passé comme argument, alors il faut repérer le bon objet d'abord. Voir la section [Transformer identifiants en objets](#).

Dans la plupart des cas, la justification GRASP pour former une association est Expert, défini dans le [Tableau des principes GRASP](#). Il faut faire attention à la [visibilité](#).

8.2.3 Modifier un attribut

Pour les postconditions où il faut modifier un attribut, c'est assez évident. Il suffit de suivre le principe GRASP Expert, défini dans le [Tableau des principes GRASP](#). Très souvent, c'est une méthode `setX(valeur)` où *X* correspond à l'attribut qui sera modifié à *valeur*. Attention à la [visibilité](#).

Lorsque l'attribut d'un objet doit être modifié juste après la création de ce dernier, ça peut se faire dans le constructeur, comme on voit dans la figure [8.2](#).

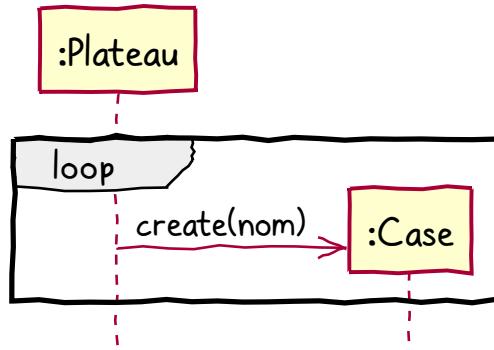


FIGURE 8.2 – Combiner la création d’instance et une modification de son attribut dans un constructeur. (PlantUML)

8.3 Visibilité

Dans tous les cas, si un message est envoyé à un objet, ce dernier doit être *visible* à l’objet qui lui envoie le message. Régler les problèmes de visibilité nécessite de la créativité. Il est difficile à enseigner cette démarche, mais les points suivants peuvent aider :

- Pour un objet racine (par exemple Université) il peut s’agir d’un objet Singleton, qui aura une visibilité globale. C’est-à-dire que n’importe quel objet pourrait lui envoyer un message. Cependant, les objets Singleton posent des problèmes de conception, notamment pour les tests. Il vaut mieux éviter ce choix, si possible.
☞ Voir cette réponse sur stackoverflow.
- Sinon, il faudra que l’objet émetteur ait une référence de l’objet récepteur. Par exemple dans la figure 8.3, la référence à *b* peut être :
 - stockée comme un attribut de *a*,
 - passée comme un argument dans un message antérieur, ou
 - affectée dans une variable locale de la méthode où unMessage() sera envoyé.

Pour plus de détails, voir le chapitre sur la Visiblilté ☞ du livre du cours.

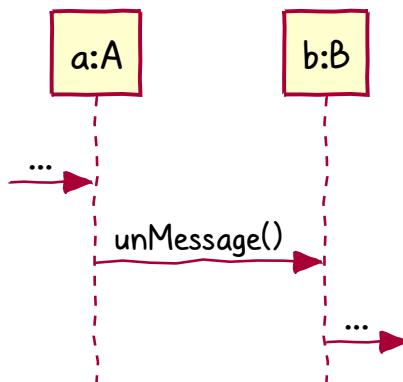


FIGURE 8.3 – L’objet *b* doit être visible à l’objet *a* si *a* veut lui envoyer un message. (PlantUML)

Pour initialiser les références nécessaires pour la bonne visibilité, Larman vous propose de faire ça dans la RDCU pour l'initialisation, le scénario Démarrer.

8.4 Transformer identifiants en objets

La directive d'utiliser les types primitifs pour les opérations système nous mène à un problème récurrent dans les RDCU : transformer un identifiant (souvent de type `String` ou `int`) en objet. Larman vous propose un idiomme (pas vraiment un patron) nommé **Transformer identifiant en objet** qui sert à repérer la référence d'un objet qui correspond à l'identifiant.

Il y a un exemple à la figure 8.4 provenant du chapitre sur l'**Application des patterns GoF (Figure 23.18)** du livre du cours. Un autre exemple du livre du cours est l'identifiant `codeArticle` transformé en objet `DescriptionProduit` par la méthode

```
CatalogueProduits.getDescProduit(codeArticle:String):DescriptionProduit
```

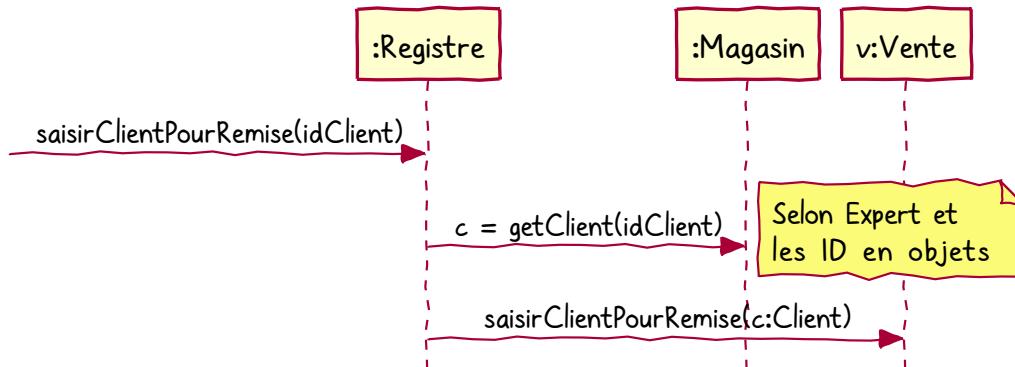


FIGURE 8.4 – Un identifiant `idClient:String` est transformé en objet `c:Client`, qui est ensuite envoyé à la Vente en cours. (PlantUML)

La section 8.5 explique comment implémenter la transformation avec un tableau associatif.

8.5 Utilisation de tableau associatif (`Map<clé, objet>`)

Pour *transformer un ID en objets*, il est pratique d'utiliser un **tableau associatif** (aussi appelé **dictionnaire ou map en anglais**). L'exemple du livre du cours concerne le problème de repérer une Case Monopoly à partir de son nom (`String`). C'est la figure A17.7 ou F17.7 du livre du cours. Voir les détails dans le livre.

Notez que les exemples du livre ne montrent qu'un seul *type* dans le tableau associatif, par exemple `Map<Case>`, tandis que normalement il faut spécifier aussi le type de la clé, par exemple `Map<String, Case>`.

Un tableau associatif fournit une méthode `get` ou `find` pour rechercher un objet à partir de sa clé (son identifiant). La figure 8.5 en est un exemple.

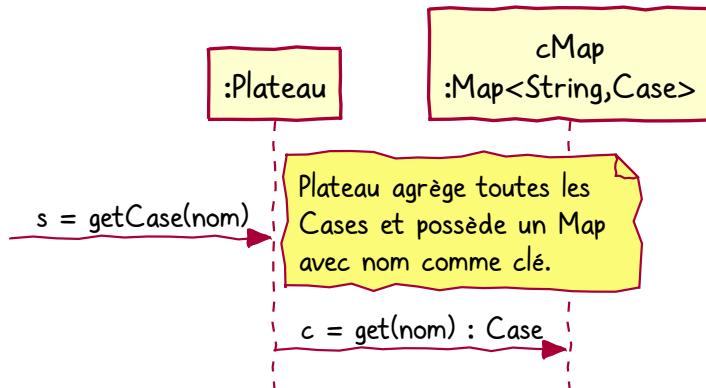


FIGURE 8.5 – Exemple de l'utilisation de tableau associatif pour trouver une Case Monopoly à partir de son nom. ([PlantUML](#))

Dans la section suivante, l'initialisation des éléments utilisés dans les RDCU (comme des tableaux associatifs) est expliquée.

8.6 RDCU pour l'initialisation, le scénario Démarrer

Le lancement de l'application correspond à la RDCU « Démarrer ». La section **Initialisation et cas d'utilisation Démarrer** (Sec. 17.4, p.345 dans le livre en français) ou **Initialization and the “Start Up” Use Case** (Sec. 18.4, p.274 dans le livre en anglais) traite ce sujet important. C'est dans cette conception où il faut mettre en place tous les éléments importants pour les hypothèses faites dans les autres RDCU, par exemple les classes de collection (Map), les références pour la visibilité, l'initialisation des contrôleurs, etc.

Voici quelques points importants :

- Le lancement d'une application dépend du langage de programmation et du système d'exploitation.
- À chaque nouvelle RDCU, on doit possiblement actualiser la RDCU « Démarrer » pour tenir compte des hypothèses faites dans la dernière RDCU. Elle est assez « instable » pour cette raison. Larman recommande de faire sa conception en dernier lieu.
- Il faut choisir l'objet du domaine initial, qui est souvent l'objet racine, mais ça dépend du domaine. Cet objet aura la responsabilité, lors de sa création, de générer ses « enfants » directs, puis chaque enfant aura à faire la même chose selon la structure. Par exemple, selon le MDD pour le jeu de Risk à la figure 5.1, JeuRisk pourrait être l'objet racine, qui devra créer l'objet PlateauRisk et les cinq instances de Dé. L'objet PlateauRisk, lors de son initialisation, pourra instancier les 42 objets Pays et les six objets Continent, en passant à chaque Continent leurs objets Pays lors de son initialisation. Si PlateauRisk fournit une méthode `getPays(nom)` qui dépend d'un tableau associatif selon [Transformer identifiants en objets](#), alors c'est dans l'initialisation de cette classe où l'instance de `Map<String, Pays>` sera créée.
- Selon l'application, les objets peuvent être chargés en mémoire à partir d'un système de persistance, par exemple une base de données ou un fichier. Pour l'exemple de Risk, PlateauRisk pourrait charger, à partir d'un fichier JSON, des données pour initialiser toutes les instances de Pays. Pour une application d'inscription de cours à l'université, il se peut que toutes les descriptions de cours soient chargées en mémoire à partir d'une base de données. Une base de données amène un lot

d'avantages et d'inconvénients, et elle n'est pas toujours nécessaire. Dans LOG210, on n'aborde pas le problème de base de données (c'est le sujet d'un autre cours).

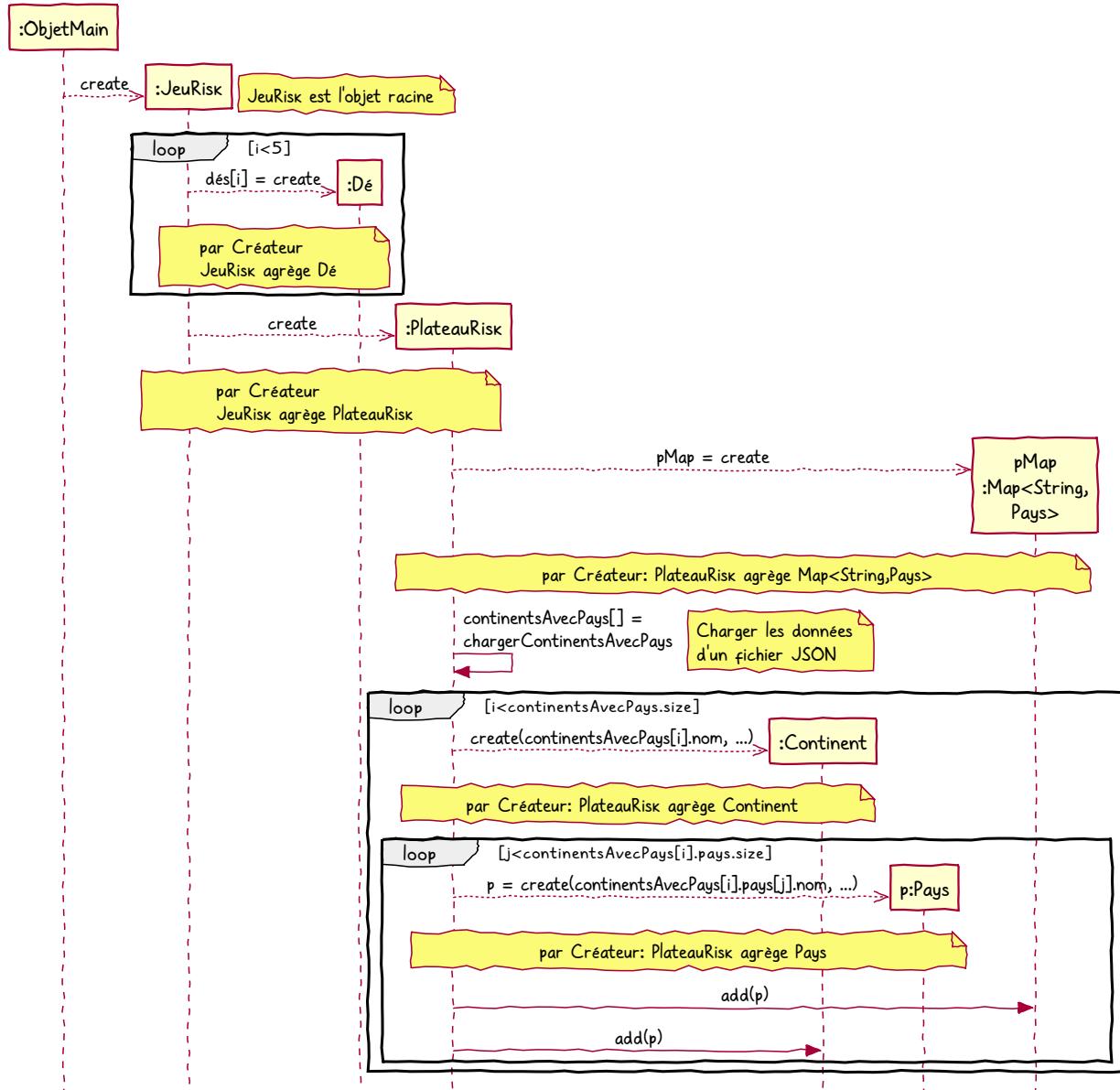


FIGURE 8.6 – Exemple de l'initialisation partielle du jeu de Risk. (PlantUML)

9 Développement piloté par les tests

Si on écrivait des logiciels pouvant se tester automatiquement ? Le développement piloté par les tests (anglais, *test-driven development, TDD*) est une pratique populaire et intéressante. Il s'agit d'écrire des logiciels avec un composant d'autovalidation (des tests automatisés). Mais, écrire beaucoup de tests n'est pas toujours une tâche agréable pour des développeurs. Historiquement, si on attend la fin d'un projet pour écrire des tests, il ne reste plus beaucoup de temps et l'équipe laisse tomber les tests. Pour pallier ce problème, le développement piloté par les tests propose de travailler **en petits pas**. C'est-à-dire écrire un test simple (en premier), puis écrire la partie du logiciel pour passer le test de manière simple (le plus simple). Ça fait moins de codage entre les validations et c'est probablement même plus stimulant pour les développeurs.

Ainsi, il y a toujours des tests pour les fonctionnalités et le développement se fait en petits incrément qui sont validés par les tests. Faire les *petits pas* réduit le risque associé à de gros changements dans un logiciel sans validation intermédiaire. Au fur et à mesure qu'on développe un logiciel, on développe également quelques tests de ce dernier. Puisque les tests sont automatiques, ils sont aussi faciles à exécuter que le compilateur.

Il y a une discipline imposée dans le TDD qui nécessite d'écrire un *test en premier*, c'est-à-dire *avant* d'écrire le code. La démarche est illustrée par la figure 9.1. Beaucoup d'outils (IDE) favorisent ce genre de développement. Nous pouvons écrire un test qui appelle à une fonction qui n'existe pas encore et l'IDE va nous proposer un squelette de la méthode, avec les arguments et une valeur de retour même. Un puriste du TDD insistera que le test soit écrit toujours en premier ! Cette discipline est parfois culturelle.

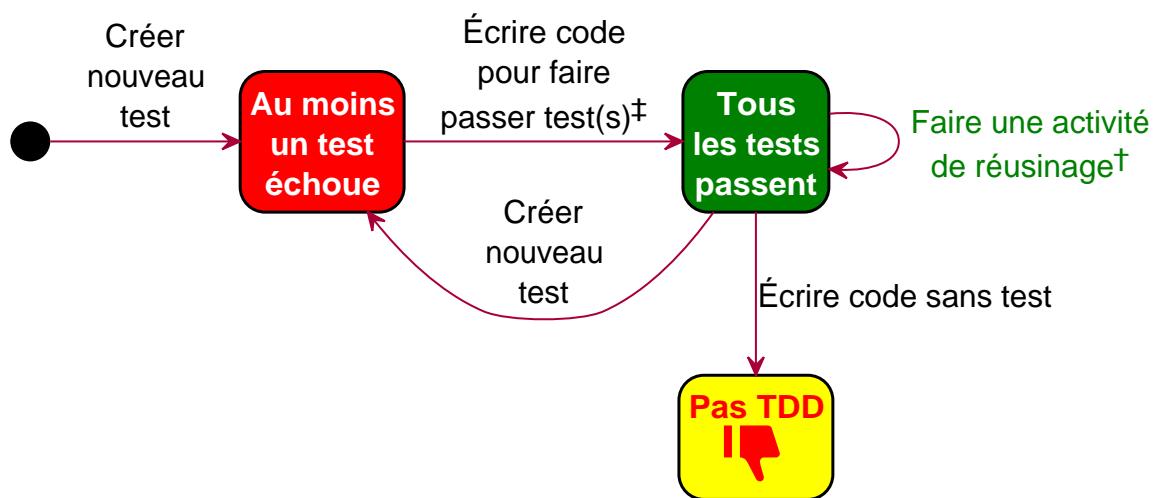
Plusieurs chercheurs ont mené des expériences, par exemple (Karac & Turhan, 2018), pour voir si tester en premier avait un vrai bénéfice. Les résultats de leurs analyses n'ont pas toujours montré que c'est le cas (ce genre d'expérience est difficile de faire, en partie parce qu'il n'y a pas beaucoup de développeurs en industrie qui le pratiquent). Les chercheurs ont trouvé que faire un petit test *après* avoir écrit le code a aussi un bénéfice sur le plan de la qualité. Dans tous les cas, des chercheurs ont trouvé que le fait de travailler en *petits pas* apporte *toujours* un avantage sur le plan de la qualité. Travailler en *petits pas* est utile, même sans faire du TDD.

Sachez qu'il existe beaucoup d'intergiciels (anglais *frameworks*) pour faciliter l'exécution automatique des tests réalisés dans le cadre du TDD. Pour Java il y a JUnit, mais il y en a pour pratiquement tous les langages et environnements. En ce qui concerne le squelette pour le laboratoire de LOG210, la version TypeScript utilise [Mocha](#) avec [Chai](#). La version Python utilise [PyTest](#).

L'exécution de tests peut être même faite à chaque commit du code dans un dépôt comme GitHub. Le squelette de LOG210 utilise [Travis](#) pour cela. Cependant, pour un dépôt privé, Travis nécessite un compte payant.

Il est possible de mesurer la [couverture de code W](#) atteinte par les tests (mais ce sujet commence à sortir du cadre de la matière du cours). Les deux versions du squelette utilisent [Coveralls](#) pour ce faire.

Les activités de réusinage sont expliquées dans la section [Réusinage \(Refactorisation\)](#).



†Une activité de réusinage n'est pas censée causer un problème avec un test. Cependant, cela peut arriver que les tests ne passent plus et il faudra corriger des problèmes.

†Il peut arriver qu'un test soit mal codé (il contient un bogue). Dans ce cas, on corrige le code du test.

FIGURE 9.1 – États du développement piloté par les tests. ([PlantUML](#))

9.1 Kata TDD

Pour apprendre à faire du développement piloté par les tests (et pour apprendre les cadriels supportant l'automatisation des tests), il existe une activité nommée « kata TDD ». *Kata* est un terme japonais désignant une séquence de techniques réalisée dans le vide dans les arts martiaux japonais. [En voici une vidéo](#) YouTube. C'est un outil de transmission de techniques et de principes de combat.



FIGURE 9.2 – Étudiante de karaté faisant le kata *Basai Dai* (photo « Karate » (CC BY-SA 2.0) par [The Consortium](#)).

Alors, le « kata TDD » a été proposé par Dave Thomas et le but est de développer la fluidité avec le développement piloté par les tests. Un kata TDD se pratique avec un IDE (environnement de développement logiciel) et un support pour les tests (par exemple JUnit). Pratiquer le même kata peut améliorer votre habileté de programmation. On peut pratiquer le même kata dans un langage différent ou avec un IDE ou environnement de test différent. Le kata vous permet d'avoir une facilité avec les aspects techniques de développement dans plusieurs dimensions (complétion de code pour test et pour l'application, API de l'environnement de test, etc.).

En plus, les activités de réusinage sont normalement intégrées dans un kata. Le fait de travailler en *petits pas* peut faire en sorte que la dette technique s'accumule. Les IDE facilitent l'application des activités de réusinage. Un langage fortement typé comme Java permet d'avoir plus de fonctionnalités automatisées de réusinage dans un IDE qu'un langage dynamique comme JavaScript ou Python. Une activité de base de réusinage est le renommage d'une variable ou d'une fonction. Le réusinage rend le code plus facile à comprendre et à maintenir.

9.1.1 Exemple de Kata TDD FizzBuzz

L'inspiration de cet exercice vient de <http://codingdojo.org/kata/FizzBuzz/>.

Dans cet exercice, il faut écrire par le développement piloté par les tests un programme qui imprime les nombres de 1 à 100. Mais pour les multiples de trois, il faut imprimer Fizz au lieu du nombre et pour les multiples de cinq, il faut imprimer Buzz. Pour les nombres étant des multiples de trois et cinq il faut imprimer FizzBuzz. Voici un exemple des sorties :

```
1  
2  
Fizz  
4  
Buzz  
Fizz  
7  
8  
Fizz  
Buzz  
11  
Fizz  
13  
14  
FizzBuzz  
16  
17  
Fizz  
19  
Buzz  
... etc. jusqu'à 100
```

9.1.1.1 Préalables

Il faut installer un IDE qui supporte les activités de réusinage (refactorings) comme VisualStudio Code, Eclipse, IntelliJ, etc. puis un framework de test (JUnit, Mocha/Chai, unittest, etc.). Pour un exemple qui fonctionne en TypeScript, vous pouvez cloner le code à [ce dépôt](#).

9.1.1.2 Déroulement

Cet exercice peut se faire individuellement ou en équipe de deux. En équipe, une personne écrit le test et l'autre écrit le code pour passer le test (c'est la variante ping-pong). Chacun réfléchit aux activités de réusinage éventuelles lorsque le projet est dans l'état vert (Figure 9.1). Les membres de l'équipe peuvent changer de rôle (testeur, codeur) après un certain nombre d'étapes, ou après avoir terminé le kata entier.

Pour respecter la philosophie de *petits pas*, il vaut mieux :

- ne lire que l'étape courante ;
- ne travailler que sur l'étape courante ;

- ne faire que les tests avec les entrées valides.

9.1.1.3 Kata pour FizzBuzz

Les étapes sont simples et précises. Il s'agit de créer une classe ayant une méthode acceptant un entier et renvoyant une valeur selon les exigences de FizzBuzz décrite plus haut.

1. Un argument de 1 retourne 1.
2. Un argument de 2 retourne 2
3. Un argument de 3 retourne Fizz
4. Un argument de 6 retourne Fizz
5. Un argument de 5 retourne Buzz
6. Un argument de 10 retourne Buzz
7. Un argument de 15 retourne FizzBuzz
8. Un argument de 30 retourne FizzBuzz
9. Supporter des exigences qui évoluent :
 - a. Il faut imprimer Fizz au lieu du nombre si le nombre est un multiple de 3 ou contient un 3 (ex. 13 → Fizz).
 - b. Il faut imprimer Buzz au lieu du nombre si le nombre est un multiple de 5 ou contient un 5 (ex. 51 → Fizz).
 - c. Il faut imprimer FizzBuzz si le nombre est un multiple de 5 et de 3 ou contient un 5 et un 3 (ex. 53 → FizzBuzz).

10 Réusinage (Refactorisation)

10.1 Introduction

Considérez l'histoire suivante, provenant de la 2e édition du livre *Refactoring* de Martin Fowler (2019) :

Il était une fois, un consultant qui a rendu visite à un projet de développement afin de regarder une partie du code qui avait été écrit. En parcourant la hiérarchie des classes au centre du système, le consultant l'a trouvée plutôt désordonnée. Les classes de niveau supérieur ont émis certaines hypothèses sur la façon dont les classes fonctionneraient, hypothèses incorporées dans le code hérité. Ce code n'était pas cohérent avec toutes les sous-classes, cependant, et a été redéfini à beaucoup d'endroits. De légères modifications à la superclasse auraient considérablement réduit la nécessité de la redéfinir. À d'autres endroits, l'intention de la superclasse n'avait pas été bien comprise et le comportement présent dans la superclasse était dupliqué. Dans d'autres endroits encore, plusieurs sous-classes ont fait la même chose avec du code qui pouvait clairement être déplacé dans la hiérarchie.

Le consultant a recommandé à la direction du projet que le code soit examiné et nettoyé, mais la direction du projet n'était pas enthousiaste. Le code semblait fonctionner et il y avait des contraintes sur l'emploi du temps considérables. Les gestionnaires ont dit qu'ils y parviendraient ultérieurement.

Le consultant a également montré ce qui se passait aux programmeurs travaillant sur la hiérarchie. Les programmeurs étaient enthousiastes et ont vu le problème. Ils savaient que ce n'était pas vraiment de leur faute; parfois, l'évaluation par une autre personne est nécessaire pour détecter le problème. Les programmeurs ont donc passé un jour ou deux à nettoyer la hiérarchie. Une fois terminés, ils avaient supprimé la moitié du code de la hiérarchie sans réduire sa fonctionnalité. Ils étaient satisfaits du résultat et ont constaté qu'il était devenu plus rapide et plus facile d'ajouter de nouvelles classes et d'utiliser les classes dans le reste du système.

La direction du projet n'était pas contente. L'échéancier était serré et il y avait beaucoup de travail à faire. Ces deux programmeurs avaient passé deux jours à effectuer un travail qui n'ajoutait rien aux nombreuses fonctionnalités que le système devait offrir en quelques mois. L'ancien code avait très bien fonctionné. Oui, la conception était un peu plus « pure » et un peu plus « propre ». Mais le projet devait expédier du code qui fonctionnait, pas du code qui plairait à un universitaire. Le consultant a suggéré qu'un nettoyage similaire soit effectué sur d'autres parties centrales du système, ce qui pourrait interrompre le projet pendant une semaine ou deux. Tout cela était pour rendre le code plus beau, pas pour lui faire faire ce qu'il ne faisait pas déjà.

Cette histoire est un bon exemple des deux forces constamment en jeu lors d'un développement de logiciel. D'un côté, on veut que le code fonctionne (pour satisfaire les fonctionnalités) et d'un autre côté, on veut

que la conception soit acceptable puisqu'il y a d'autres exigences sur un logiciel telles que la maintenabilité, l'extensibilité, etc. La section sur le [Spectre de la conception, adapté de Neal Ford. \(PlantUML\)](#) aborde cette dynamique.

Le réusinage (anglais *refactoring*) est, selon Fowler, « l'amélioration de la conception du code après avoir écrit celui-ci ». Il s'agit de retravailler le code source de façon à en améliorer la lisibilité ou la structure, sans en modifier le fonctionnement. C'est une manière de gérer la dette technique, car grâce au réusinage on peut transformer du code chaotique (écrit peut-être par les gens en mode « hacking cowboy ») en code bien structuré. De plus, beaucoup d'IDE supportent l'automatisation d'activités de réusinage, rendant le processus plus facile et robuste. Probablement vous avez déjà « renommé » une variable dans le code source, à travers un menu « Refactoring ».

10.2 Symptômes de la mauvaise conception - Code smells

En anglais, le terme « Code smell » a été proposé par Fowler pour les symptômes d'une mauvaise conception. Le but est de savoir à quel moment il faut affecter des activités de réusinage.

Par exemple, le premier « smell » dans son livre est « Mysterious Name ». Il apparaît lorsqu'on voit une variable ou une méthode dont le nom est incohérent avec son utilisation. Cela arrive puisqu'il n'est pas toujours facile de trouver un bon nom au moment où on est en train d'écrire du code. Plutôt que de rester bloqué sur le choix, on met un nom arbitraire (ou peut-être par naïveté on se trompe carrément de nom). Alors, si vous observez ce problème (smell) dans un logiciel, vous n'avez qu'à appliquer l'activité de réusinage nommée [Change Function Declaration](#), [Rename Field](#) ou [Rename Variable](#), selon le cas.

Un autre exemple serait que vous avez un programme assez complexe, mais avec seulement une ou deux classes. Ces classes ont beaucoup d'attributs et de méthodes. Alors, ce « smell » s'appelle « Large class » et la solution est d'appliquer des activités de réusinage [Extract Class](#), ou éventuellement [Extract Superclass](#) ou [Replace Type Code with Subclasses](#). Avec un IDE dominant et un langage populaire, vous aurez probablement des fonctionnalités pour supporter l'automatisation de ces activités de réusinage.

Certaines activités traitent des sujets avancés en conception, mais c'est très intéressant pour ceux qui aiment le bon design. Voici la liste complète des « smells » ainsi que les activités de réusinage à appliquer (voir le catalogue sur le site web pour les détails).

Symptôme de mauvaise conception (« Smell »)	Activités de réusinage à appliquer éventuellement
Mysterious Name	Change Function Declaration , Rename Variable , Rename Field
Duplicated Code	Extract Function , Slide Statements , Pull Up Method
Long Function	Extract Function , Replace Temp with Query , Introduce Parameter Object , Preserve Whole Object , Replace Function with Command , Decompose Conditional , Replace Conditional with Polymorphism , Split Loop
Long Parameter List	Replace Parameter with Query , Preserve Whole Object , Introduce Parameter Object , Remove Flag Argument , Combine Functions into Class
Global Data	Encapsulate Variable

Symptôme de mauvaise conception (« Smell »)	Activités de réusinage à appliquer éventuellement
Mutable Data	Encapsulate Variable, Split Variable, Slide Statements, Extract Function, Separate Query from Modifier, Remove Setting Method, Replace Derived Variable with Query, Use Combine Functions into Class, Combine functions into Transform, Change Reference to Value
Divergent Change	Split Phase, Move Function, Extract Function, Extract Class
Shotgun Surgery	Move Function, Move Field, Combine Functions into Class, Combine Functions into Transform, Split Phase, Inline Function, Inline Class
Feature Envy	Move Function, Extract Function
Data Clumps	Extract Class, Introduce Parameter Object, Preserve Whole Object
Primitive Obsession	Replace Primitive with Object, Type Code with Subclasses, Replace Conditional with Polymorphism, Extract Class, Introduce Parameter Object
Repeated Switches	Replace Conditional with Polymorphism
Loops	Replace Loop with Pipeline
Lazy Element	Inline Function, Inline Class, Collapse Hierarchy
Speculative Generality	Collapse Hierarchy, Inline Function, Inline Class, Change Function Declaration, Remove Dead Code
Temporary Field	Extract Class, Move Function, Introduce Special Case
Message Chains	Hide Delegate, Extract Function, Move Function
Middle Man	Remove Middle Man, Inline Function, Replace Superclass with Delegate, Replace Subclass with Delegate
Insider Trading	Move Function, Move Field, Hide Delegate, Replace Subclass with Delegate, Replace Superclass with Delegate
Large Class	Extract Class, Extract Superclass, Replace Type Code with Subclasses
Alternative Classes with Different Interfaces	Change Function Declaration, Move Function, Extract Superclass
Data Class	Encapsulate Record, Remove Setting Method, Move Function, Extract Function, Split Phase
Refused Bequest	Push Down Method, Push Down Field, Replace Subclass with Delegate, Replace Superclass with Delegate
Comments	Extract Function, Change Function Declaration, Introduce Assertion

10.3 Automatisation du réusinage par les IDE

À la **section F19.2/A22.2** du livre du cours, le sujet de réusinage est abordé. Il y a plusieurs exemples de base détaillés qui sont automatisés par les IDE dominants tels que Eclipse, IntelliJ IDEA, JetBrains PyCharm, Visual Studio Code, etc. Il se peut que dans un avenir proche le réusinage (l'amélioration du design) devienne une activité réalisée par des algorithmes d'intelligence artificielle.

Pour plus d'activités de réusinage, il y a le catalogue du site refactoring.com.

Voir [cette page web pour savoir comment les activités de réusinage sont faites dans Visual Studio Code](#). D'autres automatisations sont implémentées par des extensions.

10.4 Impropriété

Si quelqu'un dit que son code est cassé pendant plusieurs jours parce qu'il fait du réusinage, ce n'est pas la bonne utilisation du terme [selon Martin Fowler](#). Il s'agit de la *restructuration* dans ce cas.

Le réusinage est basé sur les petites transformations qui ne changent pas le comportement du logiciel.

11 Développement de logiciel en équipe

Le développement de logiciels se fait souvent en équipe. Cependant, il y a des défis pour travailler en équipe. Souvent, avant l'université on apprend comment s'organiser en équipe, faire des rencontres, répartir le travail, planifier, etc. Pourtant, il y a d'autres défis dans ce travail, des défis sur le plan humain. C'est le sujet du livre « Team Geek » (2012) écrit par Brian W. Fitzpatrick (anciennement de Google) et Ben Collins-Sussman (Subversion, Google).

Aujourd'hui, la demande pour le talent en technologies de l'information est importante. Les technologies évoluent constamment. Qui développe du code pour Flash W ces jours-ci ? Cette technologie est quasiment désuète. Cependant, une chose qui ne change pas (beaucoup) est le comportement humain.

Les auteurs de « Team Geek » abordent les problèmes dus aux tendances comportementales chez les développeurs. Par exemple, une personne n'a pas toujours envie de montrer son code source à ses coéquipiers pour plusieurs raisons :

- Son code n'est pas fini.
- Elle a peur d'être jugée.
- Elle a peur que quelqu'un vole son idée.

Dans tous ces cas, il s'agit de l'insécurité et c'est tout à fait normal. Par contre, ce genre de comportement augmente certains risques dans le développement :

- de faire des erreurs dans la conception initiale ;
- de « réinventer la roue » ;
- de terminer le travail plus tard que son compétiteur, qui, lui, a collaboré avec son équipe.

Les auteurs le disent et c'est un fait : si nous sommes tous plus ou moins compétents sur le plan technique, ce qui fera la différence importante dans une carrière est notre habileté à collaborer avec les autres.

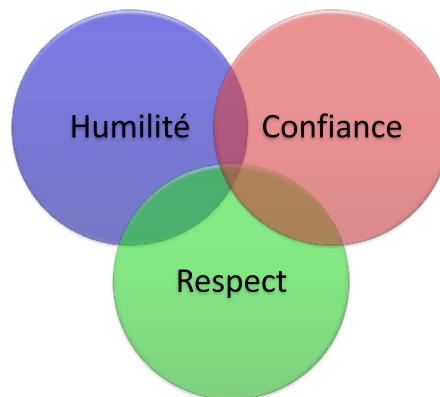


FIGURE 11.1 – Pratiquement tout conflit social est dû à un manque d'humilité, de respect ou de confiance.

11.1 Humilité, Respect, Confiance

L'humilité, le respect et la confiance (voir la figure 11.1) sont les qualités de base pour le bon travail en équipe. Cette section présente ces aspects en détail.

11.1.1 Humilité

Voici la définition d'*humilité* selon Antidote :

Disposition à s'abaisser volontairement, par sentiment de sa propre faiblesse.

Une personne humble pense ainsi :

- Je ne suis pas le centre de l'univers.
- Je ne suis ni omniscient ni infaillible.
- Je suis ouvert à m'améliorer.

⚠ L'humilité ne veut pas dire « je n'ai pas de valeur » ou « j'accepte d'être mal traité.e par les autres ».



FIGURE 11.2 – Éviter d'être le « Centre de l'univers » (CC BY-NC-ND 2.0) par Diamonddust.



FIGURE 11.3 – « Missing » (CC BY-SA 2.0) par smkybear.

Quelques exemples concrets d'humilité dans le développement :

- Un coéquipier débutant en JavaScript, git, etc. va le reconnaître et va même faire des exercices sur Internet pour s'améliorer.
- Un coéquipier (même le chef d'équipe) qui a pris une mauvaise décision (technique ou autre) va l'avouer. Il sait que les autres ne sont pas là pour l'attaquer (ils le respectent).

Moi < Équipe

FIGURE 11.4 – Un coéquipier humble va accepter une décision prise par l'équipe, même s'il n'était pas en accord à 100%. ([PlantUML](#))

- Un coéquipier va travailler fort pour que *son équipe* réussisse.
- Un coéquipier qui reçoit une critique ne va pas la prendre personnellement. Il sait que la qualité de son code n'équivaut pas à son estime de soi. (Cela n'est pas toujours facile !)

11.1.2 Respect

Une personne démontrant du respect pense ainsi :

- Je me soucie des gens avec qui je travaille.
- Je les traite comme des êtres humains.
- J'ai de l'estime pour leurs capacités et leurs réalisations.

11.1.3 Confiance

Une personne démontrant la confiance pense ainsi :

- Je crois que les autres coéquipiers sont compétents et qu'ils feront la bonne chose.
- Je suis à l'aise lorsqu'ils prennent le volant, le cas échéant.

Le dernier point peut être extrêmement difficile si vous avez déjà été déçu par une personne incompétente à qui vous avez délégué une tâche.

11.2 Redondance des compétences dans l'équipe (Bus Factor)

Pour qu'une équipe soit robuste, il faut une redondance des compétences. Sinon, la perte d'un coéquipier (pour une raison quelconque) peut engendrer de graves conséquences, voire arrêter carrément le développement. Ce principe a été nommé en anglais *Bus factor*. C'est le nombre minimum de coéquipiers à perdre (heurtés par un bus) pour arrêter le projet par manque de personnel bien informé ou compétent. Par exemple, dans un projet de stage, si c'est vous qui écrivez tout le code, alors c'est un *bus factor* de 1. Si vous n'êtes plus présent, le projet s'arrête.

Un coéquipier peut être absent (ou moins disponible) pour des raisons moins graves, par exemple, il part en vacances, il tombe malade, il prend un congé parental, il change d'emploi, ou il abandonne le cours. Cherchez à répartir les responsabilités dans l'équipe afin d'avoir un *bus factor* d'au moins 2. Partagez des compétences pour maintenir une équipe robuste. Vous pouvez également garder votre solution *simple* et garder la documentation de votre conception à jour.

⚠ Si un coéquipier quitte en cours de la session, il n'est pas facile de maintenir le même rythme. Cependant, les enseignants et les chargés de laboratoire de LOG210 s'attendront à ce que vous ayez pensé à un plan B avant de perdre le coéquipier. Au moins un autre coéquipier doit être au courant de ce que faisait l'ancien coéquipier, pour que le projet ne soit pas complètement arrêté.

11.3 Mentorat

En LOG210, ça peut être l'enseignant qui décide la composition des équipes. Ça veut dire que forcément certains coéquipiers ont plus d'expérience et de facilité à faire certaines tâches que d'autres. Les équipes doivent composer avec cette diversité. C'est une approche pédagogique reconnue par les experts.

Selon TeamGeek :

Si vous avez déjà un bon bagage en programmation, ça peut être pénible de voir un coéquipier moins expérimenté tente un travail qui lui prendra beaucoup de temps lorsque vous savez que ça vous prendra juste quelques minutes. Apprendre à quelqu'un comment faire une tâche et lui donner l'occasion d'évoluer tout seul est un défi au début, mais cela est une caractéristique importante du leadership.

Si les plus forts n'aident pas les autres, ils risquent de les éloigner de l'équipe et de se trouver seuls sur le plan des contributions techniques. Voir la section sur la [Redondance des compétences dans l'équipe \(Bus Factor\)](#).

Encadrer un coéquipier au début de la session peut prendre beaucoup de temps. Mais, si la personne devient plus autonome, c'est un gain pour toute l'équipe. Cela augmente également le *bus factor*.

Voici quelques conseils pour le mentorat :

- avoir les compétences sur un plan technique ;
- être capable d'expliquer des choses à quelqu'un d'autre ;
- savoir combien d'aide à donner à la personne encadrée.

Selon TeamGeek, le dernier point est important parce que si vous donnez trop d'information, la personne peut vous ignorer plutôt que vous dire gentiment qu'elle a compris.

11.4 Scénarios

Voici des situations qui pourraient arriver dans une session de LOG210 :

- un coéquipier se trouve à être le seul à faire de la programmation.
- il ne fait plus confiance à ses coéquipiers, car leur code est trop bogué.
- il n'a pas la patience pour accommoder les coéquipiers moins expérimentés.
- il croit qu'ils auraient dû apprendre mieux à programmer dans les cours préalables.
- un coéquipier dit qu'il a « fait ses 3 heures de contribution » chaque dimanche chez lui et que ça devrait suffire pour sa partie (il a un emploi et n'a pas beaucoup de temps disponible pour l'équipe d'un laboratoire).

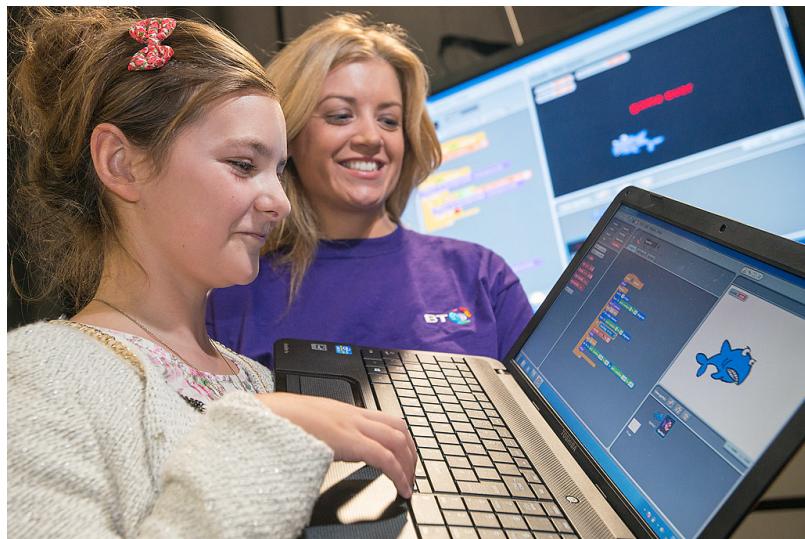


FIGURE 11.5 – Encadrer les coéquipiers est une habileté à mettre sur son CV « CultureTECH BT Monster Dojo » (CC BY 2.0) par [connor2nz](#).

- un ou deux membres d'une équipe abandonnent le cours après l'examen intra, à cause du double-seuil du cours.
- un coéquipier suit cinq (!) cours en même temps et n'a pas le temps adéquat pour travailler correctement dans LOG210.
- plusieurs coéquipiers sont « expérimentés » mais ils ont de la difficulté à s'entendre sur la direction du projet.
- l'équipe n'est pas cohésive ; chacun fait avancer sa partie, mais le code ne fonctionne pas ensemble.

Vous devez en parler avec votre équipe. Si la situation ne s'améliore pas, vous pouvez en parler avec les chargés de laboratoire et l'enseignant.

Pour mieux évaluer le travail de chacun dans l'équipe au laboratoire, il y a des conseils dans la section [Contributions de l'équipe](#).

12 Outils pour la modélisation UML

Le chapitre F2o/A22  définit quelques termes importants pour la modélisation avec UML et les outils.

En mode esquisse, lorsqu'on dessine un modèle sur un tableau blanc ou un papier, un outil pratique pour numériser le tout est **Office Lens** ([Android](#) ou [iOS](#)). Les filtres pour supprimer les reflets sur les tableaux blancs sont impeccables.

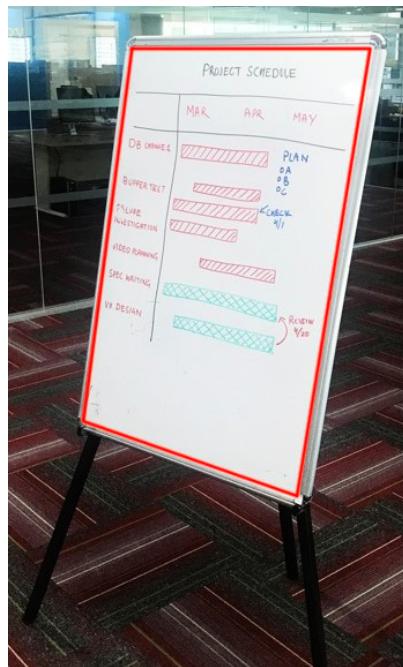


FIGURE 12.1 – Office Lens peut détecter le cadre d'un dessin sur un tableau blanc ou papier et le transformer.

Dans LOG210, on exploite l'outil PlantUML pour faire beaucoup de modèles. C'est un outil qui a plusieurs avantages :

- il est basé sur un langage dédié simple (anglais *domain specific language* ou DSL), dont les fichiers peuvent être facilement mis sur contrôle de version (git);
- il est basé sur du code libre;
- il s'occupe de la mise en page des diagrammes (cela est parfois un inconvénient si un modèle est complexe);
- il est populaire (utilisé par des ingénieurs chez Google pour documenter Android, Pay, etc.);
- il existe plusieurs supports pour les outils de documentation :
 - extension [PlantUML pour VisualStudio Code](#) (figure 12.2) avec [tutoriel YouTube](#);
 - extension [PlantUML Gizmo](#) pour Google Docs et Google Slides, développée en 2014 par le professeur Christopher Fuhrman dans le cadre de son travail à l'ÉTS (figure 12.3)

⚠ Tous les travaux demandés pour les **examens** de LOG210 doivent être faits à *la main*. Pour cette raison, il vaut mieux pratiquer dessiner les modèles en mode esquisse (à la main).

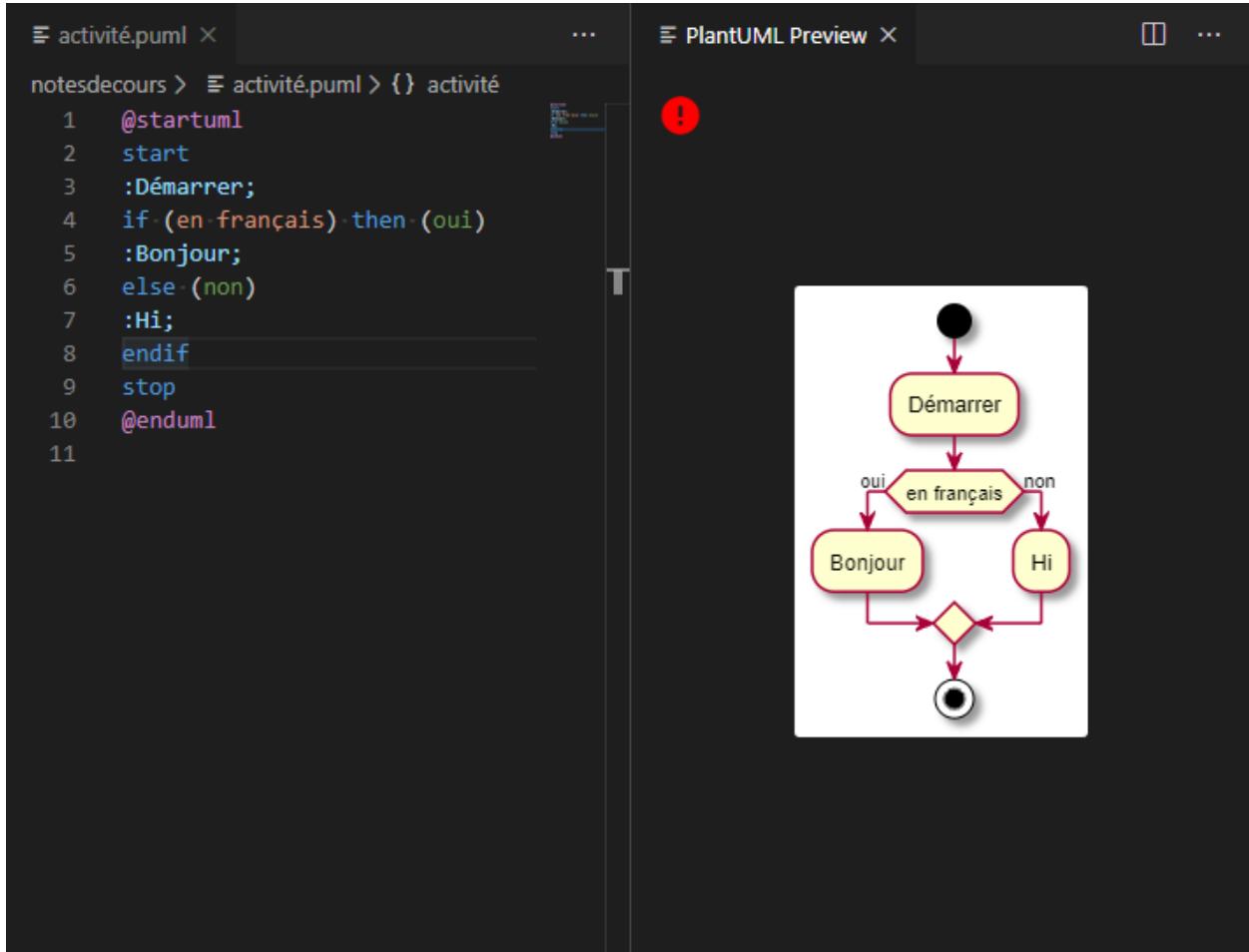


FIGURE 12.2 – L’extension PlantUML pour VisualStudio Code.

Pour un débutant, le langage PlantUML peut sembler plus compliqué qu’utiliser un outil graphique comme Lucidchart. Cependant, pour beaucoup de diagrammes (comme les diagrammes de séquence), ça peut être plus long à créer ou à modifier. Bien que ces outils aient des gabarits ou des modes « UML », ceux-ci ne sont pas toujours conviviaux ou complets. C’est souvent juste des objets groupés et le vrai sens de la notation UML n’est pas considéré (par exemple, une ligne de vie dans un diagramme de séquence est toujours verticale, mais un éditeur graphique quelconque permet de l’orienter dans n’importe quel sens). Ça peut prendre beaucoup de clics pour effectuer une modification et on peut obtenir des résultats intermédiaires qui n’ont aucun sens en UML (voir la figure 12.5). Il est possible de corriger le diagramme, mais en combien de clics ? C’est très vite tannant.

12.1 Exemples de diagramme avec PlantUML pour LOG210

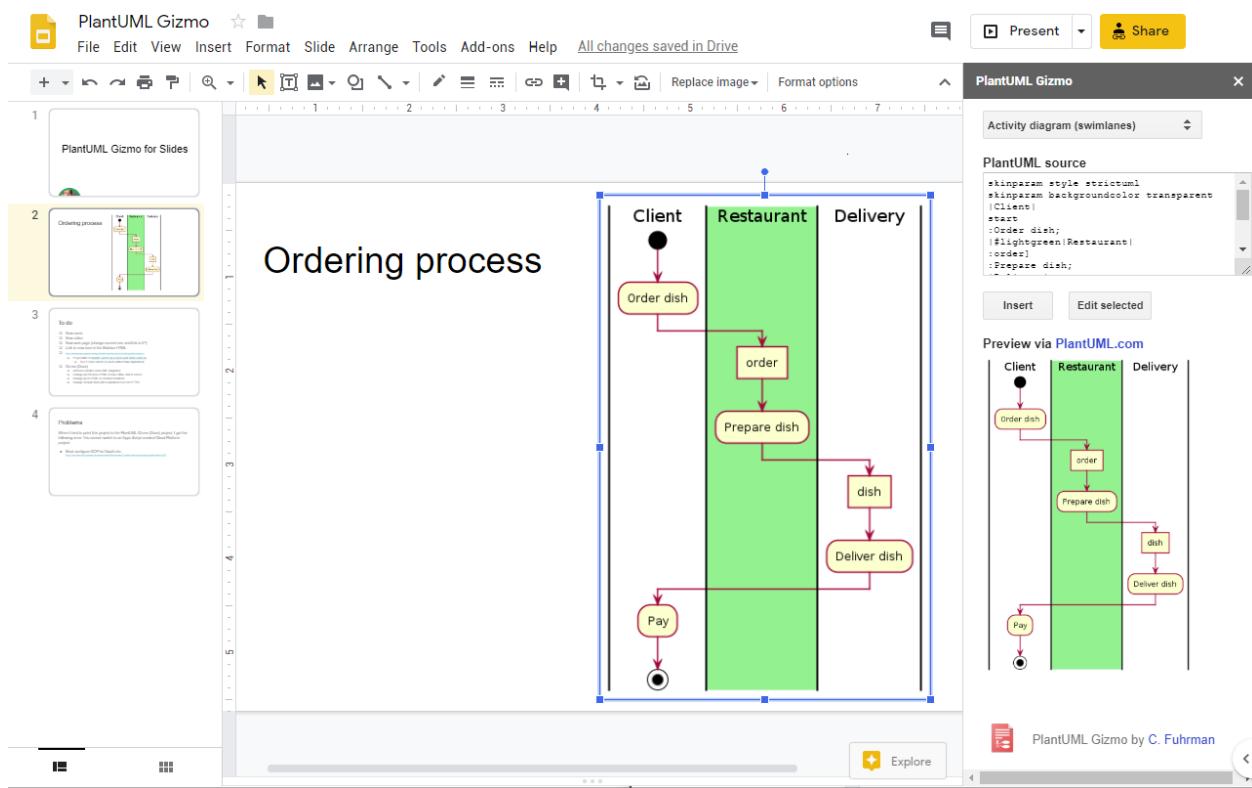


FIGURE 12.3 – PlantUML Gizmo pour Google Docs et Google Slides.

12.1 Exemples de diagramme avec PlantUML pour LOG210

Dans le menu « Select sample diagram » de PlantUML Gizmo (Google Docs), il y a plusieurs exemples de diagrammes utilisés dans le cadre de LOG210 (voir la figure 12.4).

12.2 Astuces PlantUML

- [Comment intégrer PlantUML dans le `Readme.md` de GitHub/GitLab?](#)
- Le serveur de PlantUML.com génère un diagramme à partir d'un URL :
<https://plantuml.com/plantuml/{forme}/{clé}> qui contient une clé comme
Syp9J4vLqBLJSCffib9mB2t9ICqhoKnEBCdCprC8IYqiJIqkuGBAAUW2rJY256DHLLoGdrUS2W00
 - La clé est en fait une représentation compressée du code source.
- On peut changer la forme du diagramme en changeant la partie `{forme}` de l'URL :
 - `{forme}` → `png`, `img` ou `svg` : représentation graphique correspondante ;
 - `{forme}` → `uml` : récupération du code source PlantUML (ça marche avec `http:` seulement) ;
 - On peut également récupérer le code source d'un URL avec l'outil PlantUML localement avec l'option `-decodeurl {clé}` de la ligne de commande :

```
$ java -jar plantuml.jar -decodeurl  
Syp9J4vLqBLJSCfFib9mB2t9ICqhoKnEBCdCprC8IYqiJIqkuGBAAUW2rJY256DHLLoGdrUS2W  
  
@startuml  
Alice -> Bob: Authentication Request  
Bob --> Alice: Authentication Response  
@enduml
```

- Les images png générées par le serveur ou par l'outil contiennent une copie du code source dans les meta-données PNG.
- On peut [réécupérer le code source PlantUML à partir d'une image PNG](#) avec un outil sur le Web comme [ceci](#).
- On peut également utiliser l'option `-metadata` de la ligne de commande PlantUML :

```
$ java -jar plantuml.jar -metadata diagram.png > diagram.puml
```

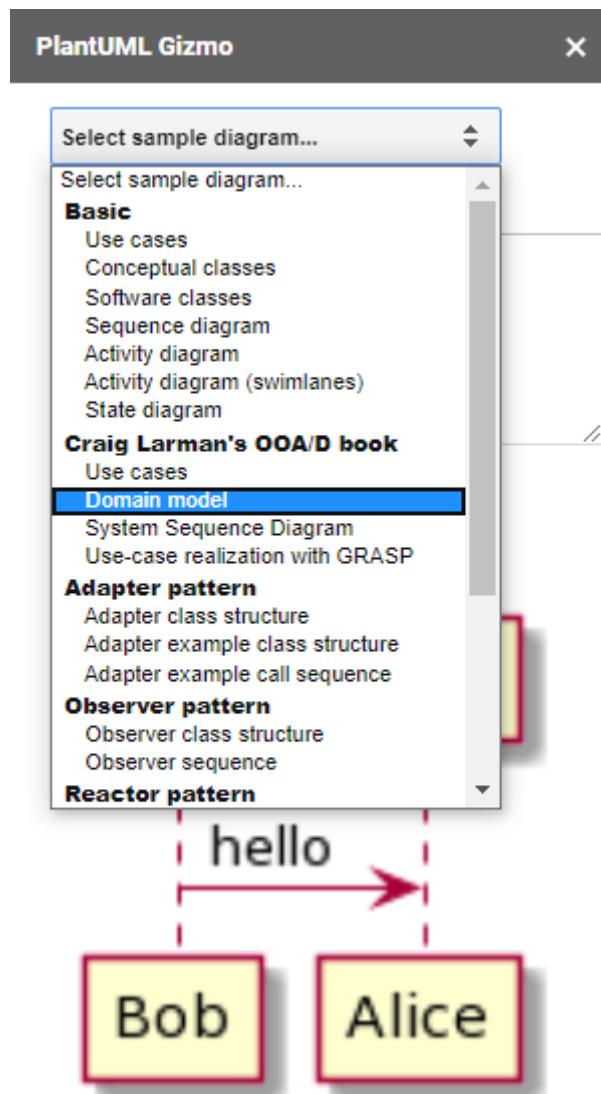


FIGURE 12.4 – PlantUML Gizmo offre plusieurs exemples de diagramme UML.

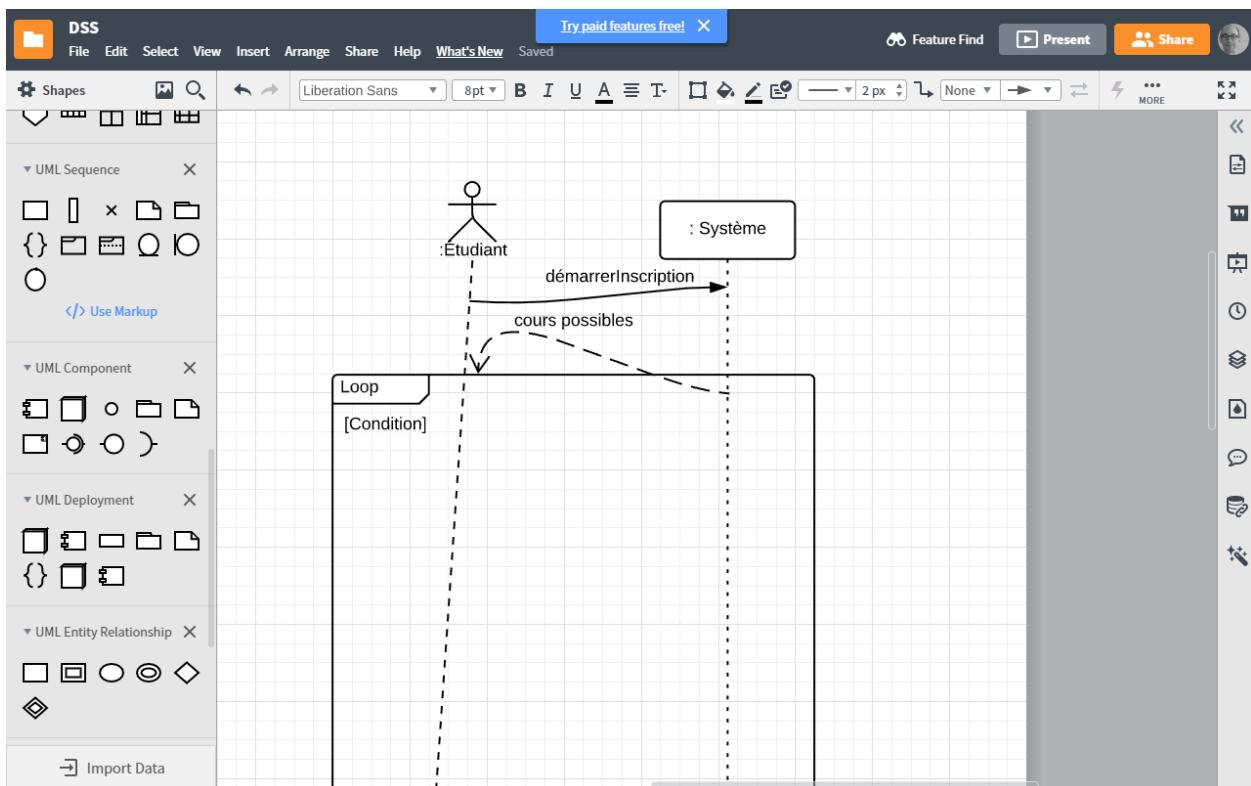


FIGURE 12.5 – Exemple de tentative de créer un diagramme de séquence système (DSS) avec Lucidcharts. C'est principalement un éditeur graphique avec les éléments graphiques UML qui sont essentiellement des éléments graphiques composés. Il n'y a pas de sémantique UML dans l'outil. Par exemple, un « messages » dans Lucidcharts est juste une ligne groupée avec un texte. Elle peut se coller dynamiquement à d'autres éléments en se transformant en courbe (!) lorsque vous déplacez un bloc « loop ». La ligne de vie de l'acteur Étudiant se transforme en diagonale lorsque l'acteur est déplacé à droite. Un vrai message UML est normalement toujours à l'horizontale et une vraie ligne de vie est toujours à la verticale. Puisque Lucidcharts ne connaît pas cette sémantique, vous risquez de perdre beaucoup de temps à faire des diagrammes UML avec ce genre d'outil.

13 Décortiquer les patterns GoF avec GRASP

Craig Larman a proposé les GRASP pour faciliter la compréhension des forces essentielles de la conception orientée-objet. Dans ce chapitre, on examine la présence des GRASP dans les patterns GoF. C'est une excellente façon de mieux comprendre et les principes GRASP et les patterns GoF.

13.1 Exemple avec Adaptateur

Le chapitre A26/F23 du livre du cours présente l'exemple du pattern Adaptateur pour les calculateurs de taxes (figure 13.1 tirée du livre de Larman, Figure A26.3/F23.3).

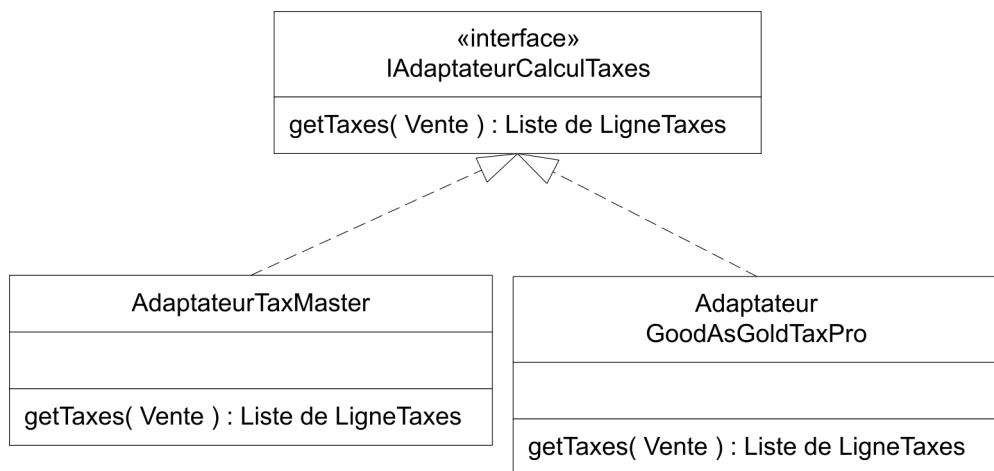


FIGURE 13.1 – Le pattern Adaptateur.

13.2 Imaginer le code sans le pattern GoF

Chaque principe GRASP est défini avec un énoncé d'un problème de conception et une solution pour le résoudre. Pourtant, beaucoup d'exemples dans le livre de LOG210 sont des patterns déjà appliqués (et le problème initial n'est pas toujours expliqué en détail).

Alors, pour mieux comprendre l'application des patterns GoF, on doit imaginer la situation du logiciel *avant* l'application du pattern. Dans l'exemple avec l'adaptateur pour les calculateurs de taxes, imaginez le code si on n'avait aucun adaptateur. À la place d'une méthode `getTaxes()` envoyée par la classe `Vente` à l'adaptateur, on serait obligé de faire un branchement selon le type de calculateur de taxes externe utilisé actuellement (si on veut supporter plusieurs calculateurs). Donc, dans la classe `Vente`, il y aurait du code comme ceci :

```

/* calculateurTaxes est le nom du calculateur utilisé actuellement
 */
if (calculateurTaxes == "GoodAsGoldTaxPro") {
    /* série d'instructions pour intéragir avec le calculateur */
} else if (calculateurTaxes == "TaxMaster") {
    /* série d'instructions pour intéragir avec le calculateur */
} else if /* ainsi de suite pour chacun des calculateurs */
    /* ... */
}

```

Pour supporter un nouveau calculateur de taxes, il faudrait coder une nouvelle branche dans le bloc de `if/then`. Ça nuirait à la lisibilité du code et la méthode qui contient tout ce code deviendrait de plus en plus longue. Même si on faisait une méthode pour encapsuler le code de chaque branche, ça ferait toujours augmenter les responsabilités de la classe Vente. Elle est responsable de connaître tous les détails (l'API distinct et immuable) de chaque calculateur de taxe externe, puisqu'elle communique directement (il y a du couplage) à ces derniers.

Le pattern Adaptateur comprend les principes GRASP Faible couplage, Forte cohésion, Polymorphisme, Indirection, Fabrication pure et Protection des variations. La figure 13.2 (tirée du livre de Larman, Figure A26.3/F23.3) démontre la relation entre ces principes dans le cas d'Adaptateur.

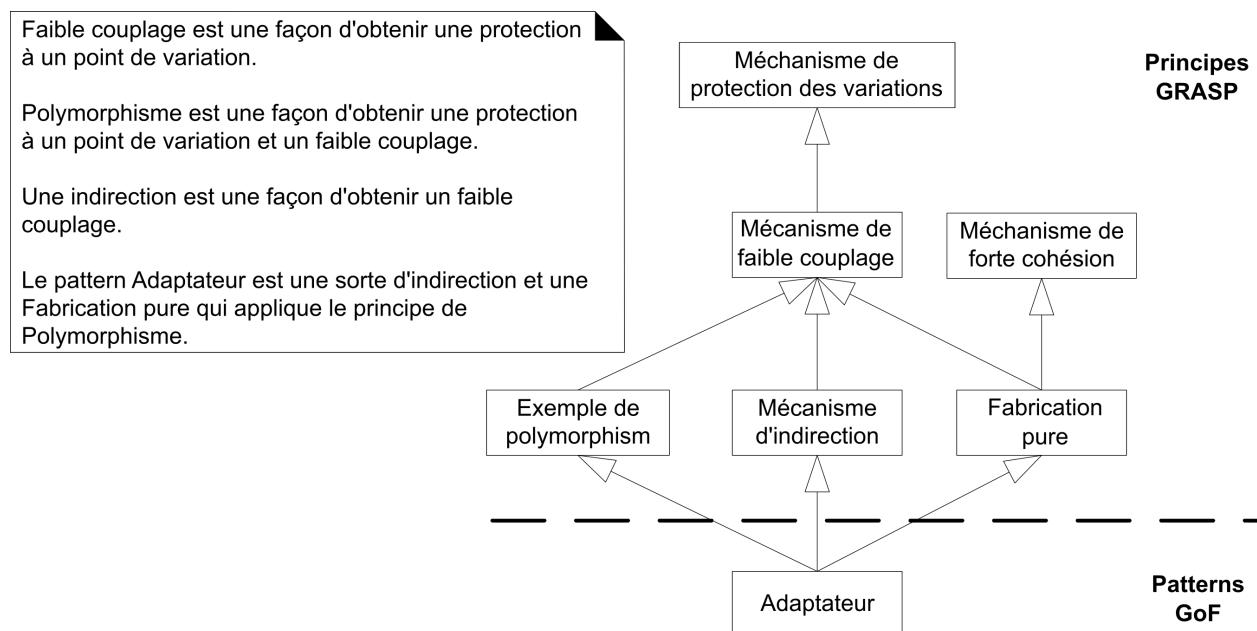


FIGURE 13.2 – Adaptateur et principes GRASP.

On peut donc voir le pattern adaptateur comme *une spécialisation* de plusieurs principes GRASP :

- Polymorphisme
- Indirection
- Fabrication pure
- Faible couplage
- Forte cohésion
- Protection des variations

Êtes-vous en mesure d'expliquer dans ce contexte comment Adaptateur est relié à ces principes ? C'est-à-dire, pouvez-vous identifier les GRASP dans le pattern Adaptateur ?

13.3 Identifier les GRASP dans les GoF

Pour identifier les principes GRASP dans un pattern GoF comme Adaptateur, on rappelle la définition de chaque principe GRASP et on essaie d'imaginer le problème qui pourrait exister et comment le principe (et le pattern GoF) résout le problème.

Référez-vous à la figure 13.1 du pattern Adaptateur pour les sections suivantes.

13.3.1 Polymorphisme

Problème : Qui est responsable quand le comportement varie selon le type ?

Solution : 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.

Le « comportement qui varie » est la manière d'adapter les méthodes utilisées par le calculateur de taxes choisi à la méthode `getTaxes()`. Alors, cette « responsabilité » est affectée au type interface `IAdaptateurCalculTaxes` (et ces implémentations) dans l'opération polymorphe `getTaxes()`.

13.3.2 Fabrication pure

Problème : En cas de situation désespérée, que faire quand vous ne voulez pas transgresser les principes de faible couplage et de forte cohésion ?

Solution : 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.

La Fabrication pure est la classe « comportementale et artificielle » qui est la hiérarchie `IAdaptateurCalculTaxes` (comprenant chaque adaptateur concret). Elle est comportementale puisqu'elle ne fait qu'adapter des appels. Elle est artificielle puisqu'elle ne représente pas un élément dans le modèle du domaine.

L'ensemble des adaptateurs concrets ont des « responsabilités cohésives » qui sont la manière d'adapter la méthode `getTaxes()` aux méthodes (immuables) des calculateurs de taxes externes. Elles ne font que ça. La cohésion est augmentée aussi dans la classe Vente qui n'a plus la responsabilité de s'adapter aux calculateurs de taxes externes. C'est le travail qui a été donné aux adaptateurs concrets.

Le couplage est diminué, car la classe Vente n'est plus couplée directement aux calculateurs de taxes externes. La réutilisation des calculateurs est facilitée, car la classe Vente ne doit plus être modifiée si l'on veut utiliser un autre calculateur externe. Il suffit de créer un adaptateur pour ce dernier.

13.3.3 Indirection

Problème : Comment affecter les responsabilités pour éviter le couplage direct ?

Solution : Pour éviter le couplage direct, affectez la responsabilité à un objet qui sert d'intermédiaire avec les autres composants ou services.

Le « couplage direct » qui est évité est le couplage entre la classe Vente et les calculateurs de taxes externes. Le pattern Adaptateur (général) cherche à découpler le Client des classes nommées Adaptee, car chaque Adaptee a une API différente pour le même genre de « service ». Alors, la responsabilité de s'adapter aux services différents est affectée à la hiérarchie de « classes intermédiaires », soit l'interface type IAdaptateurCalculTaxes et ses implémentations.

13.3.4 Protection des variations

Problème : 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 ?

Solution : Identifiez les points de variation ou d'instabilité prévisibles et affectez les responsabilités afin de créer une « interface » stable autour d'eux.

Les « variations ou l'instabilité » sont les calculateurs de taxes qui ne sont pas sous le contrôle des développeurs du projet (ce sont des modules externes ayant chacun une API différente). Quant à « l'impact négatif sur les autres », il s'agit des modifications que les développeurs auraient à faire sur la classe Vente chaque fois que l'on décide de supporter un autre calculateur de taxes (ou si l'API de ce dernier évolue).

Quant aux « responsabilités » à affecter, c'est la fonctionnalité commune de tous les calculateurs de taxes, soit le calcul de taxes. Pour ce qui est de « l'interface stable », il s'agit de la méthode `getTaxes()` qui ne changera jamais. Elle est définie dans le type-interface IAdaptateurCalculTaxes. Cette définition isole (protège) la classe Vente des modifications (ajout de nouveaux calculateurs ou changements de leur API).

13.4 GRASP et réusinage

Il y a des liens entre les GRASP et les activités de réusinage. Alors, un IDE qui automatise les refactorings peut vous aider à appliquer les GRASP :

- GRASP Polymorphisme est relié à [Replace Type Code with Subclasses](#) et [Replace conditional with polymorphism](#) – attention, il vaut mieux appliquer ce dernier seulement quand il y a des instructions conditionnelles (`switch`) répétées à plusieurs endroits dans le code.
- GRASP Fabrication pure est relié à [Extract class](#).
- GRASP Indirection est relié à [Extract function](#) et [Move function](#).

14 Fiabilité

Le chapitre A35/F30 du livre du cours présente le problème de la fiabilité pour le système NextGen POS. C'est le basculement sur un service local en cas d'échec d'un service distant.

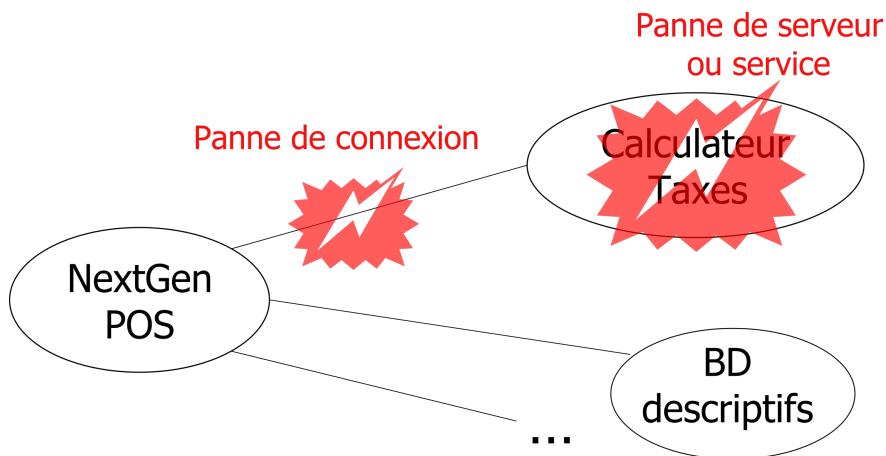


FIGURE 14.1 – Comment tolérer une panne de connexion ou de service ?

Voici les points importants :

- Définitions des termes, A35.3/F30.3 :
 - **Faute.** La cause première du problème
 - **Erreur.** La manifestation de la faute lors de l'exécution. Les erreurs sont détectées (ou non).
 - **Échec.** Déni de service causé par une erreur.
- Les solutions proposées par l'architecte système et documentées par Larman impliquent les concepts suivants :
 - Mise en cache locale d'informations recherchées au service distant, A35.2/F30.2
 - Utilisation d'*Adaptateur [GoF]* pour réaliser le service redondant (lecture d'information), A35.2/F30.2
 - Réalisation d'un scénario dans le cas d'utilisation pour supporter l'échec de tout (rien ne va plus) en permettant au Caissier de saisir l'information (description et prix), A35.3/F30.3. Dans ce cas, il faut bien gérer les exceptions.
 - Utilisation de *Procuration (Proxy) de redirection [GoF]* pour basculer sur un service local en cas de panne (écriture d'information), A35.4/F30.4

Faire une conception pour la fiabilité nécessite de l'expérience (ou l'utilisation des patterns). Un bon livre est celui de R. Hanmer (Hanmer, 2007).

L'utilisation de services dans le nuage (infonuagique) amène une redondance de serveurs. Cependant, même un service web a besoin de **redondance dans les zones géographiques**, car une erreur de configuration ou une crise régionale (ouragan, tremblement de terre) pourrait affecter toute une grappe de serveurs.

15 Diagrammes d'état

Ce chapitre contient des informations sur les diagrammes d'état en UML. Le sujet est traité dans le chapitre A29/F25  du livre du cours.

Il s'agit de la modélisation. Un état est une simplification de la réalité de quelque chose qui évolue dans le temps.

Les points importants :

- Un diagramme d'état sert à modéliser les comportements. Un concept préalable :  Automate fini
- Un diagramme d'état contient des éléments suivants :
 - Événement
 - occurrence d'un fait significatif ou remarquable
 - État
 - la condition d'un objet à un moment donné, jusqu'à l'arrivée d'un nouvel événement
 - Transition
 - relation état-événement-état
 - indique que l'objet change d'état
- La différence entre les objets
 - Un objet répondant de la même manière à un événement donné est un objet *état-indépendant* (par rapport à l'événement)
 - Un objet répondant différemment, selon son état, à un événement donné est un objet état-dépendant
- Les transitions peuvent avoir des *actions* et des *conditions de garde*
- Dans la notation, il y a également la possibilité de faire les *états imbriqués*

La figure 15.1 est un exemple tiré du livre du cours et fait en PlantUML.

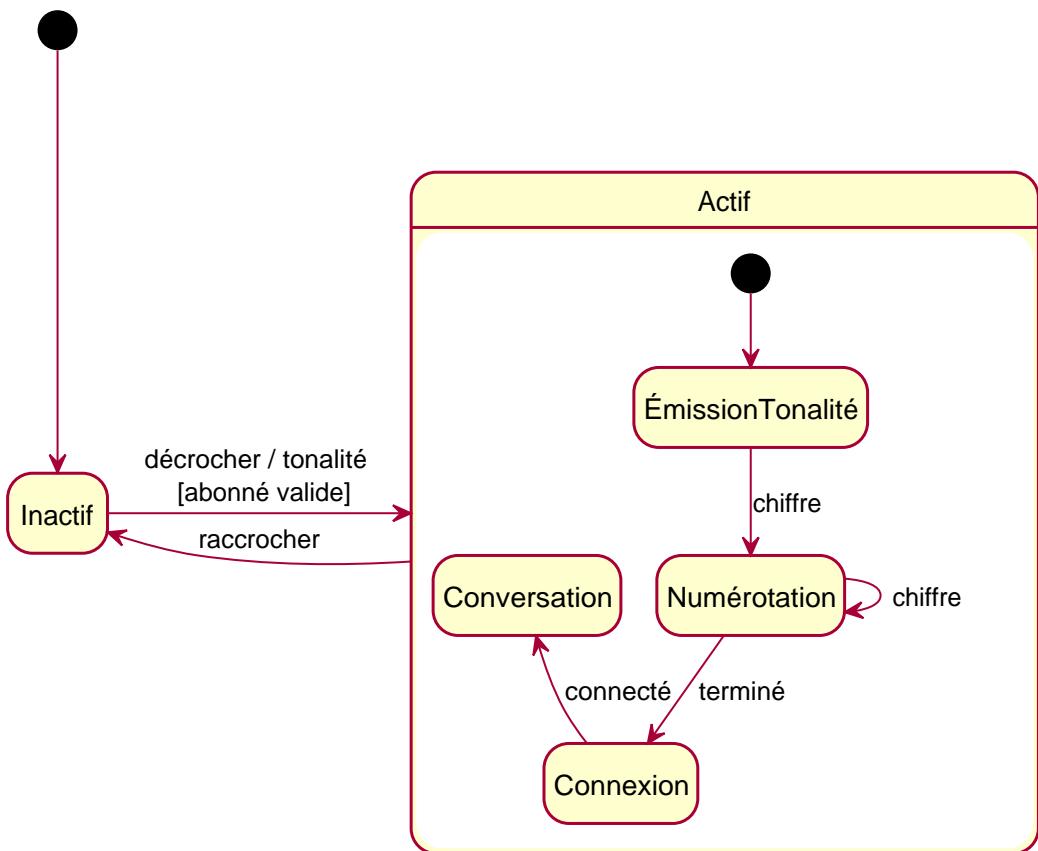


FIGURE 15.1 – Diagramme d’états (figure A29.3/F25.10 du livre). ([PlantUML](#))

16 Diagrammes d'activités

Ce chapitre contient des informations sur les diagrammes d'activités en UML. Les détails se trouvent dans le chapitre F25/A28  du livre du cours.

Les diagrammes d'activités servent à modéliser des processus d'affaires (de métier), des enchaînements d'activités (workflows), des flots de données et des algorithmes complexes.

Voici les éléments importants :

- début et fin (activité)
- partition
- action
- nœud d'objet
- débranchement et jointure (parallélisme)
- décision et fusion (exclusion mutuelle)

Pour la modélisation de flot de données, il existe une notation pour les [diagrammes de flots de données \(DFD\)](#)  . Il ne s'agit pas de l'UML, mais cette notation est encore utilisée (depuis les années 1970).

Un exemple de diagramme d'activité utilisé dans le cadre du cours de LOG210 est dans la figure 16.1. Ce diagramme qui explique comment GitHub Classrooms permet à l'étudiant qui accepte un devoir (*assignment* en anglais) sur GitHub Classrooms de choisir son identité universitaire, mais seulement si l'enseignant a téléversé la liste de classe *avant* d'envoyer les invitations aux étudiants.

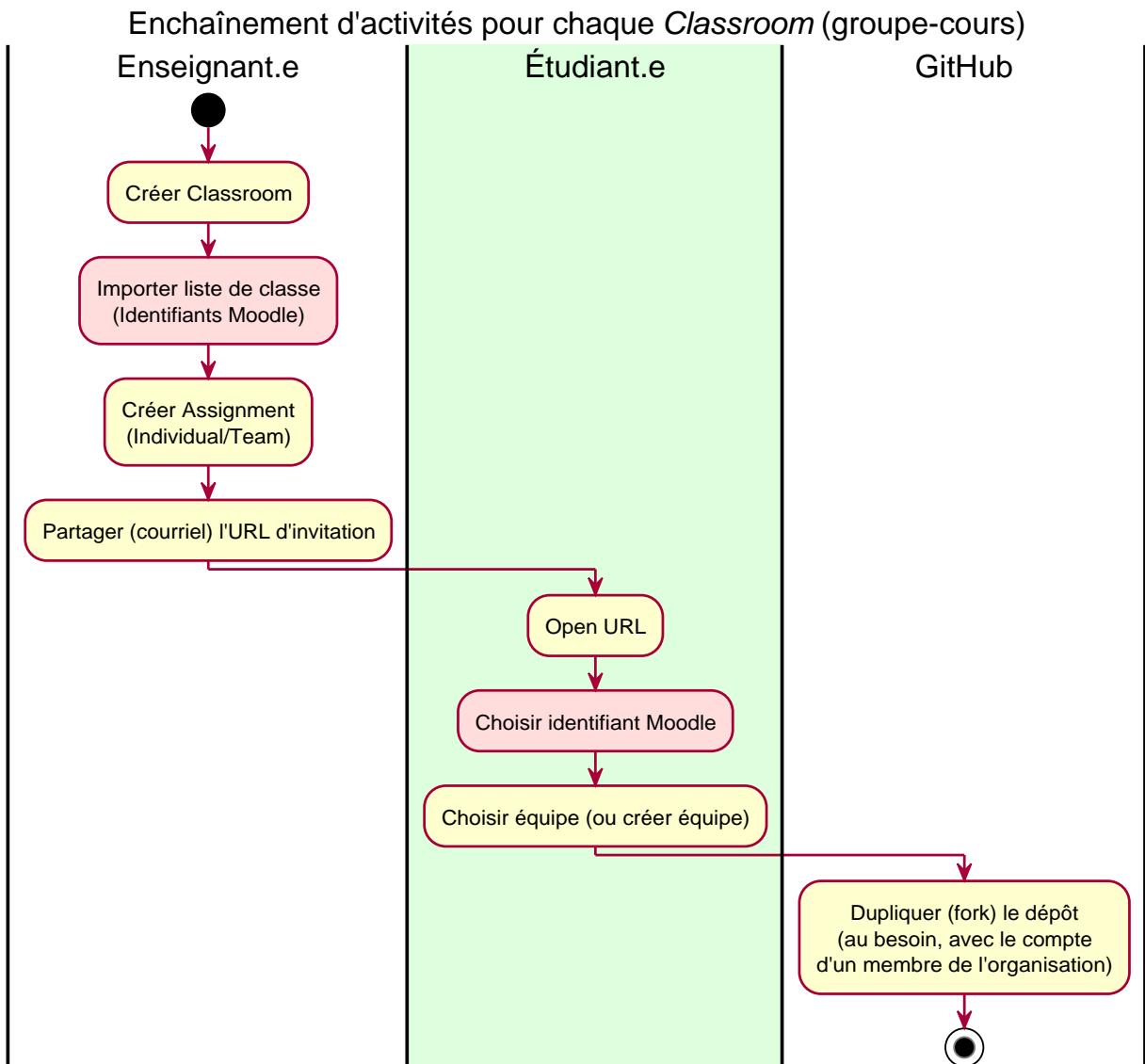


FIGURE 16.1 – Diagramme d'activités pour les activités séquentielles de GitHub Classrooms. ([PlantUML](#))

17 Conception de packages

Le chapitre A13/F12  du livre du cours contient des directives pour la conception de packages.

Les points importants sont les suivants (les détails se trouvent dans le livre) :

- La notation UML des diagrammes de package
- Organiser les packages par **cohésion**
- Organiser les packages une **famille d'interface** (convention Java)
- Créer un package par **tâche** et par **groupe de classes instables** (Branches)
- Les packages les plus responsables sont les plus stables
- Factoriser les types indépendants
- Utiliser **fabrique** (factory) pour limiter la dépendance aux packages concrets
- Comment rompre les cycles dans les packages

La figure 17.1 est un exemple d'un diagramme de package.

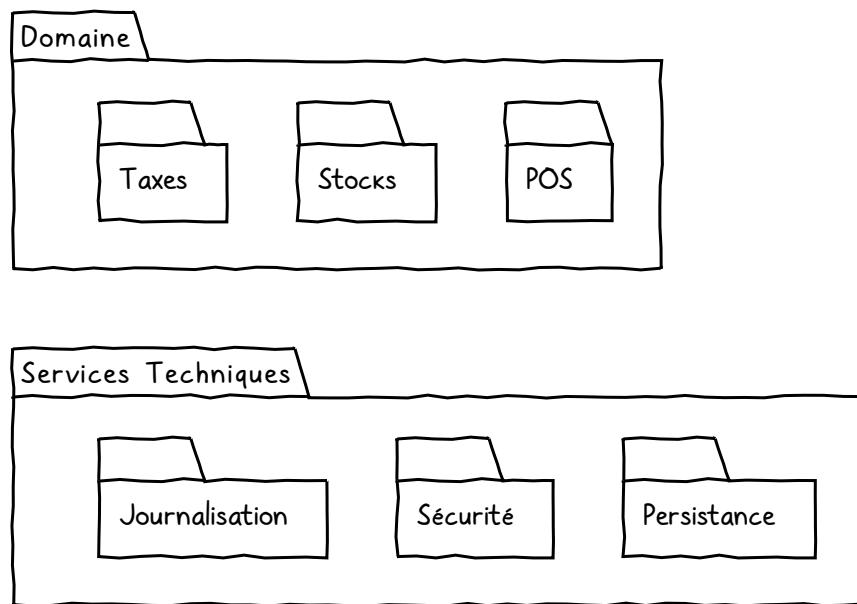


FIGURE 17.1 – Diagramme de packages (tiré de la figure F12.6  du livre du cours). ([PlantUML](#))

18 Dette technique

Ce chapitre contient des informations sur le concept de la [dette technique](#) W, qui n'est pas un sujet du livre du cours.

Pour rajouter une nouvelle fonctionnalité à un système, les développeurs ont souvent un choix entre deux façons de procéder. La première est facile à mettre en place (le « Hacking cowboy » sur [Spectre de la conception](#)), mais elle est souvent désordonnée et rendra sûrement plus difficiles des changements au système dans le futur. L'autre est une solution élégante et donc plus difficile à rendre opérationnelle, mais elle facilitera des modifications à venir. Comment prendre la décision ? La *dette technique* est une métaphore pour aider à comprendre des conséquences à long terme pour des choix de conception permettant de livrer une fonctionnalité à court terme.

La dette est une forme de risque qui peut apporter des bénéfices ou des pertes. Tout dépend de la quantité d'intérêt à payer. L'inventeur du wiki, Ward Cunningham (réf), a utilisé la métaphore de la dette dans un projet de développement de logiciel de gestion de portefeuille réalisé dans une variante du langage Smalltalk :

Un autre piège plus sérieux est l'échec à consolider [un design]. Bien que le code non raffiné puisse fonctionner correctement et être totalement acceptable pour le client, des quantités excessives de ce genre de code rendront le programme impossible à maîtriser, ce qui entraînera une surspécialisation des programmeurs et, finalement, un produit inflexible. Livrer du code non raffiné équivaut à s'endetter. Une petite dette accélère le développement tant qu'elle est remboursée rapidement avec une réécriture. [Le paradigme des] objets rendent le coût de cette transaction tolérable. Le danger survient lorsque la dette n'est pas remboursée. Chaque minute passée sur un code qui n'est pas tout à fait correct compte comme un intérêt sur cette dette. Des organisations entières peuvent être bloquées par l'endettement d'une implémentation non consolidée, orientée objet ou autre.

Comme c'est une métaphore puissante, beaucoup de développeurs l'utilisent et c'est un terme avec une certaine popularité, comme on peut voir à la figure 18.1. Dans une vidéo plus récente, Cunningham a rappelé que la notion originale de la métaphore s'inspire du code qui est incohérent par rapport à un problème complexe plutôt que du code simplement « mal écrit » :

L'explication que j'ai donnée à mon patron, et c'était un logiciel financier, était une analogie financière que j'ai appelée « la métaphore de la dette ». Et cela veut dire que si nous ne parvenions pas à aligner notre programme sur ce que nous considérions alors comme la bonne façon de penser à nos objets financiers, alors nous allions continuellement trébucher sur ce désaccord et cela nous ralentirait, comme payer des intérêts sur un prêt.

[...]

Beaucoup de gens (au moins des blogueurs) ont expliqué la métaphore de la dette et l'ont confondue, je pense, avec l'idée que vous pourriez écrire mal le code avec l'intention de faire du bon travail plus tard et de penser que c'était la principale source de dette. Je ne suis jamais favorable à l'écriture médiocre du code, mais je suis en faveur de l'écriture de code pour refléter votre compréhension actuelle d'un problème, même si cette compréhension est partielle.

Fowler a également abordé le sujet de la dette, notamment à propos de la distinction entre du code « mal écrit » et les compromis de conception faits avec une intention d'accélérer le développement :

Je pense que la métaphore de la dette fonctionne bien dans les deux cas - la différence est dans la nature de la dette. Le code mal écrit est une dette imprudente qui se traduit par des paiements d'intérêts paralysants ou une longue période de remboursement du principal. Il y a quelques projets où nous avons pris en charge une base de code avec une dette élevée et avons trouvé la métaphore très utile pour discuter avec la direction de notre client de comment l'aborder.

La métaphore de la dette nous rappelle les choix que nous pouvons faire avec les anomalies de conception. La dette prudente qui a permis de compléter une version du logiciel ne vaut peut-être pas la peine d'être remboursée si les paiements d'intérêts sont suffisamment faibles, par exemple si les anomalies sont dans une partie rarement touchée de la base de code.

La distinction utile n'est donc pas entre dette ou non-dette, mais entre **dette prudente et imprudente**.

[...] Il y a aussi une différence entre la **dette délibérée et involontaire**. L'exemple de la dette prudente est délibéré parce que l'équipe sait qu'elle s'endette et réfléchit donc à la question de savoir si le bénéfice de livrer plus tôt une version du logiciel est supérieur au coût de son remboursement. Une équipe ignorante des pratiques de conception prend sa dette imprudente sans même constater à quel point elle s'endette.

La dette imprudente pourrait aussi être délibérée. Une équipe peut connaître les bonnes pratiques de conception, voire être capable de les mettre en pratique, mais décide finalement d'aller « à la va-vite » parce qu'elle pense qu'elle ne peut pas se permettre le temps nécessaire pour écrire du code propre.

La dette peut être classifiée dans un quadrant comme dans le tableau 18.1 proposé par Fowler. Selon lui, la dette dont Ward Cunningham a parlé dans sa vidéo peut être classifiée comme « prudente et involontaire ». Fowler remarque que selon son expérience, la dette « imprudente et délibérée » est rarement rentable.

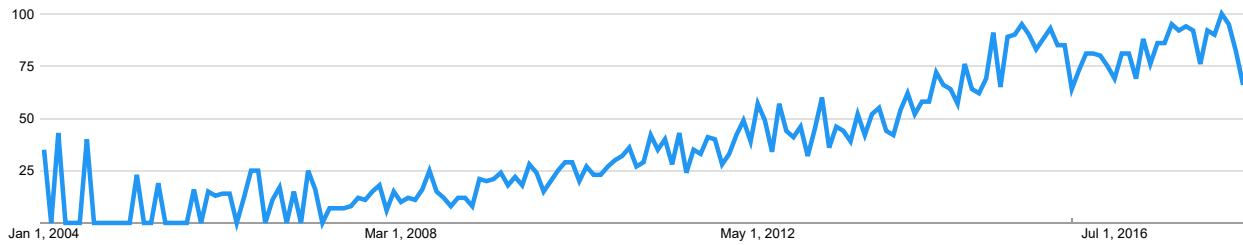


FIGURE 18.1 – Tendances Google (trends.google.com) pour le terme « dette technique » (anglais *technical debt*)

Tableau 18.1 – Classification de la dette selon Fowler (2009)

Dette	Imprudente	Prudente
Délibérée	<p><i>On n'a pas le temps pour la conception !</i> Cette forme de dette est rarement rentable.</p>	<p><i>Il faut livrer maintenant puis en assumer les conséquences.</i> Exemple : La dette est due à une partie limitée du code.</p>
Involontaire	<p><i>C'est quoi la séparation en couches ?</i> Il s'agit de l'ignorance de bonnes pratiques.</p>	<p><i>Maintenant on sait comment on aurait dû le faire.</i> C'est tenter une solution malgré une compréhension limitée du problème.</p>

19 Diagrammes de déploiement et de composants

Ce chapitre  contient des informations sur les diagrammes de déploiement et de composants en UML. Les détails se trouvent dans le chapitre F31/A37  du livre du cours.

19.1 Diagrammes de déploiement

Un diagramme de déploiement présente le déploiement sur l'**architecture physique**. Il sert à documenter :

1. comment les fichiers exécutables seront affectés sur les nœuds de traitement et
2. la communication entre composants physiques

Voici les éléments importants :

- Types de nœuds
 - **Nœud physique (équipement)** : Ressource de traitement physique (ex. de l'électronique numérique), dotée de services de traitement et de mémoire destinés à exécuter un logiciel. Ordinateur classique, cellulaire, etc.
 - **Nœud d'environnement d'exécution (EEN execution environment node)** : Ressource de traitement logiciel qui s'exécute au sein d'un nœud externe (comme un ordinateur) et offrant lui-même un service pour héberger et exécuter d'autres logiciels, par exemple :
 - Système d'exploitation (OS) est un logiciel qui héberge et qui exécute des programmes
 - Machine virtuelle (JVM ou .NET)
 - Moteur de base de données (ex. PostgreSQL) exécute les requêtes SQL
 - Navigateur Web héberge et exécute JavaScript, applets Flash/Java
 - Moteur de workflow
 - Conteneur de servlets ou conteneur d'EJB

La figure 19.1 est un exemple de diagramme de déploiement (laboratoire). La figure 19.2 est un exemple de diagramme de déploiement pour le logiciel iTunes d'Apple.

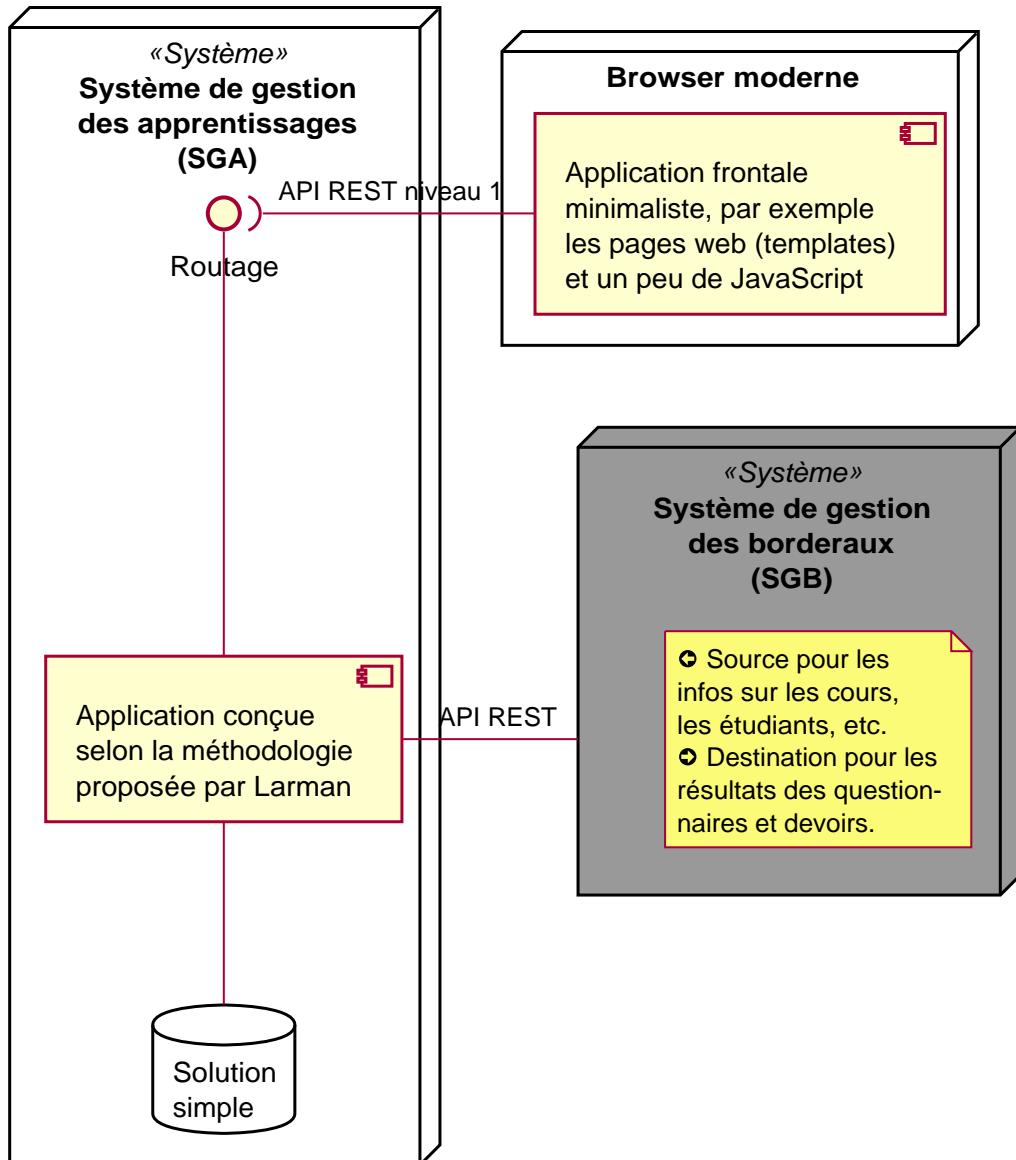
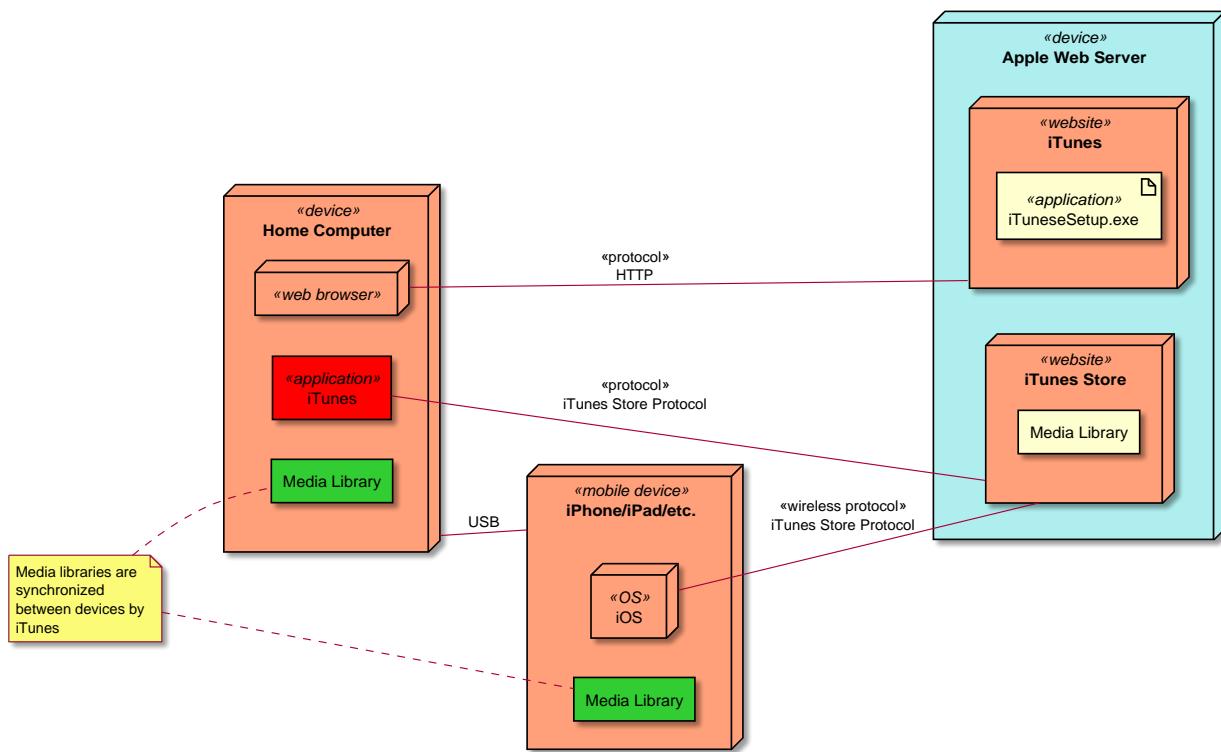


FIGURE 19.1 – Diagramme de déploiement du système à développer pour le laboratoire. ([PlantUML](#))


 FIGURE 19.2 – Diagramme de déploiement pour iTunes d'Apple, inspiré de [ceci](#). (PlantUML)

20 Laboratoires

Ce chapitre contient des informations sur le volet technique des laboratoires.

20.1 JavaScript/TypeScript

Un tutoriel intéressant (et libre) est sur [javascript.info](#). Je vous recommande de contribuer à des [traductions en français sur GitHub](#).

Voici les points qui posent plus de problèmes pour quelqu'un ayant déjà des connaissances en Java :

- Fonctions flèche (*arrow functions* en anglais)
- Traitement asynchrone en JavaScript
 - Promesses et `async/await` : [Tutoriel](#)
- REST (GET vs PUT)
- Environnement de test (Mocha/Chai)
- Les templates PUG (anciennement Jade) : [Tutoriel](#)

Le labo aborde plusieurs de ces aspects, mais certaines notions sont plus complexes. Le but de cette section est de donner des tutoriels plus spécifiques.

Il y a un [dépôt d'exemples avec TypeScript \(utilisant `ts-node` pour les voir facilement\)](#) sur GitHub.

20.2 Contributions de l'équipe

Il existe un outil nommé `gitinspector` qui peut indiquer le niveau d'implication des membres de l'équipe dans un projet sur GitHub. Étant donné que LOG210 utilise un squelette avec les tests, les fichiers `src` de TypeScript, les modèles PlantUML et le `README.md`, il est possible d'utiliser `gitinspector` pour voir des rapports de contribution sur chacun des volets.

Pour faciliter l'utilisation de l'outil, le professeur Fuhrman a créé un [script en bash](#). Voici comment l'utiliser :

- Installer `gitinspector` dans `npm` avec la commande `npm install -g gitinspector`
- Télécharger le script

```
$ git clone https://gist.github.com/fuhrmanator/  
b5b098470e7ec4536c35calce3592853 \  
contributions
```

```
Cloning into 'contributions'...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 10 (delta 3), reused 7 (delta 2), pack-reused 0
Unpacking objects: 100% (10/10), 2.02 KiB | 82.00 KiB/s, done.
```

- Lancer le script sur un dépôt de code source, par exemple sga-equipe-g02-equipe-4 :

```
$ cd contributions
$ ./contributions.sh ../sga-equipe-g02-equipe-4/
gitinspector running on ../sga-equipe-g02-equipe-4/ : patience...
ContributionsÉquipeTest.html
ContributionsÉquipeModèles.html
ContributionsÉquipeDocs.html
ContributionsÉquipeTypeScript.html
ContributionsÉquipeViews.html
```

Les fichiers .html sont créés pour les contributions Test, Modèles, Docs, TypeScript et Views. Chaque rapport indique des contributions selon deux perspectives :

1. Le nombre de soumissions par auteur (activité git)
2. Le nombre de lignes par auteur encore présentes et intactes dans la version HEAD

Vous pouvez voir un exemple du rapport à la Figure 20.1.

20.2.1 Faire le bilan de la contribution de chacun

Après l'évaluation à la fin de chaque itération, il est important de considérer combien chacun a contribué au projet et de valider avec les responsabilités prévues dans le plan de l'itération. Il est normal d'avoir un écart entre le travail prévu et le travail effectué. Un des objectifs du bilan est d'essayer d'expliquer les gros écarts et de corriger ou mitiger les problèmes.

Par exemple, on peut voir à la Figure 20.1 que les deux coéquipiers Anne et Justin ont fait une contribution beaucoup plus importante que les autres coéquipiers Francis et Mélanie. Dans le bilan de l'itération, **on peut indiquer explicitement ce fait, même avec des pourcentages** en évitant d'écrire une phrase vague comme « certains ont travaillé plus que d'autres ». Ensuite, on se pose la question : pourquoi y a-t-il eu cet écart ? Est-ce que Francis et Mélanie sont à l'aise avec les technologies, ont-ils besoin de coaching ? Est-ce que tout le monde est à l'aise avec l'écart ? Est-ce que Anne et Justin ont laissé suffisamment de place aux autres pour faire une contribution ? Est-ce que tout le monde met au moins 6 heures de travail en dehors des séances encadrées ? Est-ce que tout le monde est présent pendant les séances ? Etc.

Dans le bilan il faut *constater les faits et proposer des solutions* pour éviter des écarts importants dans les prochaines itérations. Ainsi, vous gérez les problèmes de manière plus proactive.

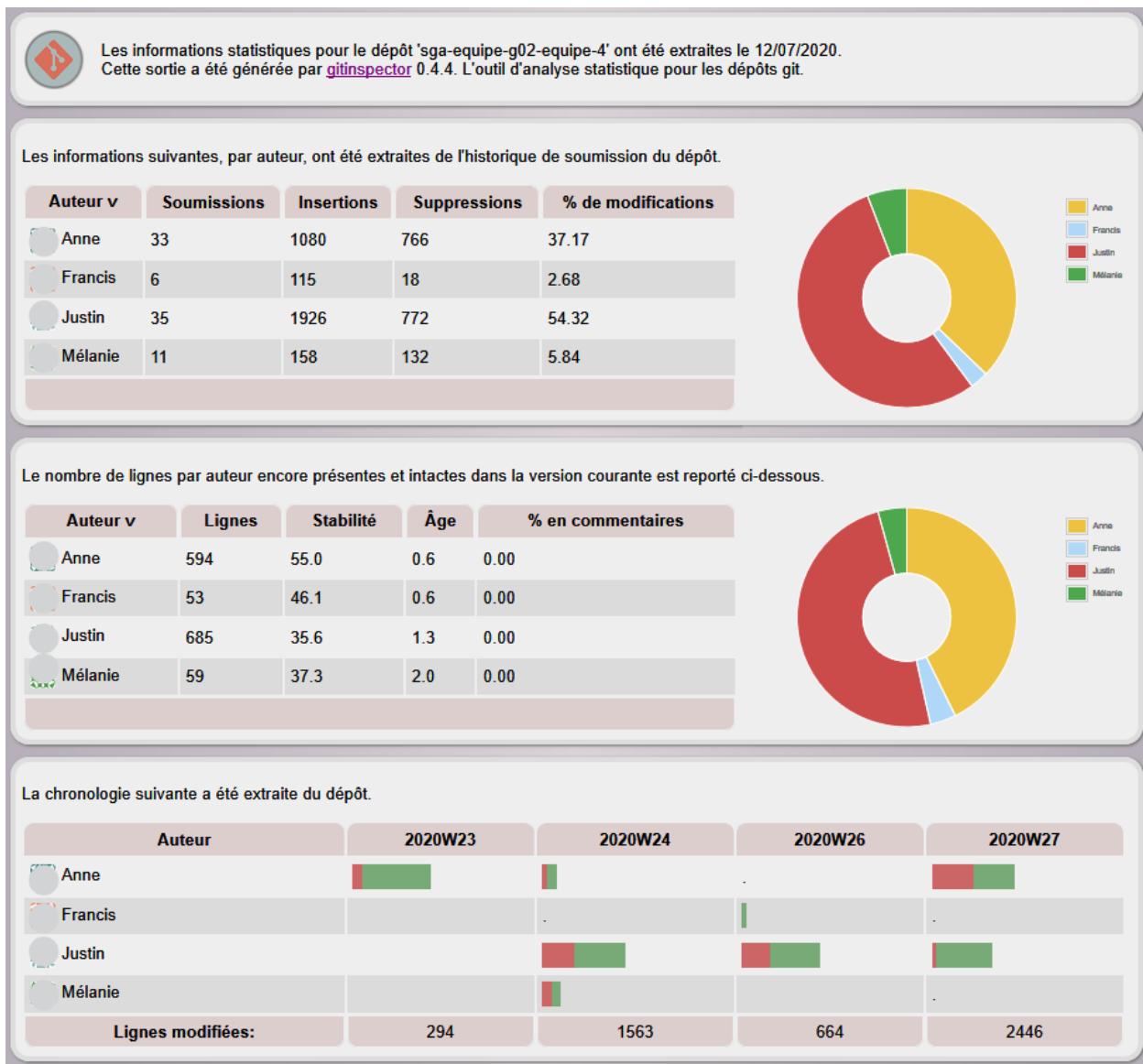


FIGURE 20.1 – Exemple de rapport généré par gitinspector.

20.2.2 FAQ pour gitinspector

Q : Comment fusionner le travail réalisé par le même coéquipier, mais avec plusieurs comptes (courriels) différents ?

R : La solution est avec le fichier `.mailmap`. Vous pouvez rapidement générer un fichier de base avec la commande :

```
git log --pretty="%an %ae" | sort | uniq > .mailmap
```

Ensuite, vous modifiez le fichier `.mailmap` pour que les deux (ou plusieurs) courriels du même auteur aient le même nom. Le nom que vous mettez sera celui qui apparaît dans les rapports la prochaine fois qu'ils seront générés.

Q : Comment exclure le travail réalisé par un chargé de laboratoire (par exemple le clone initial dans GitHub Classroom) ?

R : La solution est d'ajouter le nom de l'auteur dans le tableau du script `contributions.sh` à la ligne suivante avec `authorsToExcludeArray`. Attention :

- Il n'y a pas de `,` entre les éléments des tableaux en bash.
- Le nom d'un auteur ayant un accent ne sera pas reconnu. Il faut changer le nom dans le `.mailmap` pour qu'il n'y ait pas d'accents, ou utiliser une chaîne partielle comme "Benjamin Le" pour exclure les contributions de "Benjamin Le Dû".

```
authorsToExcludeArray=("Benjamin Le" "Yvan Ross")
```

Q : J'ai une autre question...

R : Il y a aussi une [FAQ sur le dépôt de gitinspector](#).

20.3 TODO

- modifier le squelette pour aussi utiliser PUT (REST)
- inclure exemples de `before` et `after` dans les tests, avec une référence à la doc

21 Bibliographie

- Fitzpatrick, B. W., & Collins-Sussman, B. (2012). *Team Geek : A Software Developer's Guide to Working Well with Others* (1 edition). Sebastopol. CA : O'Reilly Media.
- Ford, N. (2009, 24 février). Evolutionary architecture and emergent design : Investigating architecture and design. [CT316]. Repéré à <http://www.ibm.com/developerworks/library/j-eaedi/>
- Hanmer, R. (2007). *Patterns for Fault Tolerant Software* (1^{re} éd.). Chichester, England ; Hoboken, NJ : Wiley.
- Karac, I., & Turhan, B. (2018). What Do We (Really) Know about Test-Driven Development ? *IEEE Software*, 35(4), 81-85. <https://doi.org/10.1109/MS.2018.2801554>
- Larman, C. (2005). *UML 2 et design patterns* (3^e éd.). Paris : Village Mondial.

