

Analyse et conception de logiciels



Christopher Fuhrman

Yvan Ross

2022-05-06

Table des matières

Préface	1
1 Analyse et conception de logiciels	3
1.1 Analyse vs Conception	3
1.2 Décalage des représentations	5
1.3 La complexité et ses sources	5
1.4 Survol de la méthodologie	8
1.5 Développement itératif, évolutif et agile	8
2 Besoins (exigences)	13
2.1 FURPS+	13
3 Cas d'utilisation	17
3.1 Exemple: jeu de Risk	17
4 Modèle du domaine (MDD, modèle conceptuel)	21
4.1 Classes conceptuelles	21
4.2 Attributs	22
4.3 Associations	23
4.4 Exemple de MDD pour le jeu de Risk	24
4.5 Attributs dérivés	26
4.6 Classes de « description » et de catalogues	26
4.7 Classes d'association	27
4.8 Affinement du MDD	28
4.9 FAQ MDD	29
5 Diagrammes de séquence système (DSS)	31
5.1 Exemple: DSS pour Attaquer un pays	31
5.2 Les DSS font abstraction de la couche présentation	31
5.3 FAQ DSS	34
6 Principes GRASP	37
6.1 Spectre de la conception	37
6.2 Tableau des principes GRASP	40
6.3 GRASP et RDCU	41
6.4 GRASP et Patterns GoF	41

Table des matières

7 Dette technique	43
7.1 Origine	43
7.2 Nuances de la dette technique	45
7.3 Résumé	46
8 Contrats d'opération	49
8.1 Qu'est-ce qu'un contrat d'opération	49
8.2 Exemple: Contrats d'opération pour Attaquer un pays	50
8.3 Feuille de référence	51
9 Réalisations de cas d'utilisation (RDCU)	53
9.1 Spécifier le contrôleur	54
9.2 Satisfaire les postconditions	54
9.3 Visibilité	56
9.4 Transformer identifiants en objets	57
9.5 Utilisation de tableau associatif (<code>Map<clé, objet></code>)	58
9.6 RDCU pour l'initialisation, le scénario Démarrer	58
9.7 Réduire le décalage des représentations	59
9.8 Pattern « Faire soi-même »	59
10 Développement piloté par les tests	63
10.1 Kata TDD	65
11 Réusinage (Refactorisation)	69
11.1 Introduction	69
11.2 Symptômes de la mauvaise conception - Code smells	70
11.3 Automatisation du réusinage par les IDE	71
11.4 Impropriété	72
12 Développement de logiciel en équipe	73
12.1 Humilité, Respect, Confiance	74
12.2 Redondance des compétences dans l'équipe (Bus Factor)	76
12.3 Mentorat	77
12.4 Scénarios	78
13 Outils pour la modélisation UML	79
13.1 Exemples de diagramme avec PlantUML	81
13.2 Astuces PlantUML	81
14 Décortiquer les patterns GoF avec GRASP	85
14.1 Exemple avec Adaptateur	85
14.2 Imaginer le code sans le pattern GoF	85
14.3 Identifier les GRASP dans les GoF	87
14.4 GRASP et réusinage	88
15 Fiabilité	89

16 Diagrammes d'activités	91
16.1 Diagrammes de flot de données (DFD)	91
17 Diagrammes d'état	95
18 Conception de packages	97
18.1 Absense de package dans TypeScript	98
19 Diagrammes de déploiement et de composants	101
19.1 Diagrammes de déploiement	101
20 Laboratoires	105
20.1 JavaScript/TypeScript	105
20.2 JavaScript: Truthy et Falsy (conversion en valeur booléenne)	106
20.3 Évaluer les contributions des membres de l'équipe	107
Bibliographie	113

Préface

Ce manuel a été produit par le logiciel Quarto.

Pour en savoir plus sur Quarto, visitez <https://quarto.org/docs/books>.

1 Analyse et conception de logiciels

Ce manuel a été créé dans un contexte de cours d'ingénieur logiciel pour les personnes ayant déjà une expérience en programmation dans un langage orienté objet et une familiarité avec des patrons de conception de la « bande des quatre » (Gang of Four) : Gamma, Helm, Johnson et Vlissides Gamma et al. (1994).

Ce manuel suit la méthodologie d'analyse et de conception proposée par Craig Larman dans son livre « UML 2 et les Design Patterns » Larman (2005). Les références à ce livre sont indiquées par l'icône du livre . Puisque ce livre-là est disponible en français et en anglais et les numéros de chapitres ne sont pas toujours les mêmes, on indique la référence avec F et A pour la langue. Par exemple, **F16.10/A17.10** indique la section **16.10 du livre en français** et la section **17.10 du livre en anglais**. Toutes les références sont données pour la 3^e édition du livre. Si vous avez une autre édition, les chapitres ne sont pas toujours les mêmes et vous devez chercher le sujet dans la table des matières.

Vous trouverez également d'autres sujets importants pour un ingénieur: les notions de complexité, le contexte industriel qui affecte les décisions de conception, l'impact de la conception sur d'autres qualités d'un logiciel, le travail en équipe, etc.

1.1 Analyse vs Conception

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.

Imaginez un jeu qui est joué dans la vraie vie avec deux dés à six faces. Ensuite, on veut construire un logiciel pour ce jeu et donc on peut spécifier la règle du jeu, dont un de nombreux besoins est 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.

Dans l'approche proposée par ce manuel une modélisation orientée objet est utilisée et pour l'analyse (classes conceptuelles décrivant le problème et les besoins comme à la figure 1.1) et pour la conception (classes logicielles proposant une solution dont sa représentation est proche de la modélisation du problème comme à la figure 1.2).

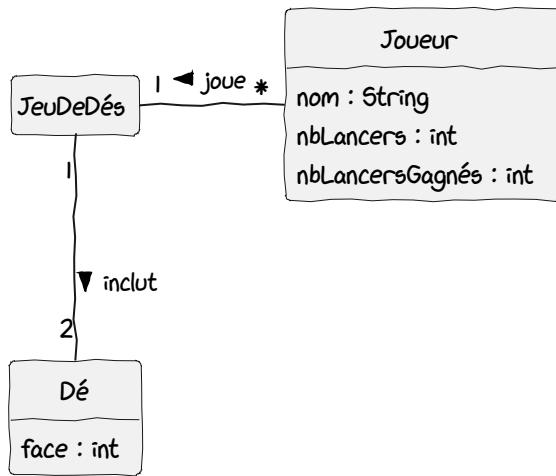


Figure 1.1: Diagramme de *classes conceptuelles* décrivant le *problème* d'un jeu de dés (adapté du Jeu de dés de Larman 2005, chap. 1). Ceci est élaboré lors d'une activité d'analyse.

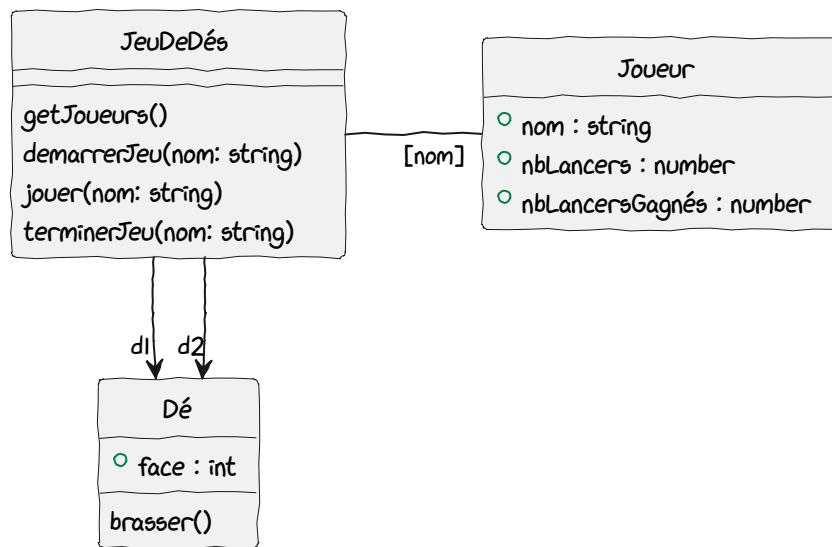


Figure 1.2: Diagramme de *classes logicielles* décrivant une *solution* au problème du jeu de dés. La conception s'inspire du modèle du problème, afin de faciliter sa compréhension.

1.2 Décalage des représentations

Vous avez sûrement remarqué que le modèle du problème (figure 1.1) ressemble beaucoup au modèle de la solution (figure 1.2) pour notre exemple de jeu de dés. Cependant, il y a des différences, car une solution comporte des détails sur la dynamique du jeu qui sera codée. Le modèle du problème et le modèle de la solution ne sont pas identiques.

 Imaginez une autre solution n'ayant qu'une seule classe `Jeu` contenant toute la logique du jeu. Avez-vous déjà codé une solution simple comme ça? C'est un bon design au départ, car c'est simple. Mais au fur et à mesure que vous codez la logique du jeu, bien que ça fonctionne parfaitement, la classe `Jeu` grossit et devient difficile à comprendre.

Une caractéristique souhaitable d'un design est qu'il soit facile à comprendre et à valider par rapport au problème qu'il est censé résoudre. Plus une solution (conception) ressemble à une description (modèle d'analyse) du problème, plus elle est facile à comprendre et à valider. La différence entre la représentation d'un problème et la représentation de sa solution s'appelle le *décalage des représentations*. C'est un terme complexe pour un principe très intuitif. Méfiez-vous des classes importantes dont leur nom est difficile à tracer au problème. Elles vont rendre votre solution plus difficile à comprendre. Pour des explications de Larman, lisez la section 9.3 .

L'exemple du jeu est trivial, puisque le problème est relativement simple. Réduire le décalage des représentations est un principe très important surtout lorsque le problème à résoudre est complexe.

1.3 La complexité et ses sources

Un.e ingénieur.e logiciel est constamment dans une bataille avec un adversaire dont le nom est la complexité. Mais qu'est-ce que la complexité? La figure 1.3 est une image de la complexité. Reconnaissez-vous le domaine d'où vient cette image?

Voici une définition de la complexité:

Complexité: Caractère de ce qui est complexe, difficile à comprendre, de ce qui contient plusieurs éléments.

En voici quelques exemples en développement de logiciels:

- Un *problème* peut être complexe, par exemple le domaine des lois fiscales pour lequel des logiciels existent pour aider les gens à faire des déclarations de revenus.
- Un *projet logiciel* peut être complexe, avec plusieurs packages, chacun ayant beaucoup de classes, etc.
- Un cadre d'applications (cadriel, *framework*) est toujours complexe. Par exemple un framework comme Angular ou React pour développer un *front-end* (application frontale), car l'interaction entre l'utilisateur et une application (possiblement répartie dans le nuage) nécessite beaucoup de fonctionnalités supportées par le cadriel.

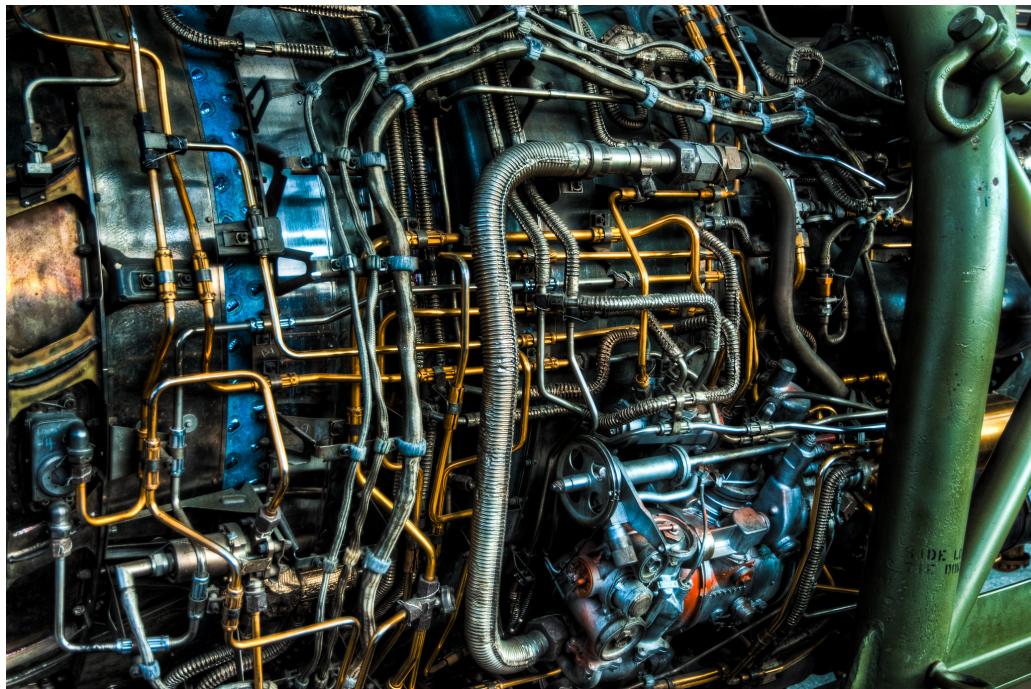


Figure 1.3: « Complexity » (CC BY-SA 2.0) par [lytfyre](#).

- Un *algorithme* peut être complexe, par exemple, l'algorithme de [tri de Shell](#) est plus complexe qu'un simple algorithme de [tri à bulles](#). Notez que la complexité d'un algorithme peut parfois apporter des gains de performance, mais coder, déboguer et maintenir une implémentation d'un algorithme complexe sera plus coûteux.
- Un *patron de conception* peut être complexe, par exemple, les patrons Visiteur, Décorateur, Médiateur, etc. Un patron définit des rôles et parfois des classes et du code supplémentaires à créer. Le tout doit s'intégrer dans un design existant (qui est possiblement déjà complexe).
- Un *environnement* peut être complexe, par exemple les applications mobiles sont plus complexes à développer et à déboguer que les applications simples sur PC, à cause de l'environnement sans fil, des écrans tactiles, la pile limitée, etc.

La figure 1.4 présente les sources de complexité ainsi que leurs noms qu'on va utiliser dans ce manuel:

1.3.1 Complexité inhérente (provenant du problème)

La complexité inhérente est au sein du problème que résout un logiciel. Elle est souvent *visible* à l'utilisateur du logiciel. Elle se compose des parties du logiciel qui sont nécessairement des problèmes difficiles. N'importe quel logiciel qui tente de résoudre ces problèmes aura une manifestation de cette complexité dans son implémentation. Exemple: un logiciel qui aide à faire des déclarations de revenus aura une complexité inhérente due à la complexité des lois fiscales qui spécifient comment doit être préparée une déclaration.

1.3.2 Complexité circonstancielle (provenant des choix de conception)

Les choix que font les ingénieur.e.s dans un projet peuvent amener de la complexité circonstancielle. En tant qu'ingénieur.e.s, nous avons un devoir de contrôler cette forme de complexité, par exemple en prenant soin avec un choix de cadriel Web ou d'architecture logicielle. La complexité circonstancielle peut aussi être due à des contraintes imposées sur la conception, comme une utilisation obligatoire d'une vieille base de données ou d'une bibliothèque logicielle héritée, d'un langage de programmation, etc. La complexité circonstancielle (aussi appelée accidentelle) peut être gérée avec des technologies, par exemple les débogueurs, les patrons de conception (un Adaptateur pour les bases de données différentes), etc.

1.3.3 Complexité environnementale (provenant de l'environnement d'exécution)

Cette forme de complexité comprend des aspects d'une solution qui ne sont pas sous le contrôle des ingénieur.e.s. Dans un environnement d'exécution, il y a des dimensions comme le ramasse-miettes (*garbage collection*), l'ordonnancement des fils d'exécution (*threads*) sur un serveur, l'utilisation de *containers* (à la Docker), etc. qui peuvent affecter la qualité d'un logiciel. Les ingénieur.e.s doivent gérer ces formes de complexité, mais il n'y a pas beaucoup de stratégies évidentes face aux technologies qui évoluent très vite.

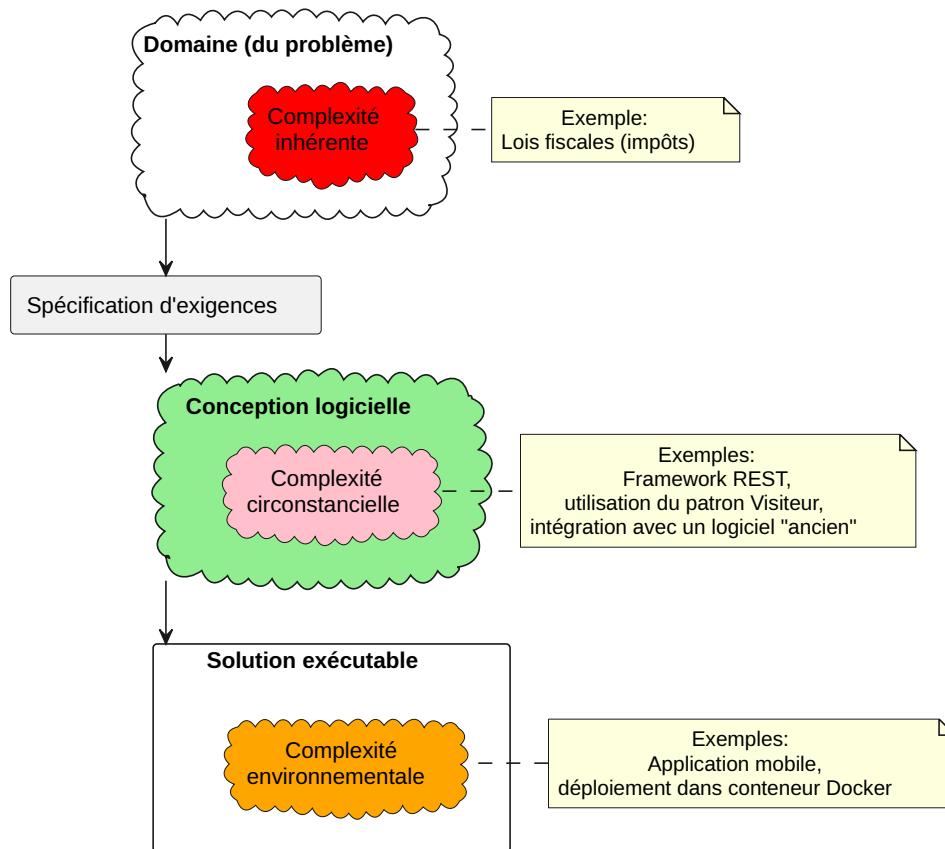


Figure 1.4: Sources de complexité.

1 Analyse et conception de logiciels

1.4 Survol de la méthodologie

La figure 1.5 présente la méthode d'analyse et de conception proposée dans ce manuel. C'est une adaptation de plusieurs figures présentées par Larman Larman (2005).

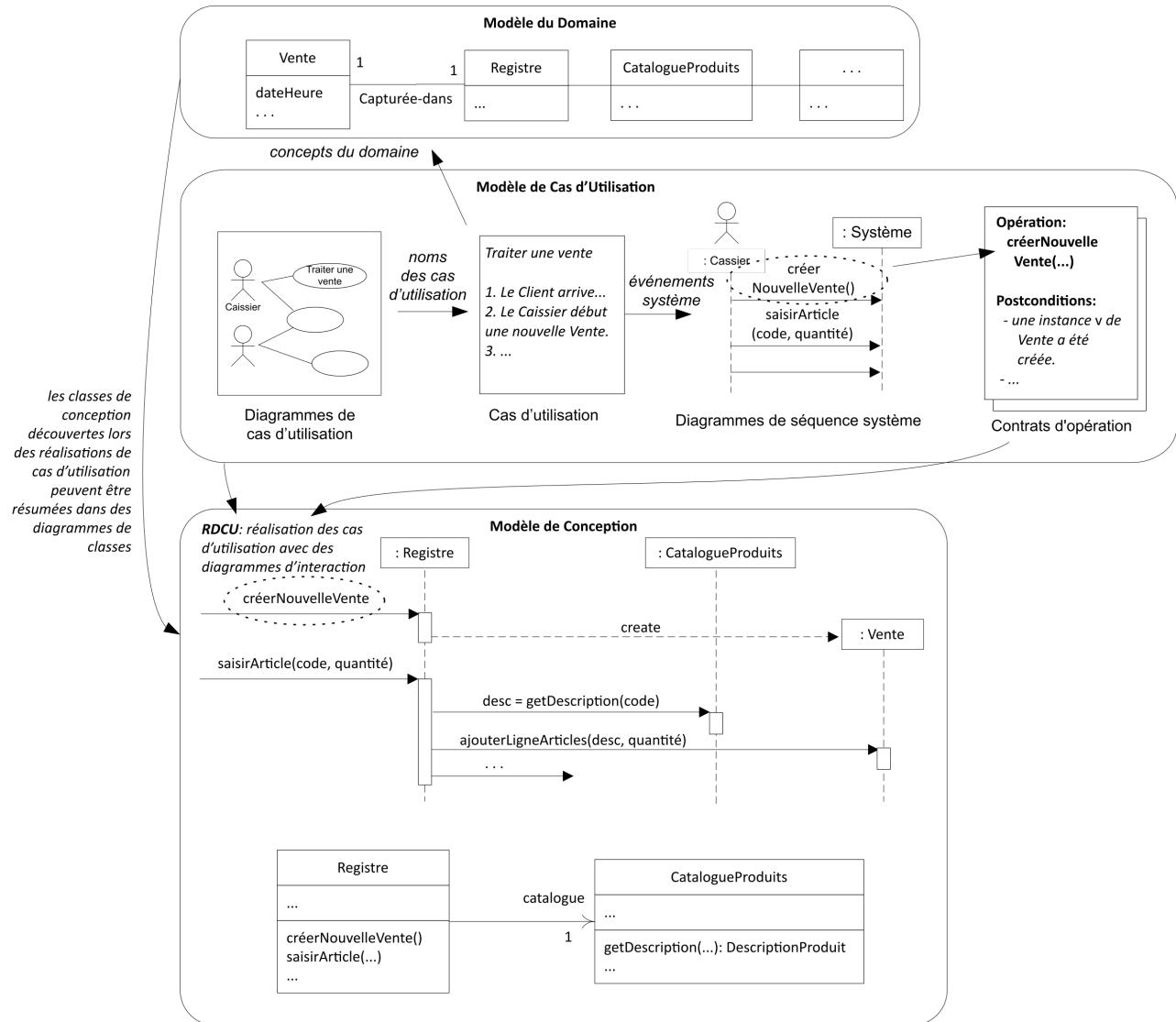


Figure 1.5: Survol de la méthodologie.

1.5 Développement itératif, évolutif et agile

Nous adaptons également un processus moderne de développement avec des itérations, selon une méthodologie « agile ». Dans le chapitre 2 du livre de Craig Larman, on définit le processus itératif et adaptatif ainsi que les

concept fondamental du « Processus Unifié », qui est une représentation générique de cette stratégie de développement.

Nous résumons les points importants ainsi:

- Le développement itératif et évolutif implique de programmer et de tester précocement un système partiel dans des cycles répétitifs.
- Un cycle est nommé une itération et dure un temps fixe (par exemple, trois semaines) comprenant les activités d'analyse, de conception, de programmation et de test, ainsi qu'une démonstration pour solliciter des rétroactions du client (voir la figure 1.6).
- 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 reprend éventuellement dans une itération ultérieure.
- 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 1.7). Cette instabilité peut être particulièrement prononcée dans un contexte d'entreprise en démarrage.
- Dans une itération, la modélisation (par exemple, avec l'UML) se fait au début et devrait prendre beaucoup moins de temps (quelques heures) que la programmation, qui n'est pas triviale (voir la figure 1.8). Selon le contexte du projet (voir le **Spectre de la conception**), on peut décider de ne pas faire de la modélisation. Cependant, en fonction de la complexité du projet à réaliser, cela peut amener des risques, ce que l'on appelle la **dette technique**.

Le développement itératif et incrémental amène plusieurs avantages selon Larman (2005):

- diminution d'échecs, une amélioration de la productivité et de la qualité;
- une gestion proactive des risques élevés (risques techniques, exigences, objectifs, convivialité, etc.);
- des progrès immédiatement visibles;
- la rétroaction, l'implication des utilisateurs et l'adaptation précoce;
- la complexité est gérée (restreinte à une itération);
- la possibilité d'exploiter méthodiquement les leçons tirées d'une itération.

1 Analyse et conception de logiciels

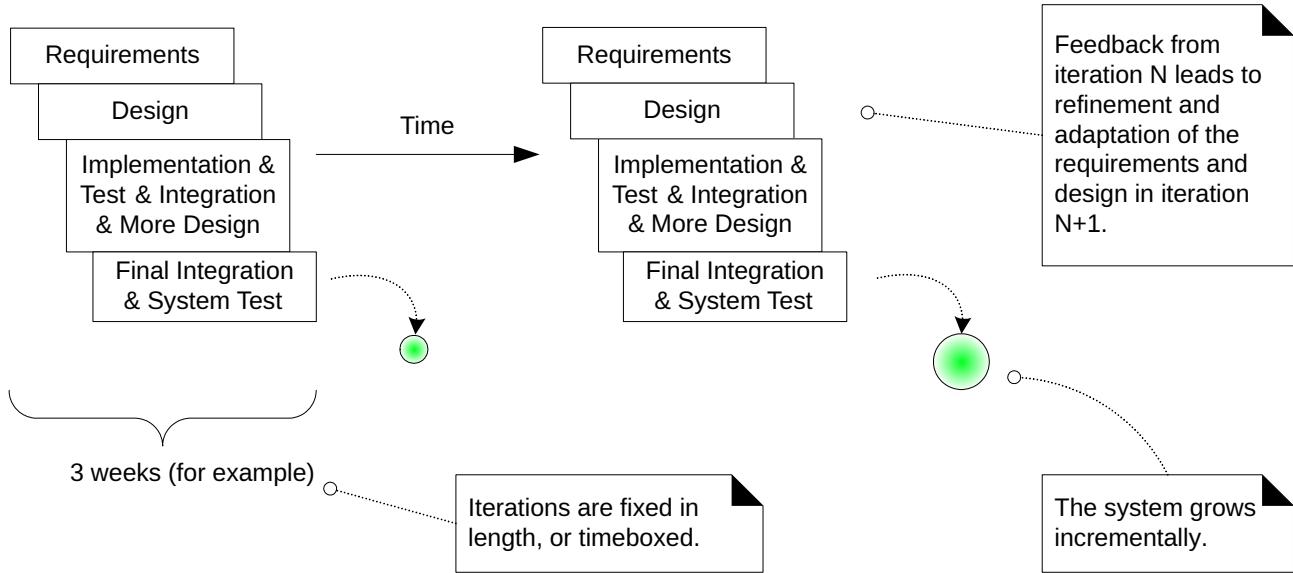


Figure 1.6: Le développement itératif et incrémental (Figure 2.1 du livre).

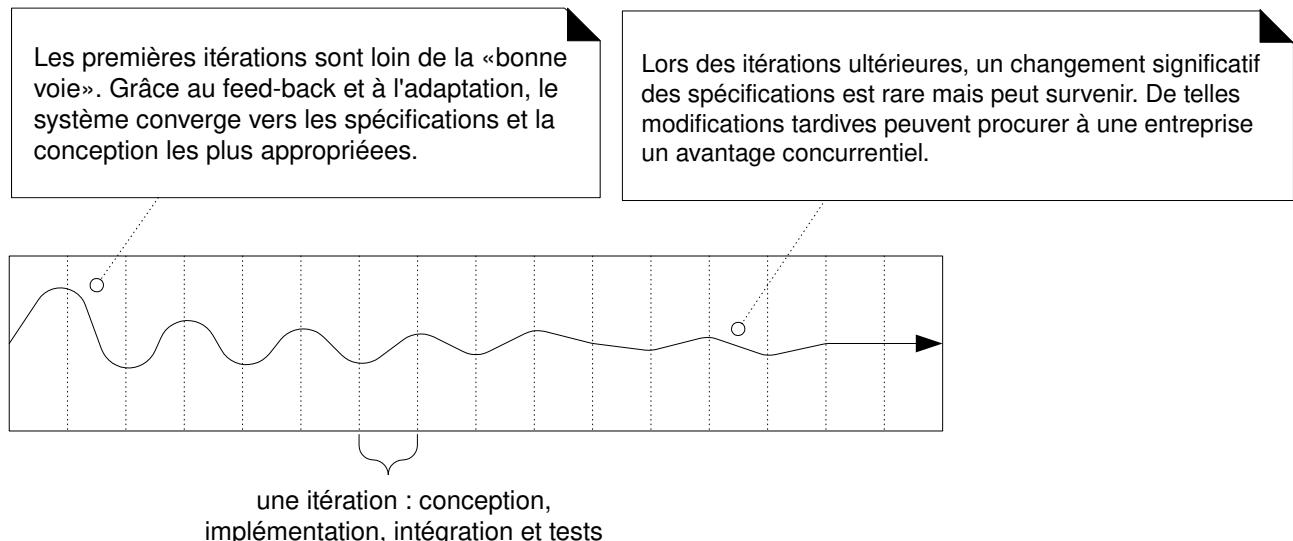


Figure 1.7: Rétroaction et adaptation itératives convergent vers le système souhaité (Figure 2.2 du livre).

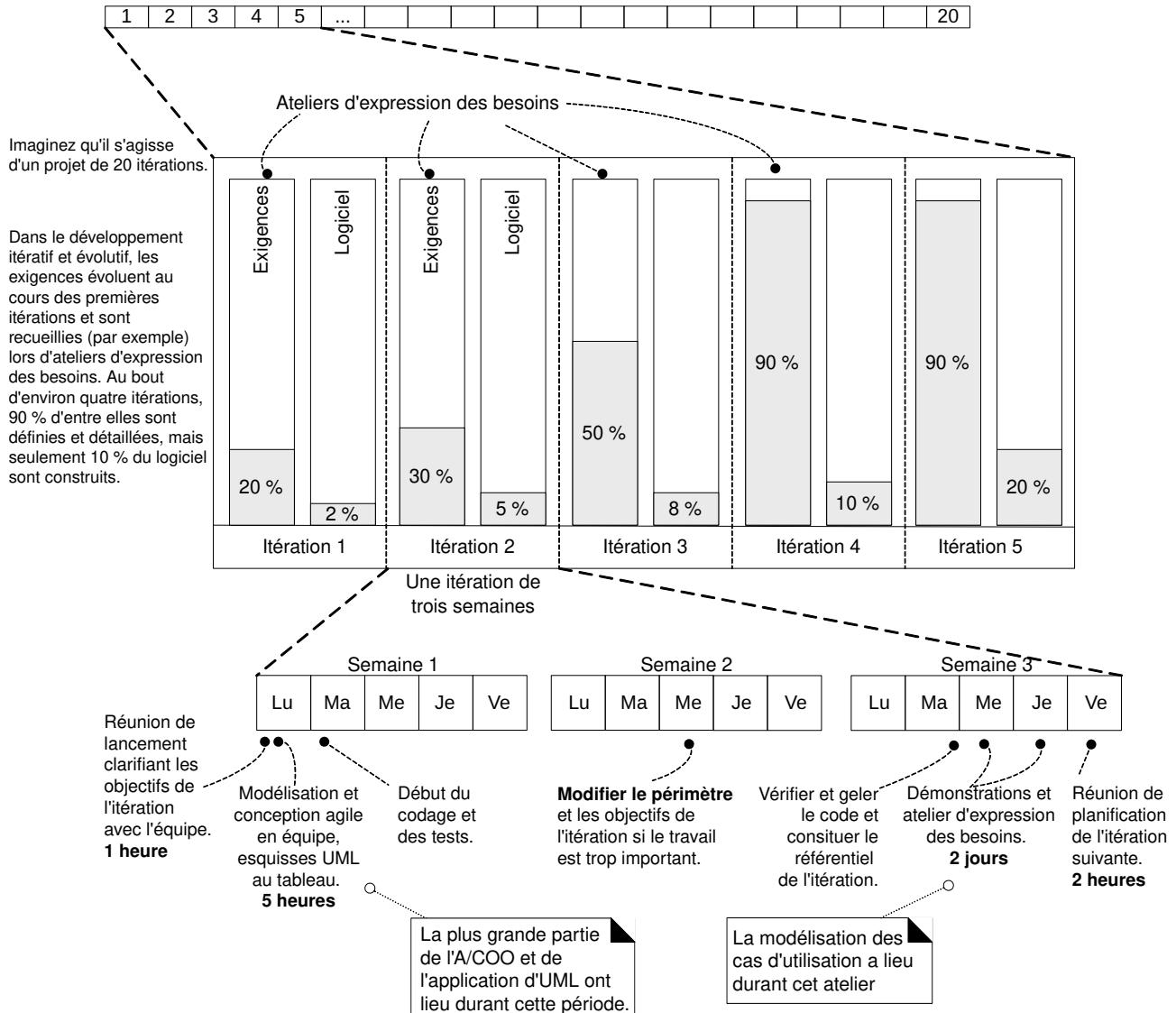


Figure 1.8: Analyse et conception évolutives, majoritairement effectuées dans les premières itérations (Figure 2.4 du livre).

1 Analyse et conception de logiciels

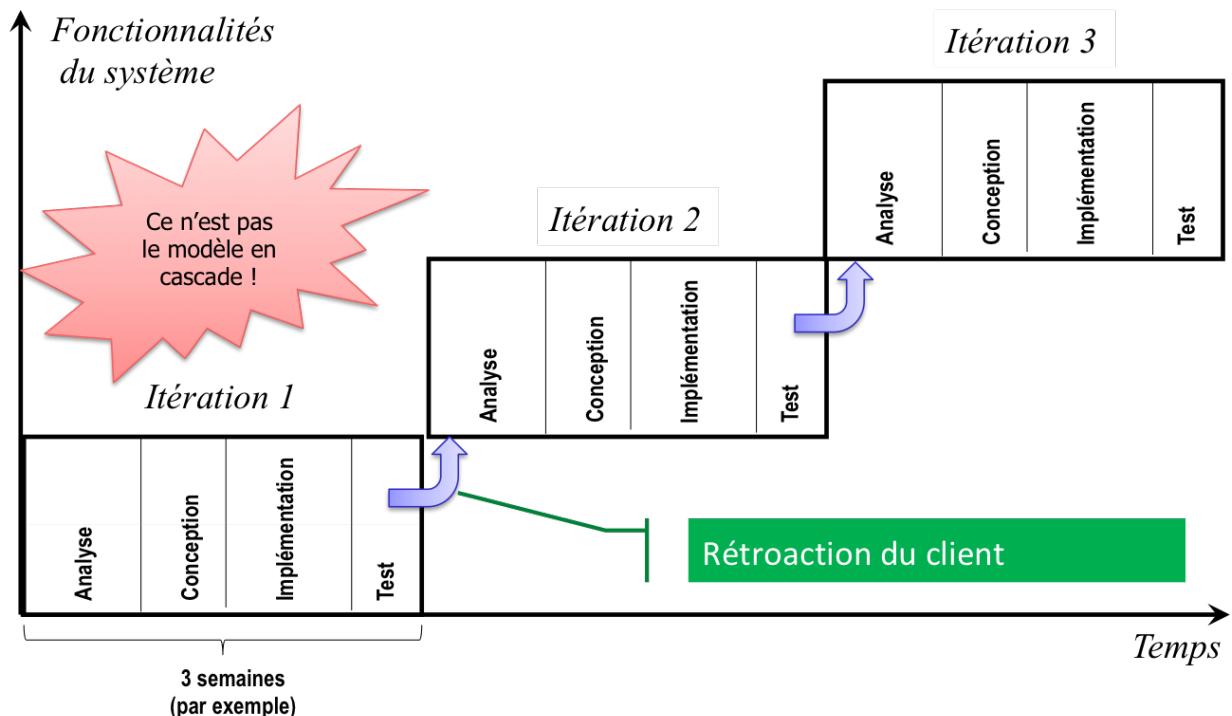


Figure 1.9: Processus itératif et évolutif.



Figure 1.10: Nuage de mots importants.

2 Besoins (exigences)

Dans le développement de logiciels, il est normal de spécifier ce que le logiciel est censé faire. Selon la méthodologie proposée dans ce manuel, la spécification des besoins n'est pas facultative! La spécification des besoins est la base de la méthodologie de développement.

D'abord, qu'est-ce qu'une exigence? C'est une condition documentée à laquelle un logiciel (ou un système) doit satisfaire. C'est quelque chose qui facilite la vie d'un utilisateur ou qui amène une valeur socio-économique.

Voici un exemple d'une exigence d'un système de messagerie instantanée (comme Discord ou Slack): il doit permettre à des utilisateurs d'écrire les messages qui sont affichés dans un canal. Cette fonctionnalité a une utilité évidente – la communication. Il s'agit d'une *exigence de fonctionnalité*. Si ces messages doivent être visibles à tous les utilisateurs dans le canal en moins de 0,5 secondes, alors il s'agit d'une exigence sur *la qualité de la performance* du système. Lorsqu'il s'agit des 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. Le nom porte à la confusion, car une chose qui ne fonctionne pas est « non fonctionnelle ». Pour cette raison, elle sont aussi appelées des exigences sur les qualités ou sur les contraintes. On peut aussi les appeler informellement les « ilités », car ces qualités sont souvent des aptitudes du système: la maintenabilité, la convivialité, la testabilité, etc.

2.1 FURPS+

FURPS+ est un modèle (avec un acronyme) pour classer les exigences (besoins) d'un logiciel. Voici un résumé de FURPS+ (voir [Larman 2005, sect. 5.4](#)):

- **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 universitaire, le système doit avoir un temps de réponse de moins de 2 secondes.

2 Besoins (*exigences*)



Figure 2.1: Ramasser les besoins non fonctionnels? « Dog Clean Up » de Luis Prado, utilisé selon CC0

- **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 et d'utilisation des données personnelles, etc.

3 Cas d'utilisation

Les cas d'utilisation sont des documents textuels décrivant l'interaction entre un système (un 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.

Les cas d'utilisation sont une manière de documenter les fonctionnalités (les exigences fonctionnelles).

Note

D'autres méthodologies de développement peuvent déterminer les besoins avec les récits utilisateur (*user stories*), qui sont généralement plus courts et moins préscriptifs que des cas d'utilisation. Par exemple, dans un récit utilisateur on ne spécifie pas un ordre d'interactions entre l'acteur et le système. Une raison pour ne pas spécifier autant de détails est que ça peut changer beaucoup (surtout au début du projet). Voir [cette discussion sur stackexchange.com](#) pour savoir plus sur les différences.

La théorie sur *comment écrire* les cas d'utilisation ne fait pas partie de ce manuel (voir [Larman 2005, chap. 6](#)).

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

3.1 Exemple: jeu de Risk

Nous décrivons un cas d'utilisation à l'aide d'un exemple concernant le jeu de Risk.

Selon « Risk ». 2019. [Wikipédia](#). (accédé le 9 décembre 2019):

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.

Alors, nous proposons les étapes (les interactions entre les acteurs et le système) pour ce scénario:

3 Cas d'utilisation



Figure 3.1: Cinq dés utilisés dans le jeu de Risk. By Val42 - <https://en.wikipedia.org/wiki/Image:Risk-dice-example.jpg>, CC By-SA 3.0 Link

3.1.1 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 les étapes 2 à 5 jusqu'à ce que l'attaquant ne puisse plus attaquer ou ne veuille plus attaquer.

3.1.2 Diagramme de cas d'utilisation

La figure 3.2 est un exemple de diagramme de cas d'utilisation.

Un diagramme de cas d'utilisation n'est qu'une sorte de *table des matières* des fonctionnalités. Le diagramme ne montre qu'une faible partie des détails trouvés dans le texte de chaque cas d'utilisation. Le diagramme ne peut donc remplacer la documentation textuelle.

Dans la figure 3.2, le cas d'utilisation « ... » signifie qu'il y a d'autres cas d'utilisation à spécifier concrètement. C'est à dire 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 proposée par Larman Larman (2005), car il faudra concevoir et coder ce scénario, bien qu'il ne soit pas une fonctionnalité connue par l'utilisateur.

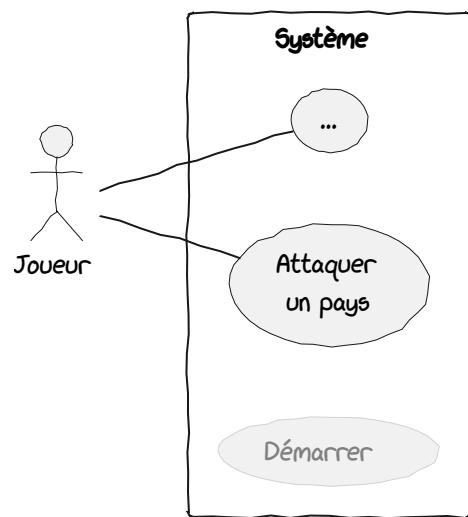


Figure 3.2: Diagramme de cas d'utilisation. ([PlantUML](#))

4 Modèle du domaine (MDD, modèle conceptuel)

Les MDD sont expliqués en détail dans le chapitre 9 , mais voici des points importants:

- 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**.

4.1 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.

4.1.1 Catégories pour identifier des classes conceptuelles

Tableau 4.1: Extrait du tableau 9.1 

Catégorie	Exemples
Transactions d'affaire: Elles sont essentielles, commencez l'analyse par les transactions.	<i>Vente, Attaque, Réservation, Inscription, EmpruntVélo</i>
Lignes d'une transaction: Éléments compris dans une transaction.	<i>LigneArticles, ExemplaireLivre, GroupeCours</i>
Produit ou service lié à une transaction ou une ligne de transaction: Pour quel concept sont faites des transactions?	<i>Article, Vélo, Vol, Livre, Cours</i>
Où la transaction est-elle enregistrée?	<i>Caisse, GrandLivre, ManifesteDeVol</i>
Rôle des personnes liées à la transaction: Qui sont les parties impliquées dans une transaction?	<i>Caissier, Client, JoueurDeMonopoly, Passager</i>
Organisations liées à la transaction : Quelles sont les organisations impliquées dans une transaction?	<i>Magasin, CompagnieAérienne, Bibliothèque, Université</i>
Lieu de la transaction; lieu du service	<i>Magasin, Aéroport, Avion, Siège, LocalCours</i>
Événements notables, à mémoriser	<i>Vente, Paiement, JeuDeMonopoly, Vol</i>

Catégorie	Exemples
Objets physiques: Important surtout lorsqu'il s'agit d'un logiciel de contrôle d'équipements ou de simulation.	<i>Article, Caisse, Plateau, Pion, Dé, Vélo</i>
Description d'entités : Voir section 9.13 pour plus d'informations.	<i>DescriptionProduit, DescriptionVol, Livre</i> (en opposition avec <i>Exemplaire</i>), <i>Cours</i> (en opposition avec <i>CoursGroupe</i>)
Catalogues : Les descriptions se trouvent souvent dans des catalogues	<i>CatalogueProduits, CatalogueVols, CatalogueLivres, CatalogueCours</i>
Conteneurs : Un conteneur peut contenir des objets physiques ou des informations.	<i>Magasin, Rayonnage, Plateau, Avion, Bibliothèque</i>
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>
Instruments financiers	<i>Espèces, Chèque, LigneDeCrédit</i>
Plannings, manuels, documents régulièrement consultés pour effectuer un travail	<i>MiseAJourTarifs, PlanningRéparations</i>

4.2 Attributs

Les attributs sont le sujet de la section 9.16 . 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 de Larman (2005) il y a plusieurs exemples sans type.
- On ne se soucie pas de la visibilité des attributs dans un MDD.
- Faites attention à la confusion des attributs et des classes. Si on ne pense pas un concept *X* en termes alphanumériques dans le monde réel, alors il s'agit probablement d'une classe conceptuelle. Par exemple, dans le monde réel, une université n'est composée ni de chiffres ni de lettres. Elle doit être une classe conceptuelle. Voir la section 9.12 .
- De la même manière, faites attention aux informations qui sont mieux modélisées par des associations. Par exemple dans la figure 4.1 la classe *Pays* n'a pas un *attribut joueur:Joueur* (qui contrôle le Pays); elle a plutôt une *association* avec la classe *Joueur* et un verbe *contrôle*.

⚠ Il est vrai que dans un langage de programmation comme Java, les associations doivent être les attributs dans les classes, car il s'agit des classes *logicielles*. Cependant, dans un modèle du domaine nous évitons des attributs si une association peut mieux décrire la relation. La relation relie visuellement les deux classes conceptuelles et elle est décrite avec un verbe.

4.3 Associations

Les associations dans le MDD sont le sujet de la section 9.14 . 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 4.1:
 - 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 4.2.
- 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 4.1.

Tableau 4.2: Extrait du tableau 9.2  (liste d'associations courantes).

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>

4 Modèle du domaine (MDD, modèle conceptuel)

Catégorie	Exemple
A est un produit pour une transaction (ou un élément de transaction) B	<i>Article – LigneArticles (ou Vente) Vol – Réservation</i>
A est un rôle lié à une transaction B	<i>Client – Paiement Passager – Billet</i>
A est une partie logique ou physique de B	<i>Tiroir – Registre Case – Plateau Siège – Avion</i>
A est physiquement ou logiquement contenu dans B	<i>Registre – Magasin Joueur – Monopoly Passager – Avion</i>
A est une description de B	<i>DescriptionProduit – Article DescriptionVol – Vol</i>
A est connu/consigné/enregistré/saisi dans B	<i>Vente – Registre Pion – Case Réservation – ManifesteDeVol</i>
A est un membre de B	<i>Caissier – Magasin Joueur – Monopoly Pilote – CompagnieAérienne</i>
A est une sous-unité organisationnelle de B	<i>Rayon – Magasin Maintenance – CompagnieAérienne</i>
A utilise, gère ou possède B	<i>Caissier – Registre Joueur – Pion Pilote – Avion</i>
A est voisin de B	<i>Article – Article Case – Case Ville – Ville</i>

4.4 Exemple de MDD pour le jeu de Risk

La figure 4.1 est un MDD pour le jeu de Risk, selon l'exemple mentionné dans le chapitre sur les cas d'utilisation.

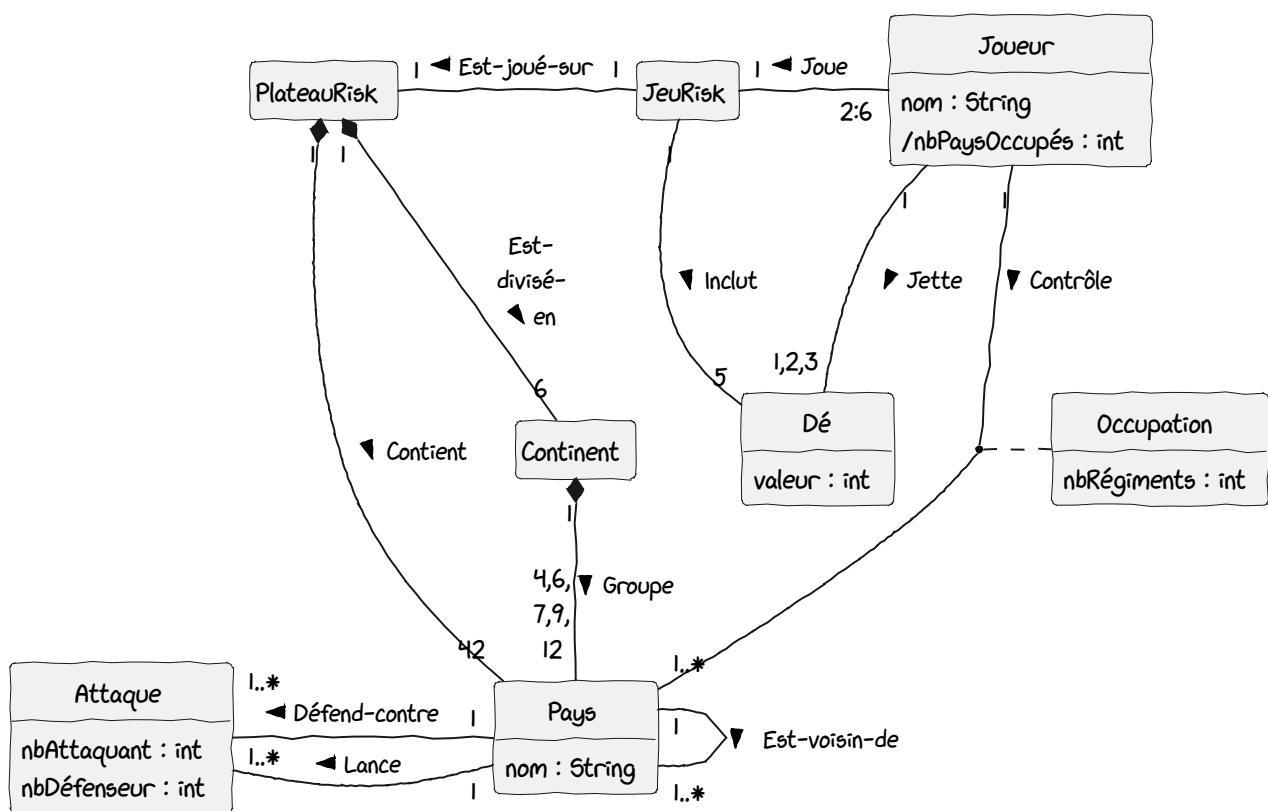


Figure 4.1: Modèle du domaine du jeu de Risk. ([PlantUML](#))

4.5 Attributs dérivés

Les attributs dérivés sont expliqués en détail dans la section 9.16 [■](#). 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 4.2 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.

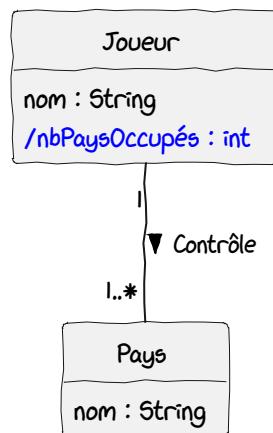


Figure 4.2: `nbPaysOccupés` est un attribut dérivé et sa valeur sera calculée selon le nombre de pays de l'association. ([PlantUML](#))

4.6 Classes de « description » et de catalogues

Deux catégories de classes conceptuelles qui vont de pair sont les *descriptions d'entités* et les *catalogues* qui agrègent les descriptions. Elles sont expliquées en détail dans la section 9.13 [■](#). Voici des conditions pour utiliser correctement une classe de description d'une autre classe « X »:

- Il faut disposer de la description d'un produit ou d'un service « X » indépendamment de l'existence actuelle des « X ». Par exemple, il pourrait y avoir une rupture de stock d'un Produit (aucune instance actuelle), mais on a besoin de connaître son prix. La classe DescriptionProduit permet d'avoir cette information, même s'il n'y a plus d'instances de Produit. Un autre exemple est un trimestre où un cours *LOG711* ne se donne pas (il n'y a pas de GroupeCours de *LOG711* dans le trimestre actuel). Alors une classe Cours (qui joue le rôle de description) sert pour spécifier le nombre de crédits, les cours préalables, etc.
- La suppression d'instances de « X » entraîne la perte de l'information qui doit être mémorisée, mais a été incorrectement associée à l'entité en question.
- La classe de description réduit la duplication des informations.

La figure 4.3 présente une classe de description pour le contexte de cours et groupe-cours.

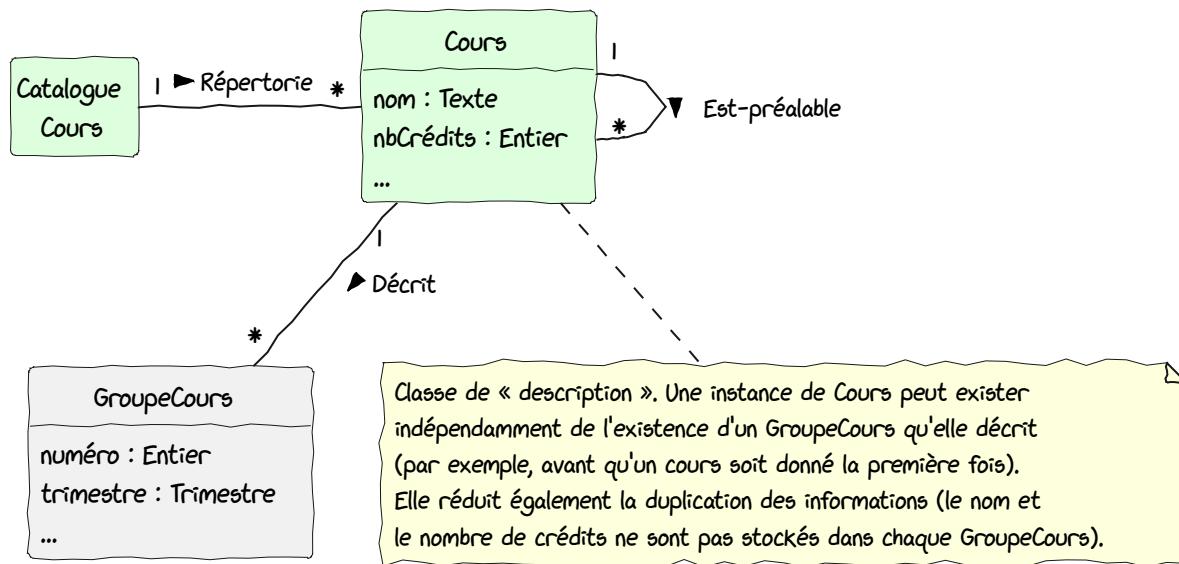


Figure 4.3: Cours joue le rôle de description d'entités (les groupes-cours).

⚠ Avertissement

Attention de ne pas faire l'erreur naïve d'utiliser une classe de description simplement pour « décrire » une autre classe. Voir la figure 4.4 pour un exemple.

4.7 Classes d'association

Les classes d'association dans le MDD sont le sujet de la section A32.10/F26.10 E.

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 4.5, 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.

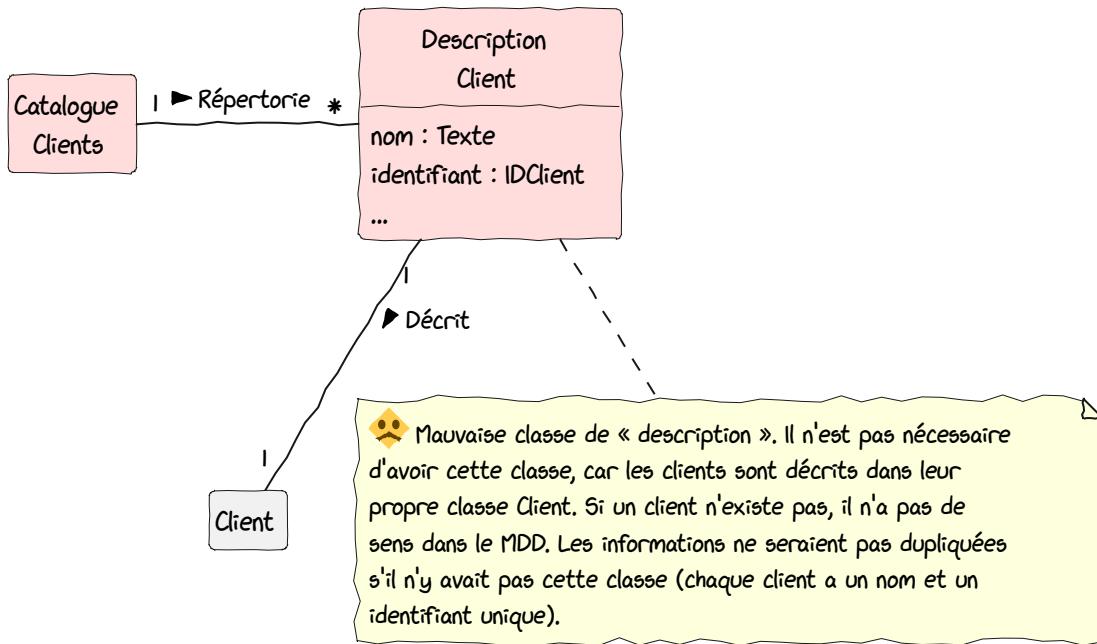


Figure 4.4: Erreur fréquente - utiliser une classe de description sans justification.

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.

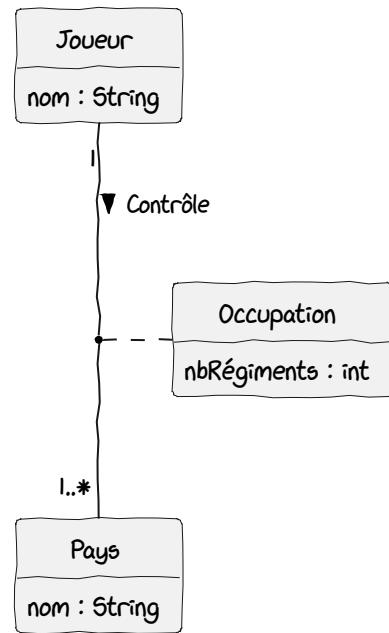
Voir le livre obligatoire pour plus d'exemples.

4.8 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 . 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, par exemple, la classe Continent qui groupe les Pays dans la figure 4.1. Larman propose d'utiliser la composition lorsque:
 - la durée de vie du composant est limitée à celle du composite (lorsqu'un objet Continent est instancié ça doit grouper des instances de Pays pour être cohérent), il existe une dépendance création-suppression de la partie avec le tout (ça ne fait pas de sens de supprimer un objet Pays d'une instance de Continent dans le jeu de Risk).

Figure 4.5: Classe d'association dans le MDD Jeu de Risk. ([PlantUML](#))

- il existe un assemblage logique ou physique évident entre le tout et les parties (on ne peut construire un Continent sans les Pays).
- certaines propriétés du composite, comme son emplacement, s'étendent aux composants.
- les opérations que l'on peut appliquer au composite, telles que destruction, déplacement et enregistrement, se propagent aux composants.
- Généralisation/specialisation, voir le livre pour les exemples et les directives, notamment la règle des 100% (conformité à la définition) et la règle « est-un » (appartenance à l'ensemble).
- Attribut dérivé, par exemple, /nbPaysOccupés dans la classe Joueur est un attribut dérivé de l'association entre Joueur et Pays (figure 4.2).
- Hiérarchies dans un MDD et héritage dans l'implémentation
- Noms de rôles
- Organisation des classes conceptuelles en Packages (surtout lorsque le MDD contient un nombre important de classes conceptuelles)

4.9 FAQ MDD

4.9.1 Y a-t-il un MDD pour chaque cas d'utilisation?

Selon la méthodologie de ce manuel, bien qu'une application ait souvent plusieurs fonctionnalités (cas d'utilisation), il n'y a qu'un seul MDD.

4 Modèle du domaine (MDD, modèle conceptuel)

Cela dit, le MDD est comme un fichier de code source, puisque sa *version* peut évoluer avec le projet. Le MDD évoluera normalement après chaque itération, car on fait une nouvelle analyse pour les nouvelles fonctionnalités visées dans l'itération. Au début du projet, le MDD 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 modélise davantage de concepts reliés aux problèmes traités par les nouvelles fonctionnalités à réaliser.

Par exemple, la version initiale du MDD (chapitre 9 ) ne traite pas la fonctionnalité de paiement par carte de crédit. Les classes conceptuelles modélisant la problématique de paiements par carte de crédit sont absentes dans le MDD initial. Plus tard (après plusieurs itérations, dans le chapitre sur le raffinement du MDD), on voit un MDD beaucoup plus riche qui reflète la modélisation des concepts reliés à des fonctionnalités comme les paiements par carte de crédit, les demandes d'autorisation de paiement, etc.

4.9.2 Un modèle du domaine est-il la même chose qu'un modèle de données?

Voici la réponse de Craig Larman dans la section 9.2  :

Un modèle du domaine n'est pas un modèle de données (qui représente par définition des objets persistants stockés quelque part).

Il peut y avoir des concepts dans un domaine qui ne sont pas dans la base de données. Considérez l'exemple de la carte de crédit utilisée pour payer mais qui n'est jamais stockée pour les raisons de sécurité. Avec seulement un modèle de données, cette classe conceptuelle ne serait jamais modélisée. Larman précise:

N'excluez donc pas une classe simplement parce que les spécifications n'indiquent pas un besoin évident de mémoriser les informations la concernant (un critère courant pour la modélisation des données quand on conçoit des bases de données relationnelles mais qui n'a pas cours en modélisation d'un domaine), ou parce que la classe conceptuelle ne possède pas d'attributs. Il est légal d'avoir une classe conceptuelle sans attribut, ou une classe conceptuelle qui joue un rôle exclusivement comportementale dans le domaine.

Vous pouvez aussi lire [cette question !\[\]\(f67d1f11738c6cddcd12729f5c48a09e_img.jpg\)](#).

5 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 détail dans le chapitre 10 , mais voici des points importants pour la méthodologie de ce manuel:

- 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 5.1 (le « : » signifie une instance).
- Le Système est un objet (une instance :Système) 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*.

5.1 Exemple: DSS pour Attaquer un pays

La figure 5.1 est un exemple de DSS pour le cas d'utilisation *Attaquer un pays*. Vous pouvez noter tous les détails (titre, arguments, types).

5.2 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 5.2

5 Diagrammes de séquence système (DSS)

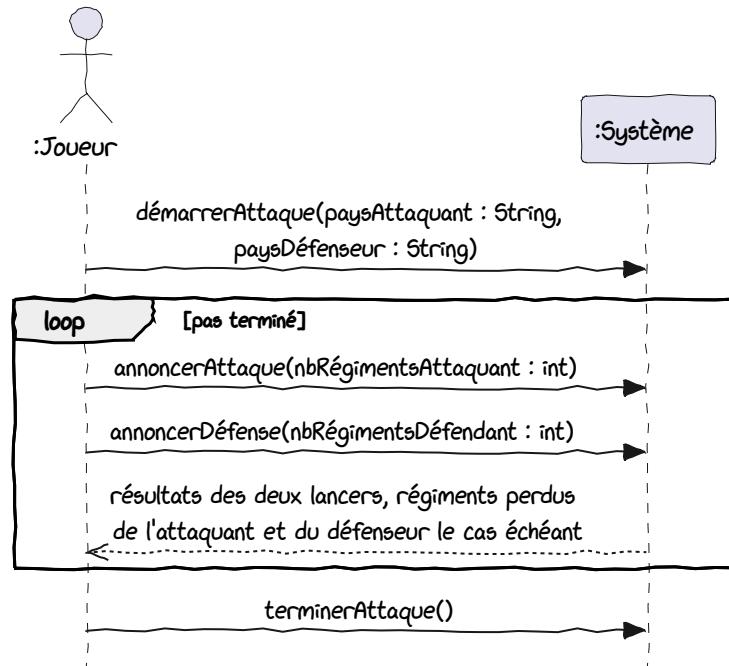


Figure 5.1: Diagramme de séquence système pour *Attaquer un pays*. ([PlantUML](#))

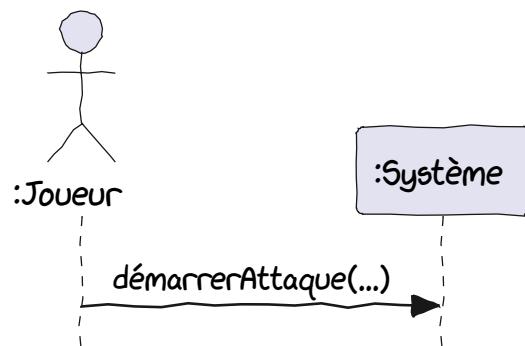


Figure 5.2: Une opération système dans un DSS. C'est une abstraction. ([PlantUML](#))

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 5.3 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).

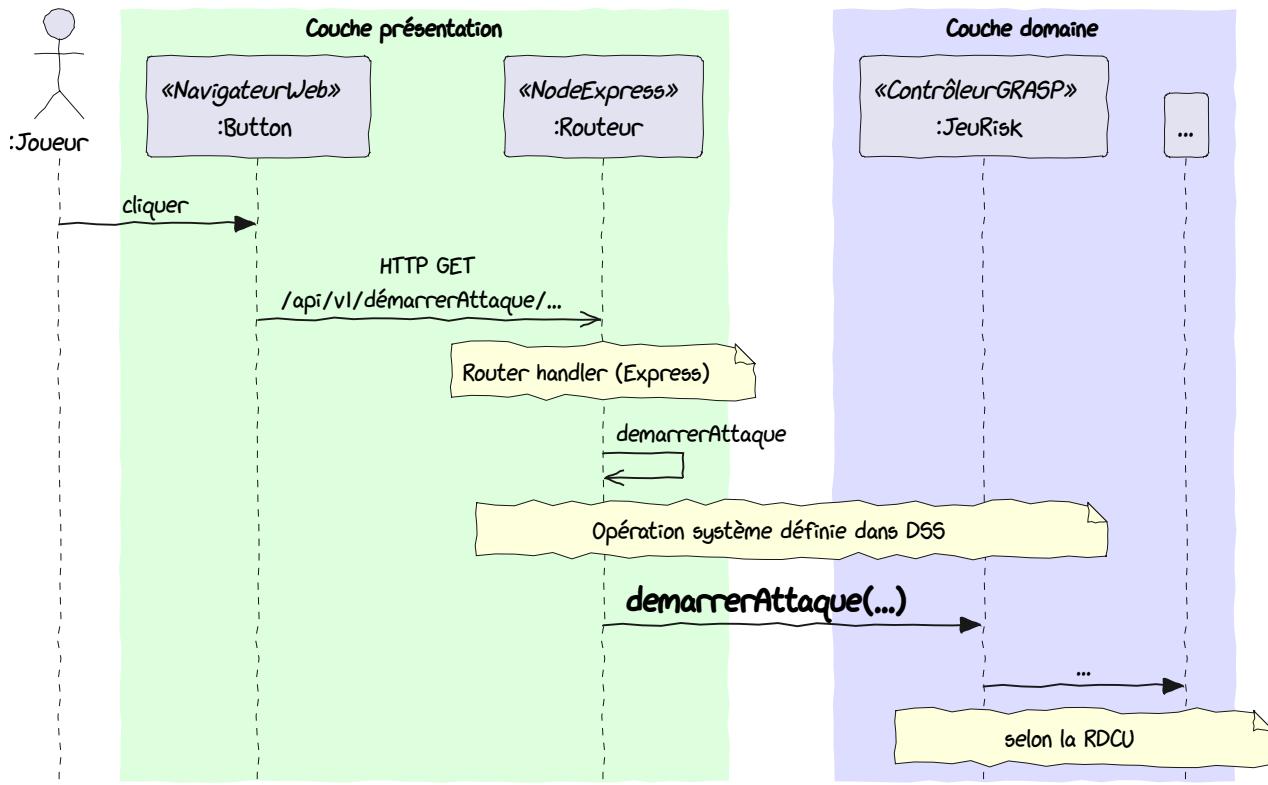


Figure 5.3: Une opération système est envoyée par la couche présentation et elle est reçue dans la couche domaine par son contrôleur GRASP. Ceci est un exemple avec un navigateur web, mais d'autres possibilités existent pour la couche présentation. ([PlantUML](#))

A La figure 5.3 est à titre d'information seulement. Un DSS ne rentre pas dans tous ces détails.

5.3 FAQ DSS

5.3.1 Faut-il une opération système après une boucle?

Dans l'exemple pour *Attaquer un pays*, à l'extérieur de la boucle il y a une opération système `terminerAttaque`. Est-ce obligatoire d'avoir une opération système après une boucle?

L'opération système `terminerAttaque` sert pour signaler la fin de la boucle. Le système saura que l'acteur ne veut plus répéter les actions dans la boucle. Mais elle permet aussi de faire des calculs concernant ce qui s'est passé dans la boucle, p. ex. pour déterminer qui contrôle quel pays après les attaques.

Cependant, si vous avez une boucle pour indiquer la possibilité de répéter une action (p. ex. ajouter des produits dans un système d'inventaire) et vous n'avez pas besoin de faire un calcul à la fin, alors une opération système pour terminer une telle boucle n'est pas nécessaire (surtout avec une application web).

5.3.2 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?

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.

5.3.3 Est-ce normal d'avoir une opération système avec beaucoup d'arguments (de type primitif)?

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?

Il n'est pas conseillé de passer des *objets du domaine* comme argument, puisque c'est la couche présentation qui invoque l'opération système. La couche présentation n'est pas censée manipuler directement les objets dans la couche domaine, sinon elle empiète sur les responsabilités de la couche domaine.

Une solution pour réduire le nombre d'arguments sans utiliser un objet du domaine est d'appliquer un **réusinage** pour le *smell* nommé *Long Parameter List*, par exemple [Introduce Parameter Object](#). Notez que l'objet de paramètres que vous introduisez n'est pas un objet (classe) du domaine! La distinction est importante, car la logique d'affaires demeure dans la couche domaine. En TypeScript, une fonction peut être définie avec un

objet de paramètre. Cet exemple montre même comment on peut « déstructurer » l'objet pour déclarer les variables utilisées dans la fonction:

```
// inspiré de https://leanpub.com/essentialtypescript/read
function compteARebours({ initial: number, final = 0,
                           increment: increment = 1, initial: actuel }) {
  while (actuel >= final) {
    console.log(actuel);
    actuel -= increment
  }
}
compteARebours({ initial: 20 });
compteARebours({ initial: 20, increment: 2, final: 4 });
```

5.3.4 Ne serait-il pas plus simple de passer l'objet body de la page web au contrôleur GRASP?

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?

Dans un sens ça serait plus simple (pour le code de la couche présentation). Cependant, nous voulons séparer les couches pour favoriser le remplacement de la couche 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 aviez trois types d'application frontale (navigateur web, application iOS et application Android). Le contrôleur GRASP recevra des représentations de « formulaire » de chaque couche présentation différente. En passant, l'objet `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. Pour respecter les responsabilités, on laisse la couche présentation faire le décorticage et construire une opération système selon l'API définie dans le DSS. Cela simplifie aussi le contrôleur GRASP.

6 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. Dans la méthodologie enseignée dans ce manuel, vous devez être en mesure de justifier vos choix de conception et cela va vous ralentir au début (réflexe du « hacking cowboy » peut-être?). Le but avec les principes GRASP est de (ré)apprendre à faire du code qui fonctionne, mais qui soit également facile à maintenir. C'est normal au début que ça prenne plus de temps, car il faut réfléchir pour appliquer les principes. Une fois que vous avez l'habitude à utiliser les GRASP, vous serez encore rapide avec votre développement, mais en plus votre design sera meilleur sur le plan de la maintenabilité et vous aurez plus de confiance dans vos choix.

6.1 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 6.1 illustre le principe.

À une extrémité, il y a la mentalité 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 pour montrer une idée intéressante.

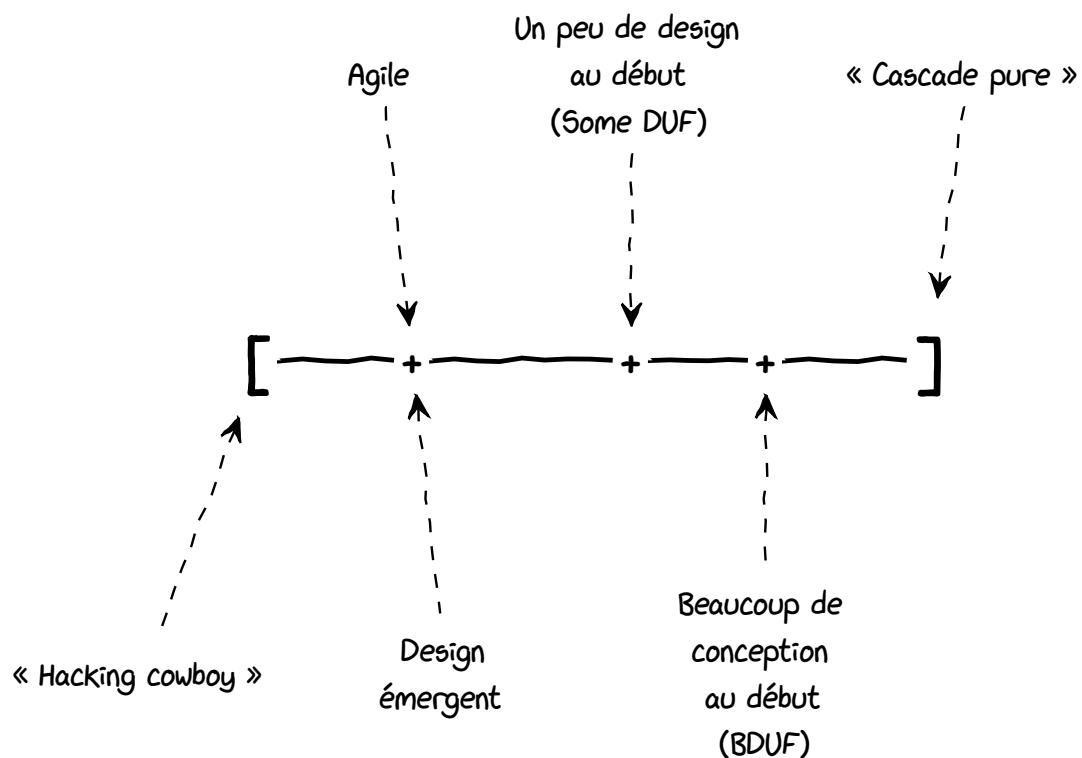


Figure 6.1: Spectre de la conception, adapté de Ford (2009). ([PlantUML](#))

6.1 Spectre de la conception

Au fait, dans certains contextes d'entreprise (par exemple une entreprise en démarrage qui a seulement six mois de financement), c'est une situation similaire. Si une solution de « produit minimum viable » (MVP en anglais) W n'existe pas à la fin de la période de financement, l'entreprise n'existera plus, car il n'y aura pas une deuxième période de financement. Si la compagnie est financée pour une deuxième période, la conception du code pourrait avoir besoin de beaucoup de soins, car elle a été négligée. Cette négligence à la conception (pour la maintenabilité) 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 son livre, Larman (2005) 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 encore utilisée, en dépit des problèmes dus à l'approche. La sécurité et la robustesse des logiciels sont très importantes, 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 le monde réel, parce que une ingénierie ou un ingénieur devrait pouvoir s'adapter selon où les attentes de son travail. Le dogme sur « la bonne manière » de développer un logiciel est souvent sans contexte. C'est le contexte de l'entreprise pour laquelle vous travaillez qui peut déterminer combien d'effort à mettre sur la conception. Cependant, méfiez-vous des entreprises qui ne portent aucune attention à la conception (l'extrême « hacking cowboy » du spectre), même si on vous dit que c'est « agile ».

6.2 Tableau des principes GRASP

Voici un extrait du livre de Larman (2005).

Tableau 6.1: Patterns (principes) GRASP

Pattern	Description
Expert en information	Un principe général de conception d'objets et d'affectation des responsabilités.
F16.11/A17.11	Affecter une responsabilité à l'expert – la classe qui possède les informations nécessaires pour s'en acquitter.
Créateur	Qui crée? (Notez que Fabrique Concète est une solution de rechange courante.)
F16.10/A17.10	Affectez à la classe B la responsabilité de créer une instance de la classe A si l'une des assertions suivantes est vraie: 1. B contient A 2. B agrège A 3. B a les données pour initialiser A 4. B enregistre A 5. B utilise étroitement A
Contrôleur	Quel est le premier objet en dehors de la couche présentation qui reçoit et coordonne (« contrôle ») les opérations système?
F16.13/A17.13	Affectez une responsabilité à la classe qui correspond à l'une de ces définitions: 1. Elle représente le système global, un « objet racine », un équipement ou un sous-système (contrôleur de façade). 2. Elle représente un scénario de cas d'utilisation dans lequel l'opération système se produit (<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)	Comment minimiser les dépendances?
F16.12/A17.12	Affectez les responsabilités de sorte que le couplage (inutile) demeure faible. Employez ce principe pour évaluer les alternatives.
Forte Cohésion (évaluation)	Comment conserver les objets cohésifs, compréhensibles, gérables et, en conséquence, obtenir un Faible Couplage?
F16.14/A17.14	Affectez les responsabilités de sorte que les classes demeurent cohésives. Employez ce principe pour évaluer les différentes solutions.

Pattern	Description
Polymorphisme F22.1/A25.1 	<p>Qui est responsable quand le comportement varie selon le type?</p> <p>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.</p>
Fabrication Pure F22.2/A25.2 	<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 F22.3/A25.3 	<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 F22.4/A25.4 	<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>

6.3 GRASP et RDCU

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

6.4 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.

7 Dette technique

Ce chapitre contient des informations sur le concept de la [dette technique](#) W, qui n'est pas un sujet abordé explicitement par Larman (2005).

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 le [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.

Martin Fowler (2007) a posé la question « Est-ce que ça vaut la peine de faire du bon design? » – peut-on en faire moins pour développer plus vite? Il a proposé un pseudographique comparant la fonctionnalité cumulative (élément difficile à mesurer) avec le temps (voir la figure 7.1). Selon Fowler, le temps pour atteindre la limite de gain de conception (le temps où faire attention à la conception permet un gain de temps) est une question de semaines plutôt que des mois. Mais il avoue que c'est une hypothèse, car il est difficile de mesurer les fonctionnalités cumulatives et d'évaluer ce qu'est un bon design. Le graphe sert à illustrer le principe qu'à un certain moment, ignorer une conception va nuire à la performance des développeurs en ce qui concerne les nouvelles fonctionnalités.

Voici une courte définition complémentaire de la dette technique (Avgeriou et al. 2016):

Un ensemble de constructions ou de mises en œuvre de conception qui sont utiles à court terme, mais qui mettent en place un contexte technique qui peut rendre les changements futurs plus coûteux ou impossibles.

7.1 Origine

La dette technique 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, 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

7 Dette technique

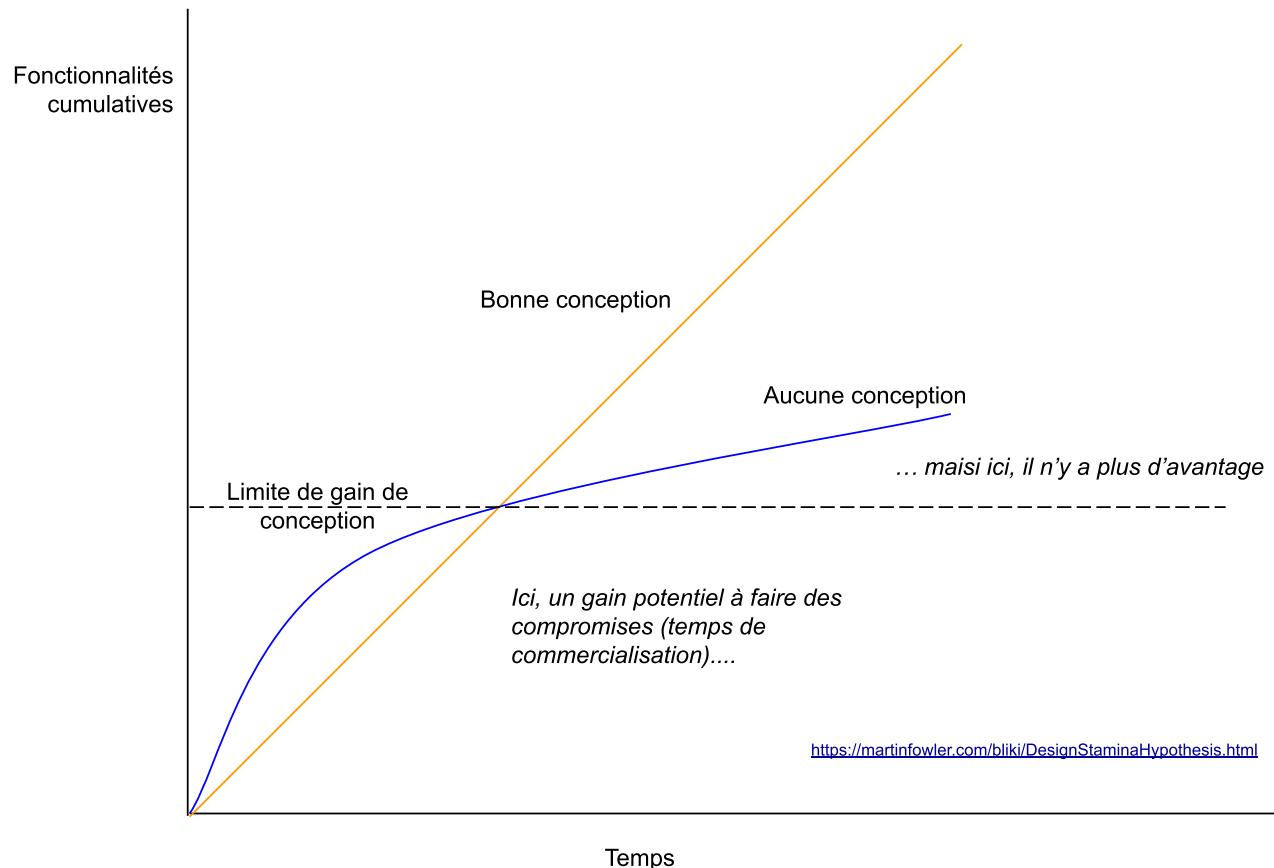


Figure 7.1: « Le pseudographique affiche des fonctionnalités (cumulatives) en fonction du temps pour deux projets stéréotypés imaginaires: l'un avec une bonne conception et l'autre sans conception. Le projet qui ne fait aucune conception ne consacre aucun effort aux activités de conception, qu'il s'agisse de conception initiale ou de techniques agiles. Comme aucun effort n'est consacré à ces activités, ce projet produit des fonctions plus rapidement au départ. » (Fowler 2007)

une réécriture. [Le paradigme des] objets rend 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.

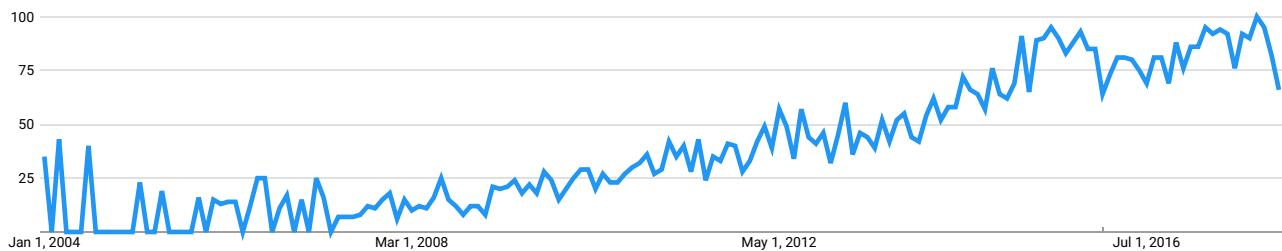


Figure 7.2: Tendances Google (trends.google.com) pour le terme « dette technique » (anglais *technical debt*)

Comme c'est une métaphore puissante, beaucoup de développeurs l'utilisent et c'est un terme avec une certaine popularité (voir à la figure 7.2). Dans une [vidéo](#) plus récente, Cunningham a rappelé que l'origine 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.

7.2 Nuances de la dette technique

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 l'administration 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 7.1 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.

Tableau 7.1: Classification de la dette selon Fowler (2009)

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

7.3 Résumé

- Il n'est pas toujours possible de faire une conception facile à utiliser et à modifier (sans dette technique), puisque certaines choses sont impossibles à prévoir surtout dans une application avec beaucoup de complexité.
- Ignorer complètement le design en faisant du « hacking cowboy » peut apporter un avantage à court terme pour valider des hypothèses rapidement, par exemple dans un contexte d'entreprise en démarrage

sans beaucoup de financement, ou un concours de programmation. Cependant, le code produit dans ce genre de contexte aura des problèmes importants (la dette technique) à long terme.

- La dette technique peut aussi être due à l'incompétence (l'ignorance de bonnes pratiques comme la séparation des couches).

8 Contrats d'opération

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

- 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.

8.1 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 8.1, 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 ».

8 Contrats d'opération

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.

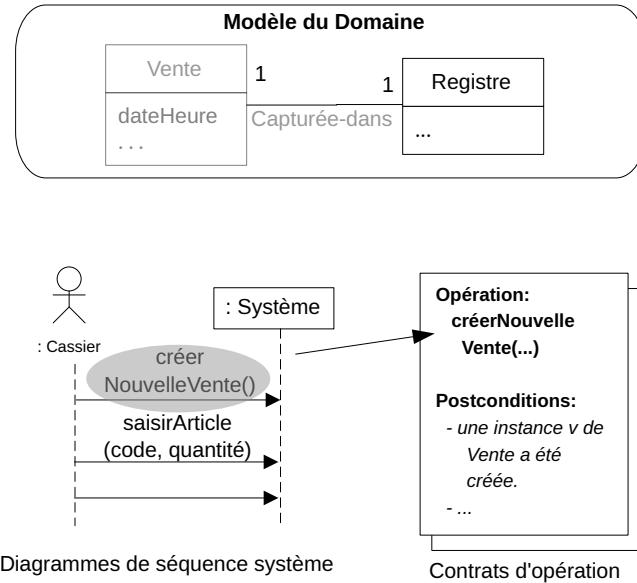


Figure 8.1: Pendant l'opération système `créerNouvelleVente`, une instance de Vente doit être créée. Le contrat d'opération le spécifie dans une postcondition.

8.2 Exemple: Contrats d'opération pour Attaquer un pays

Voir la figure 8.2 pour les changements dans les objets du modèle du domaine correspondant aux postconditions.

8.2.1 Attaquer un pays

8.2.1.1 Opération: démarrerAttaque(paysAttaquant:String, paysDéfenseur:String)

8.2.1.1.1 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`

8.2.1.2 Opération: annoncerAttaque(nbRégimentsAttaquant:int)

8.2.1.2.1 Postconditions

- `a.nbAttaquant` est devenu `nbRégimentsAttaquant`

8.2.1.3 Opération: `annoncerDéfense(nbRégimentsDéfendant:int)`

8.2.1.3.1 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.
- Occupation.nbRégiments est ajusté selon le résultat des valeurs sur une base de correspondance avec paysDéfendant.

8.2.1.4 Opération: `terminerAttaque()`

8.2.1.4.1 Postconditions

- TODO: Handle the change of Occupation?

8.3 Feuille de référence

Pour faire des contrats, voici une démarche générale:

1. Faire un contrat pour chaque opération système
2. Regarder toujours le MDD (vocabulaire du domaine)
3. Rappeler les formes de postconditions
 - a) créer/supprimer instance
 - b) former/briser association
 - c) modifier attributs
4. Ne rien oublier. Marquer le MDD ou dessiner un diagramme d'objets, comme à la figure 8.2 si nécessaire.

8 Contrats d'opération

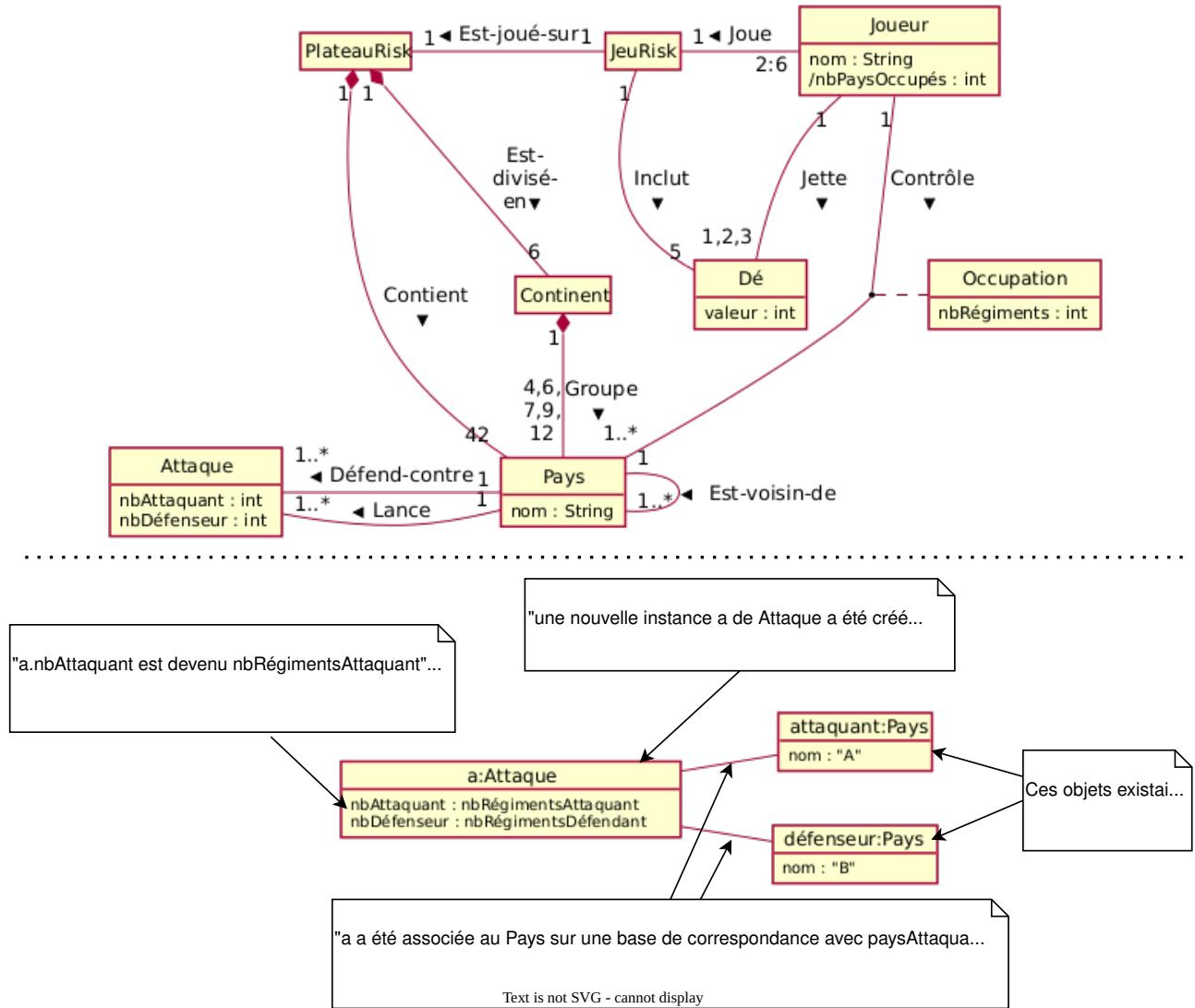


Figure 8.2: Les postconditions décrivent la manipulation d'objets dans un MDD (la partie inférieure ici est un diagramme d'objets)

9 Réalisations de cas d'utilisation (RDCU)

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

- Une RDCU est une synthèse des informations spécifiées dans le MDD, le DSS et les contrats d'opération. Elle sert à esquisser une solution (qui n'est pas encore codée) afin de rendre plus explicite l'activité impliquant des choix de conception. Si vous n'avez pas bien compris les autres éléments (MDD, DSS, etc.), il est difficile de réussir les RDCU. Il est normal de ne pas tout comprendre au début, alors posez les questions si vous ne comprenez pas.
- 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 9.4
- Le livre de Larman (2005) présente quelques RDCU qui sont des *diagrammes de communication*. Cette notation n'est pas utilisée dans ce manuel, 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 une solution codée, il serait nécessaire d'ouvrir plusieurs fichiers afin de voir le code de chaque méthode (message) et on ne peut pas voir toute la dynamique de la même manière. Faire des changements à un diagramme (avant de le coder) est en principe plus facile que changer le code source. On peut également se servir des structures (`List`, `Array`, `Map`, etc.) dans un diagramme, avant que celles-ci ne soient créées.
- Faire des RDCU est une activité créative. Un diagramme dynamique en UML peut avoir une mauvaise logique, car il s'agit d'un dessin. *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 insignifiante.

9 Réalisations de cas d'utilisation (RDCU)

Faire une RDCU est comme faire un *plan* pour un bâtiment, tandis que faire de la programmation est comme *construire* le bâtiment. Si un plan contient des erreurs de conception, on va les savoir lors de la construction. Alors, votre RDCU sera en doute jusqu'à ce que vous la traduisiez en code exécuté et testé.

Tout le processus de proposer une solution (RDCU) peut être visualisé comme un diagramme d'activités, comme dans la figure 9.1.

9.1 Spécifier le contrôleur

Pour commencer une RDCU, on spécifie le contrôleur selon GRASP. Dans les travaux réalisés selon la méthodologie de ce manuel, 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 le tableau 6.1.

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.

9.2 Satisfaire les postconditions

9.2.1 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.

9.2.2 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 .

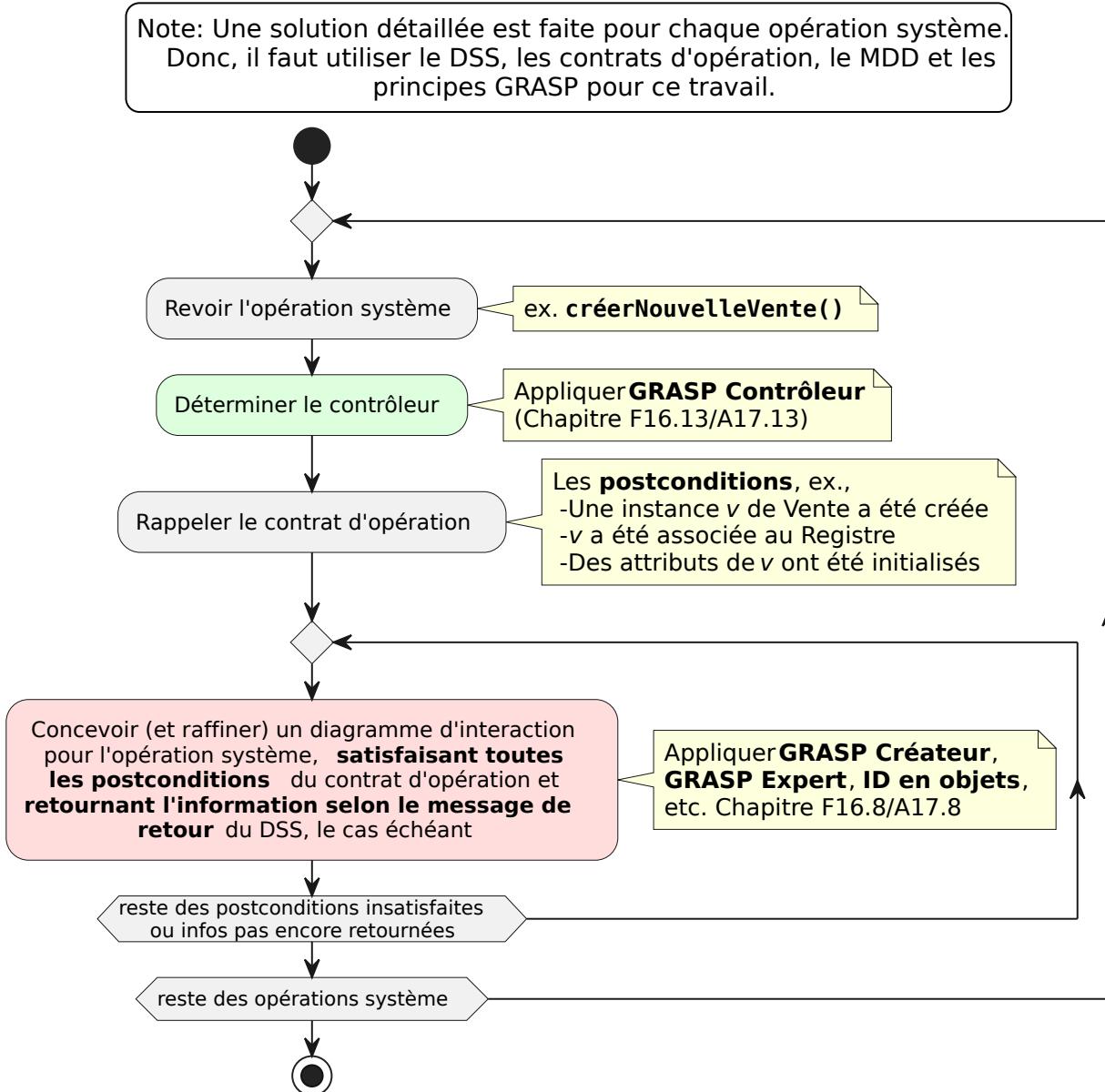


Figure 9.1: Aide mémoire pour faire une RDCU. L'étape en rouge nécessite beaucoup de pratique, selon la complexité des postconditions. Vous pouvez vous attendre à ne pas la réussir du premier coup.
(PlantUML)

9 Réalisations de cas d'utilisation (RDCU)

9.2.3 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 9.2.

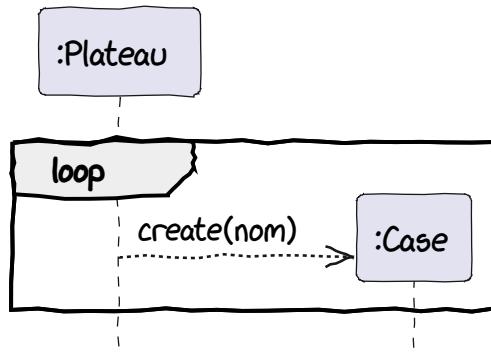


Figure 9.2: Combiner la création d'instance et une modification de son attribut dans un constructeur. ([PlantUML](#))

9.3 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 9.3, 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 Visibilité (F18/A19) .

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

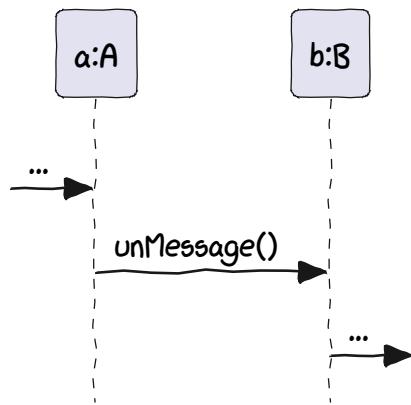


Figure 9.3: L'objet *b* doit être visible à l'objet *a* si *a* veut lui envoyer un message. ([PlantUML](#))

9.4 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 9.4 provenant du chapitre sur **l'Application des patterns GoF** (Figure 23.18) [Larman 2005]. Un autre exemple du livre de Larman (2005) est l'identifiant `codeArticle` transformé en objet `DescriptionProduit` par la méthode
`CatalogueProduits.getDescProduit(codeArticle:String):DescriptionProduit.`

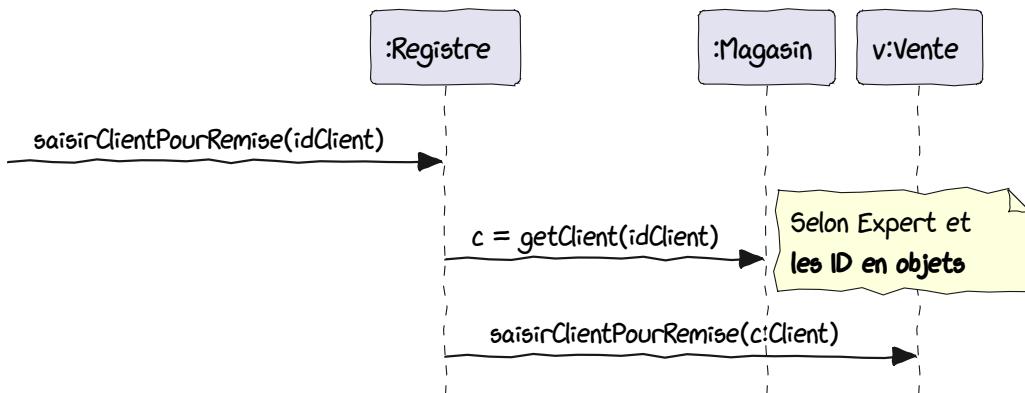


Figure 9.4: Un identifiant `idClient:String` est transformé en objet `c:Client`, qui est ensuite envoyé à la Vente en cours. ([PlantUML](#))

La Section 9.5 explique comment implémenter la transformation avec un tableau associatif.

9.5 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) W. L'exemple du livre de Larman (2005) concerne le problème de repérer une **Case Monopoly** à partir de son nom (**String**). C'est la figure A17.7/F17.7 E.

Notez que les exemples de Larman (2005) 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 9.5 en est un exemple.

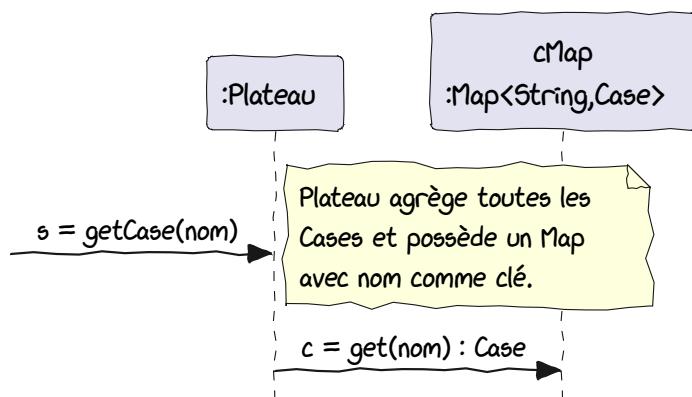


Figure 9.5: Exemple de l'utilisation de tableau associatif pour trouver une Case Monopoly à partir de son nom. (PlantUML)

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

9.6 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** (F17.4, p.345) ou **Initialization and the <Start Up> Use Case** (A18.4, p.274) E 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 4.1, 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 la méthodologie de ce manuel, on n'aborde pas le problème de base de données (c'est le sujet d'un autre cours).

9.7 Réduire le décalage des représentations

Le principe du **Décalage des représentations** est la différence entre la modélisation (la représentation) du problème (du domaine) et la modélisation de la solution. Lorsqu'on fait l'ébauche d'une RDCU, on peut réduire le décalage des représentations principalement en s'inspirant des classes conceptuelles (du modèle du domaine) pour proposer des classes logicielles dans la solution décrite dans la RDCU. Plus une solution ressemble à la description du problème, plus elle sera facile à comprendre.

Une application de pattern GoF à la solution peut nuire à ce principe, car ces patterns rajoutent souvent des classes logicielles n'ayant aucun lien avec le modèle du domaine. Par exemple, un Visiteur ou un Itérateur sont des classes logicielles sans binôme dans le modèle du domaine. Il faut vérifier avec une personne expérimentée (l'architecte du projet si possible) que l'application du pattern est justifiée, qu'elle apporte de vrais bénéfices au design en dépit des désavantages dus à des classes ajoutées. Chaque fois qu'on propose des classes logicielles qui n'ont pas de liens avec la représentation du problème du domaine, on *augmente le décalage des représentations* et on rend la solution un peu plus difficile à comprendre. C'est aussi une forme de **Complexité circonstancielle (provenant des choix de conception)**. Ce dilemme est un bon exemple de la nature pernicieuse de la conception de logiciels. Il est très difficile, même pour les experts en conception, de trouver un bon équilibre entre toutes les forces: la maintenabilité, la simplicité, les fonctionnalités, etc. Vous pouvez en lire plus dans [cette réponse sur StackOverflow](#) ↗.

9.8 Pattern « Faire soi-même »

Dans la section F30.8/A33.7 ↗, Larman mentionne le pattern « Faire soi-même » de Peter Coad (1997) qui permet de réduire le **Décalage des représentations**, même s'il ne représente pas exactement la réalité des objets (voir la figure 9.7):

9 Réalisations de cas d'utilisation (RDCU)

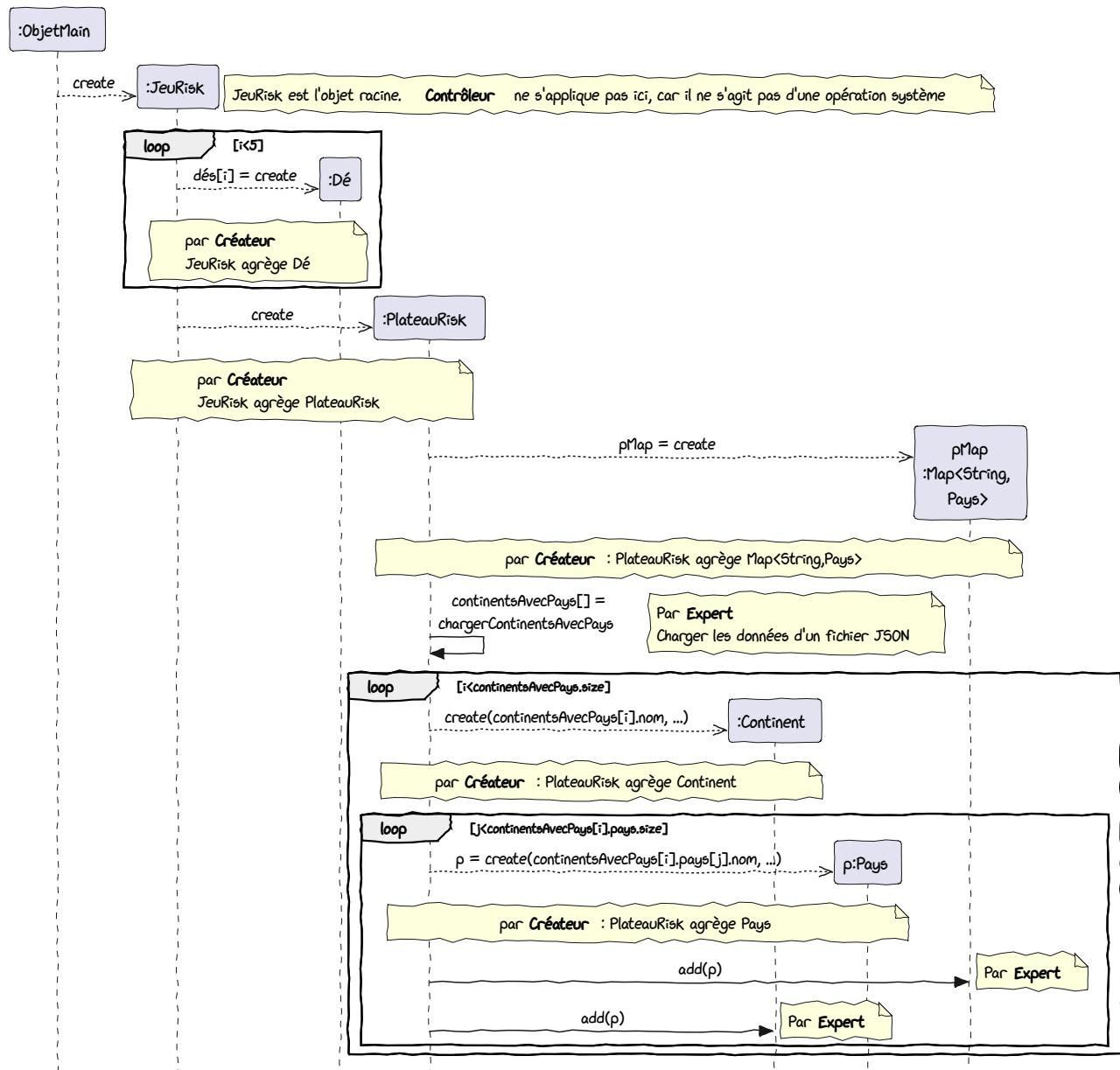


Figure 9.6: Exemple de l'initialisation partielle du jeu de Risk. (PlantUML)

9.8 Pattern « Faire soi-même »

Faire soi-même: « Moi, objet logiciel, je fais moi-même ce qu'on fait normalement à l'objet réel dont je suis une abstraction »

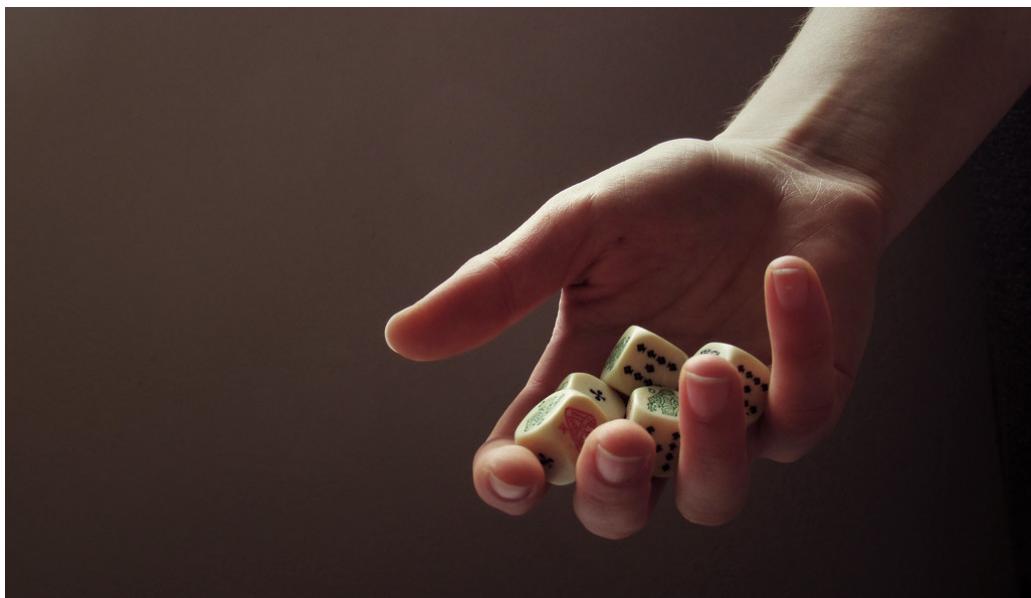


Figure 9.7: Dans la vraie vie, les dés sont lancés par une main (« Hand of chance » (CC BY 2.0) par [Alexandra E Rust](#)).

10 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 10.1. 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 sur le fait que le test soit écrit toujours en premier! Cette discipline est parfois culturelle.

Plusieurs chercheurs ont mené des expériences, par exemple Karac et 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 de manière dogmatique.

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, il s'agit de [Jest](#).

L'exécution de tests peut être même faite à chaque commit du code dans un dépôt comme GitHub.

Il est possible de mesurer la [couverture de code](#) W atteinte par les tests (mais ce sujet sort du cadre de la matière de ce manuel).

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

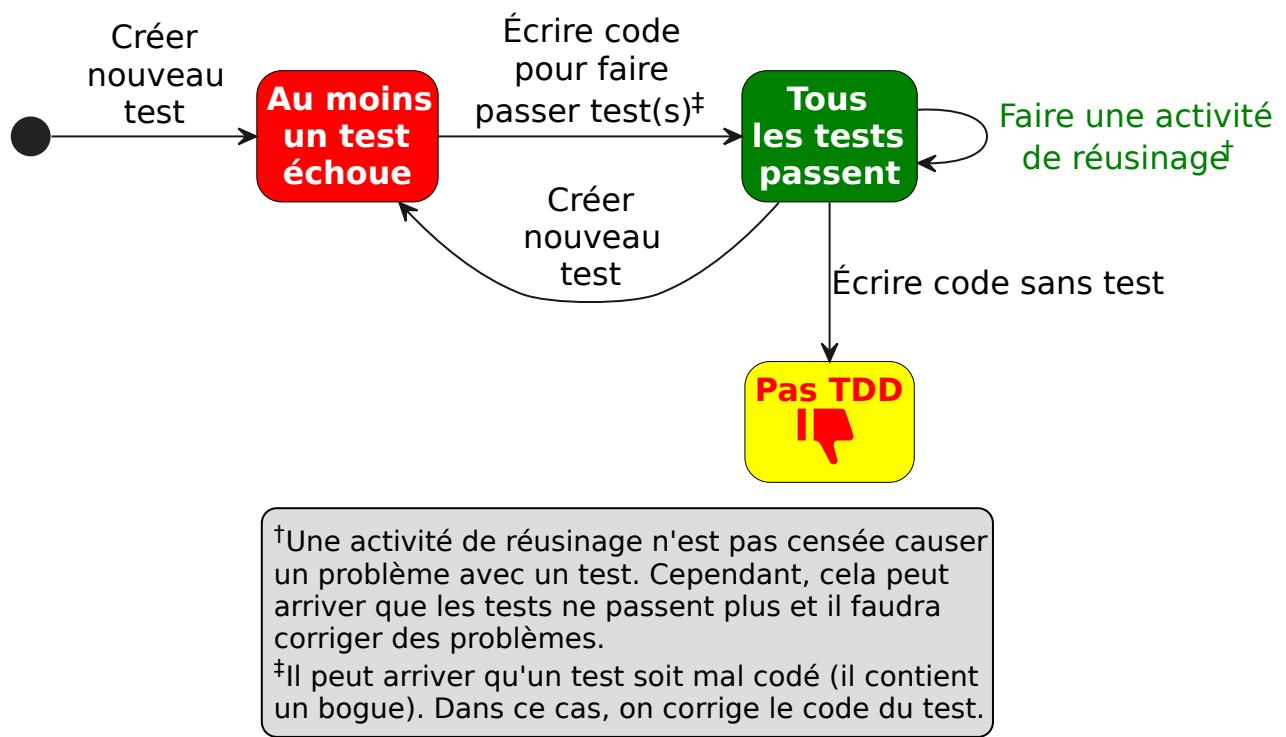


Figure 10.1: États du développement piloté par les tests. ([PlantUML](#))

10.1 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](#)  C'est un outil de transmission de techniques et de principes de combat.



Figure 10.2: 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.

10.1.1 Exemple de Kata FizzBuzz

L'inspiration de cet exercice vient de codingdojo.org.

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 de 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
```

10.1.1.1 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, jest, unittest, etc.). Pour un exemple qui fonctionne en TypeScript, vous pouvez cloner le code à [ce dépôt](#).

10.1.1.2 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 10.1). 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.

10.1.1.3 Kata pour FizzBuzz

Les étapes sont simples et précises. Il s'agit de créer une classe ayant une méthode acceptant un entier et retournant 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. Attention aux conflits dans les exigences:
 - 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. 59 → **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**).

11 Réusinage (Refactorisation)

11.1 Introduction

Considérez l'histoire suivante, provenant de la 2e édition du livre *Refactoring* de Fowler (2018):

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.

11 Réusinage (Refactorisation)

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** 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 ».

Le réusinage est une activité intégrale du **Développement piloté par les tests**.

11.2 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 <https://refactoring.com/catalog/> 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

Symptôme de mauvaise conception (« Smell »)	Activités de réusinage à appliquer éventuellement
Long Parameter List	Replace Parameter with Query, Preserve Whole Object, Introduce Parameter Object, Remove Flag Argument, Combine Functions into Class
Global Data	Encapsulate Variable
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

11.3 Automatisation du réusinage par les IDE

À la section F19.2/A22.2 du livre de Larman (2005), 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, WebStorm, 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.

11 Réusinage (Refactorisation)

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.

11.4 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.

12 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 et le temps que vous investissez pour maîtriser une technologie est important. Pourtant, il y a des risques avec certains investissements de temps à long terme. Par exemple, qui développe du code encore pour Flash W? Cette technologie est maintenant désuète et ça ne sert à rien de mentionner cette compétence sur un CV.

La bonne nouvelle est qu'une « technologie » ne changera jamais: le comportement humain. Donc, il est toujours rentable d'investir du temps pour mieux maîtriser cet aspect du développement. Les entreprises en technologies de l'information sont toujours à la recherche de développeurs qui ont également des compétences générales (« soft skills »).

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.

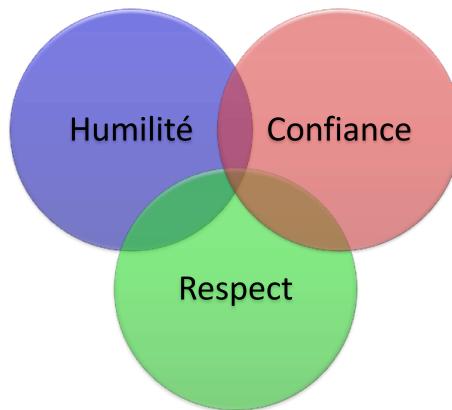


Figure 12.1: Pratiquement tout conflit social est dû à un manque d'humilité, de respect ou de confiance.

12.1 Humilité, Respect, Confiance

L'humilité, le respect et la confiance (voir la figure 12.1) sont les qualités de base pour le bon travail en équipe. Cette section présente ces aspects en détail.

12.1.1 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 ». Voir la section [Proposer des solutions si besoin](#).



Figure 12.2: Éviter d'être le « Centre de l'univers » (CC BY-NC-ND 2.0) par [Diamonddust](#).

Quelques exemples concrets d'humilité dans le développement:



Figure 12.3: « Missing » (CC BY-SA 2.0) par smkybear.

Équipe > Moi

Figure 12.4: Un coéquipier humble va accepter une décision prise par l'équipe, même s'il n'était pas en accord à 100%. ([PlantUML](#))

- Un coéquipier 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).
- 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!)

12.1.2 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.

12.1.3 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.

12.2 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.



Figure 12.5: *Facteur de bus* (nom): le nombre de personnes qui doivent être heurtées par un bus avant que votre projet ne soit complètement condamné. ([Fitzpatrick et Collins-Sussman 2012](#))

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 du trimestre, il n'est pas facile de maintenir le même rythme. Cependant, les enseignants et les chargés de laboratoire 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.

12.3 Mentorat

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 du trimestre 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'informations, la personne peut vous ignorer plutôt que vous dire gentiment qu'elle a compris.

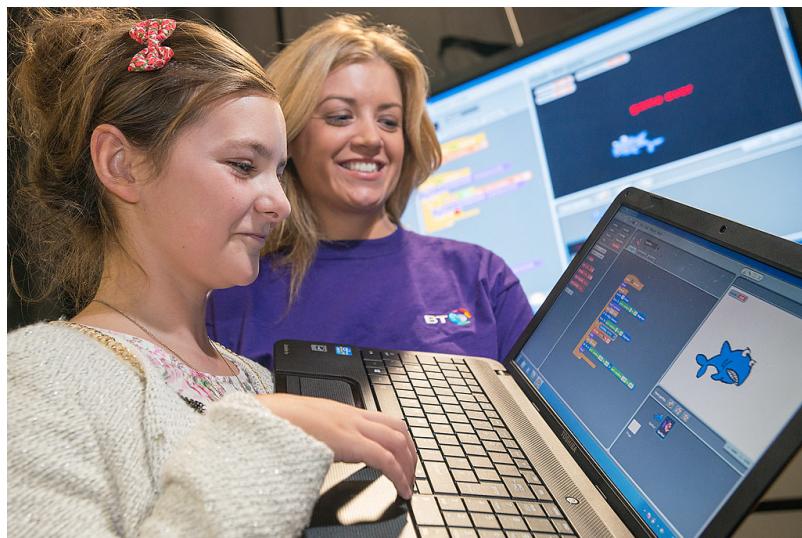


Figure 12.6: Encadrer les coéquipiers est une habileté à mettre sur son CV « CultureTECH BT Monster Dojo » (CC BY 2.0) par [connor2nz](#).

12.4 Scénarios

Considérez les volets HRC lorsque vous vous trouvez dans une des situations suivantes:

- 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 que les autres 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).
- un ou deux membres d'une équipe abandonnent le cours après les évaluation de mi-trimestre, par crainte d'échec.
- un coéquipier suit cinq (!) cours en même temps et n'a pas le temps adéquat pour travailler correctement dans les laboratoires de cette matière.
- 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 devez 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 [Évaluer les contributions des membres de l'équipe](#).

13 Outils pour la modélisation UML

Le chapitre F20/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.

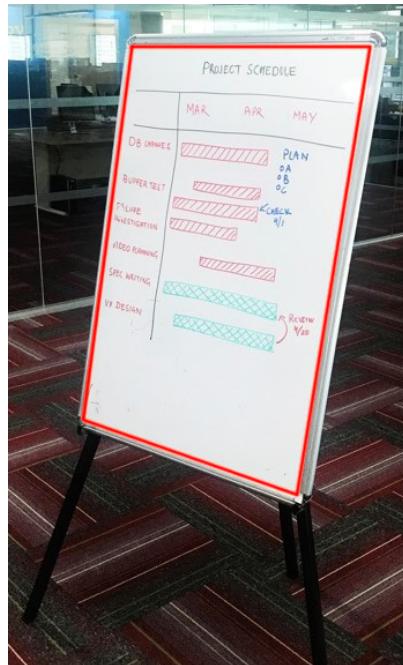


Figure 13.1: Office Lens peut détecter le cadre d'un dessin sur un tableau blanc ou papier et le transformer.

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

Dans la méthodologie de ce manuel, 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;

13 Outils pour la modélisation UML

- 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 13.2) avec [tutoriel](#) ▶;
 - 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 13.3)

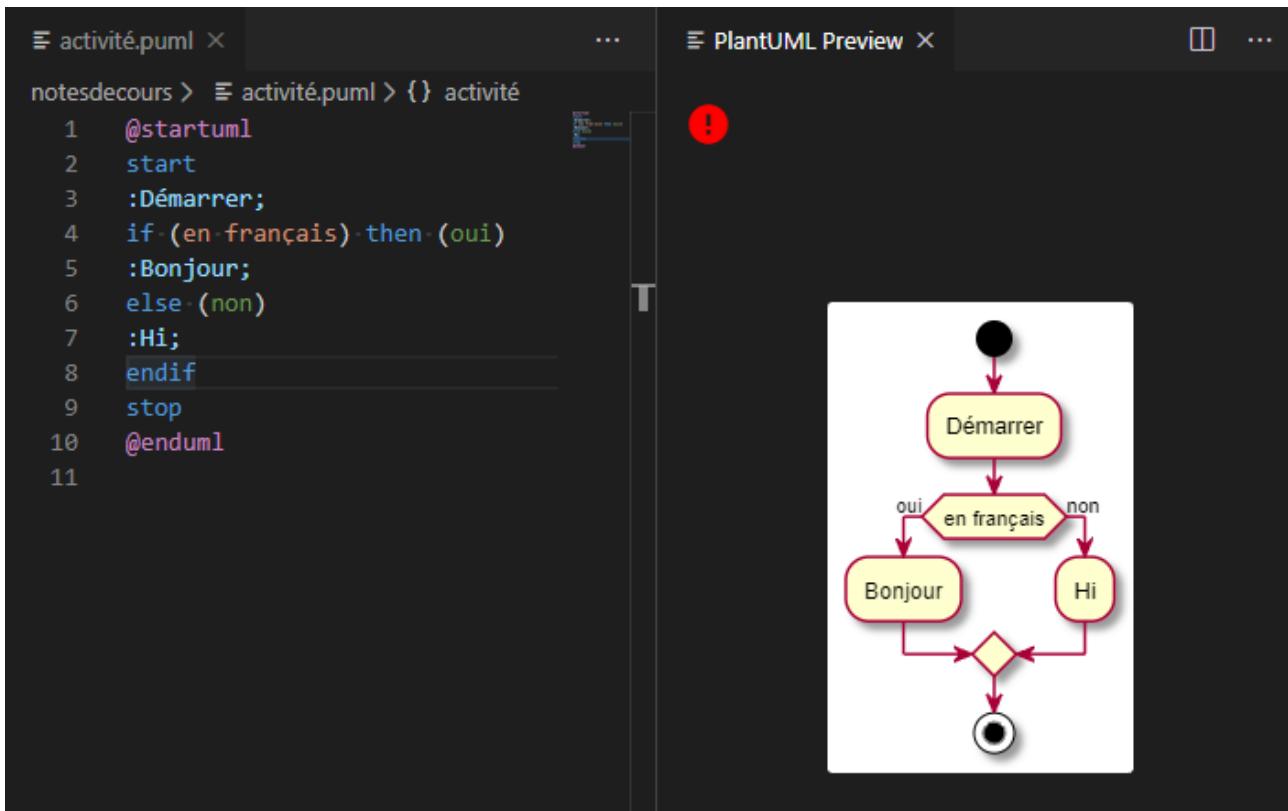


Figure 13.2: L'extension PlantUML pour VisualStudio Code.

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 13.5). Il est possible de corriger le diagramme, mais en combien de clics? C'est très vite tannant.

13.1 Exemples de diagramme avec PlantUML

Dans le menu « Select sample diagram » de PlantUML Gizmo (Google Docs), il y a plusieurs exemples de diagrammes utilisés dans le cadre de ce manuel et le livre de Larman (2005) (voir la figure 13.4).

13.2 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 Syp9J4vLqBLJSCffib9mB2t9ICqhoKnEBCdCprC8IYqiJIqkuGBAAUW2rJY256DHLLoGdr
@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 récupérer 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
```

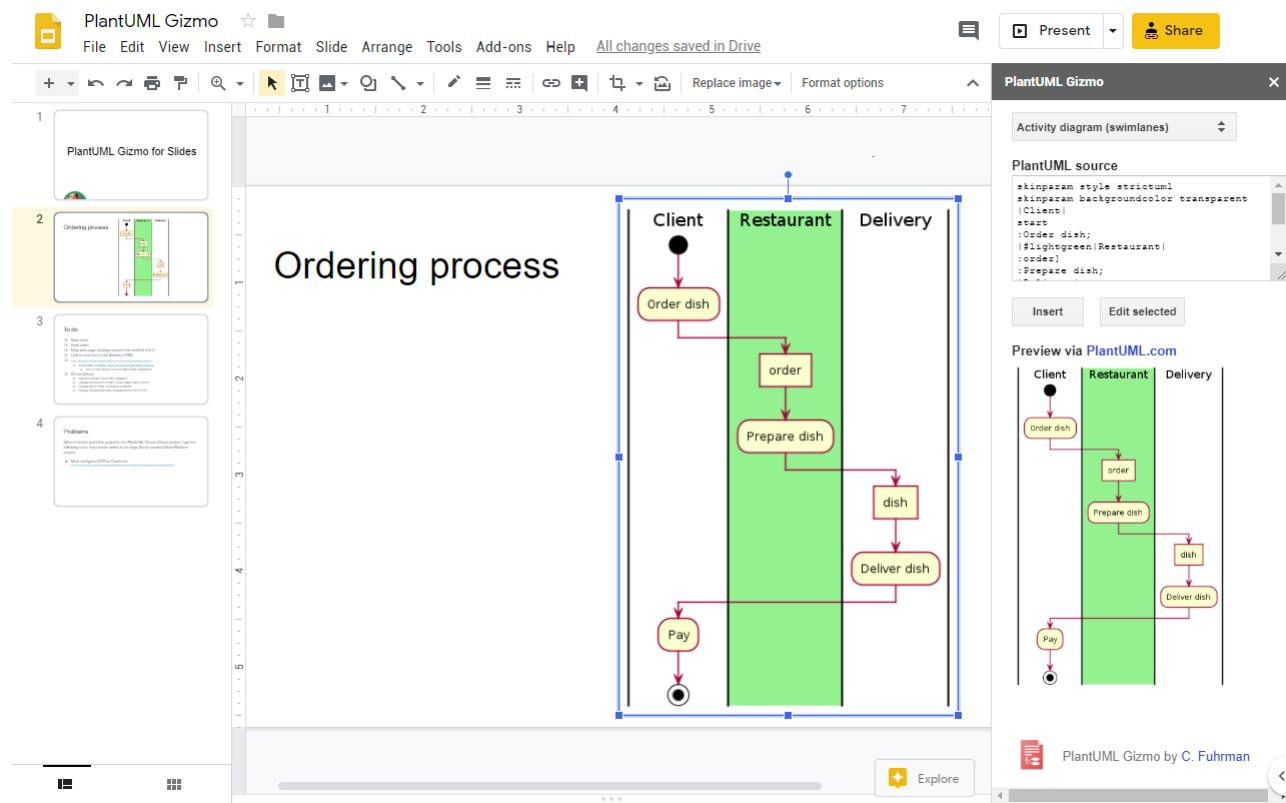


Figure 13.3: PlantUML Gizmo pour Google Docs et Google Slides.

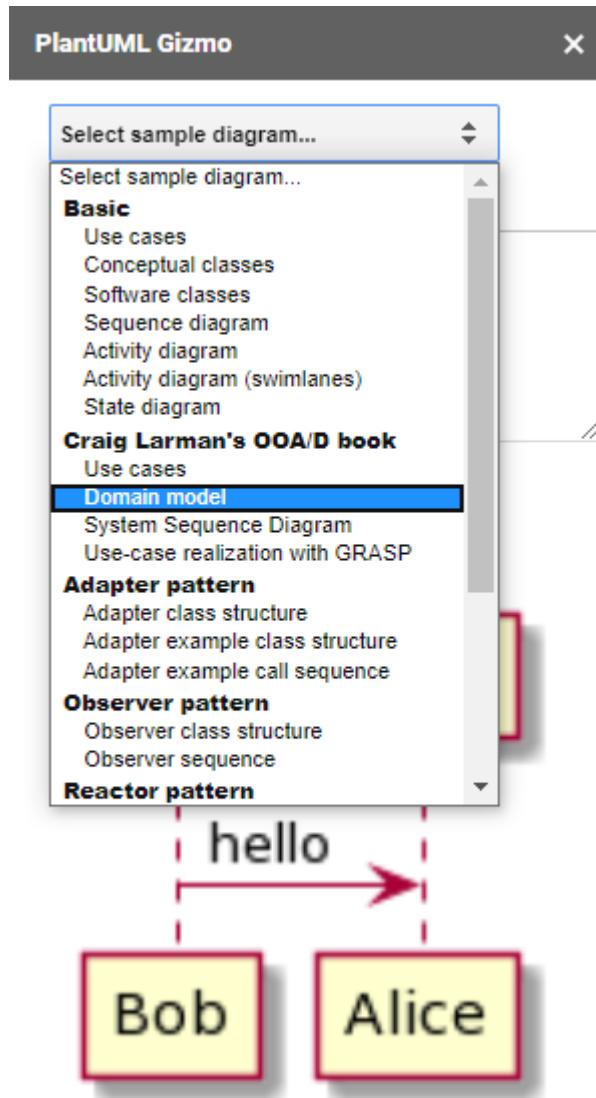


Figure 13.4: PlantUML Gizmo offre plusieurs exemples de diagramme UML.

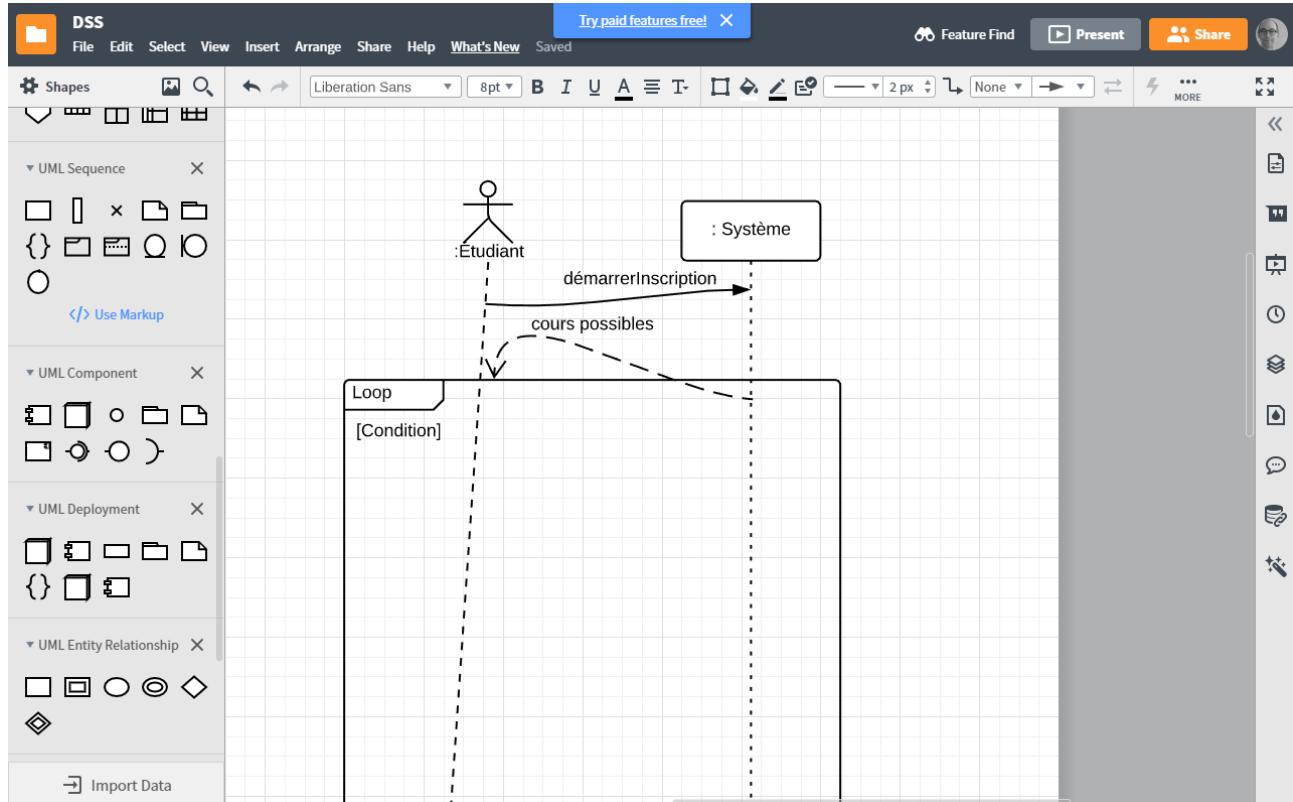


Figure 13.5: Exemple de tentative de créer un diagramme de séquence système (DSS) avec Lucidcharts. C'est principalement un éditeur graphique avec les éléments graphiques UML qui sont essentiellement des éléments graphiques composés. Il n'y a pas de sémantique UML dans l'outil. Par exemple, un « messages » 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.

14 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.

14.1 Exemple avec Adaptateur

Le chapitre A26/F23  présente l'exemple du pattern Adaptateur pour les calculateurs de taxes (figure 14.1 tirée du livre de Larman, Figure A26.3/F23.3).

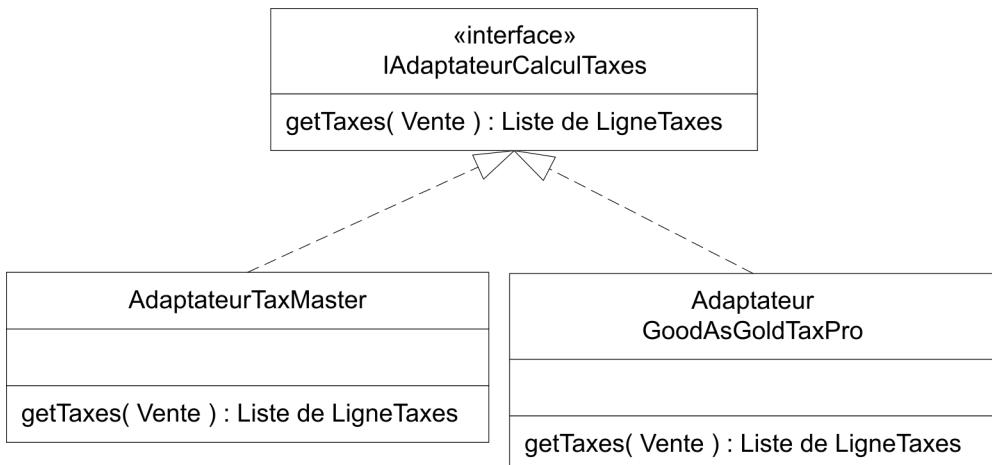


Figure 14.1: Le pattern Adaptateur.

14.2 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 Larman (2005) 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 14.2 (tirée du livre de Larman, Figure A26.3/F23.3) démontre la relation entre ces principes dans le cas d'Adaptateur.

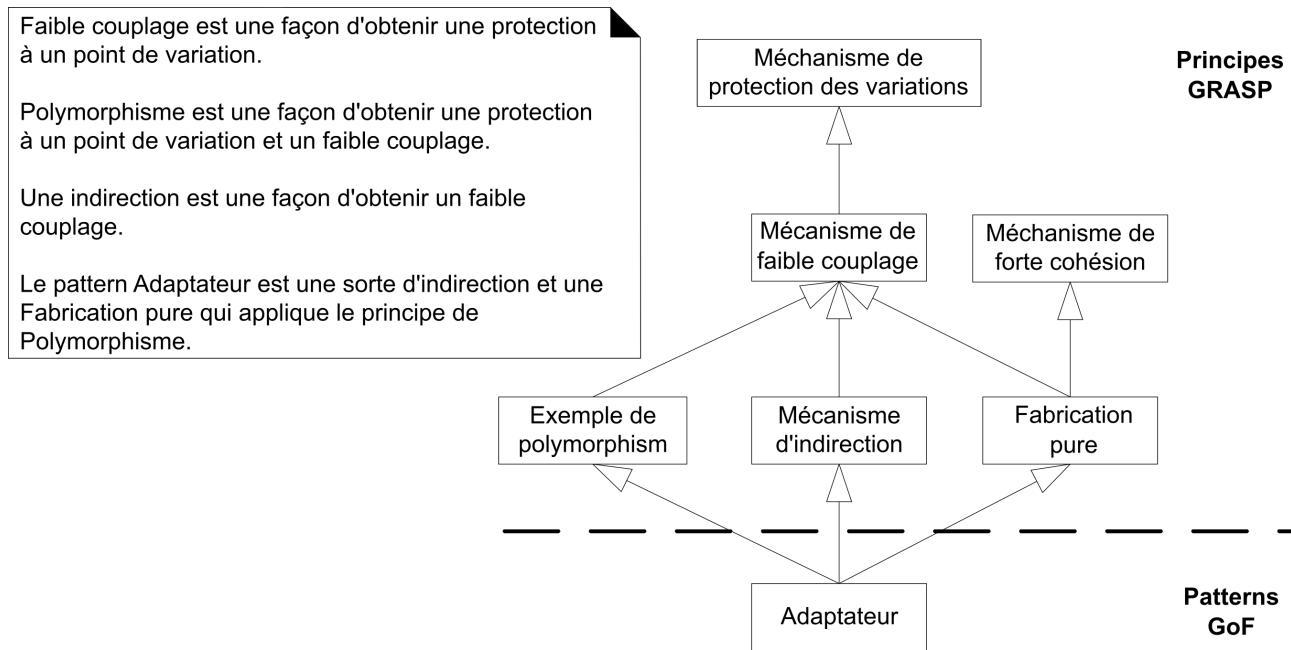


Figure 14.2: Adaptateur et principes GRASP.

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?

14.3 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 14.1 du pattern Adaptateur pour les sections suivantes.

14.3.1 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 ses implémentations) dans l'opération polymorphe `getTaxes()`.

14.3.2 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.

14.3.3 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.

14.3.4 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).

14.4 GRASP et réusinage

Il y a des liens entre les GRASP et les activités de Réusinage (Refactorisation). Alors, un IDE qui automatise les *refactorings* peut vous aider à appliquer certains 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](#).

15 Fiabilité

Le chapitre A35/F30  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.

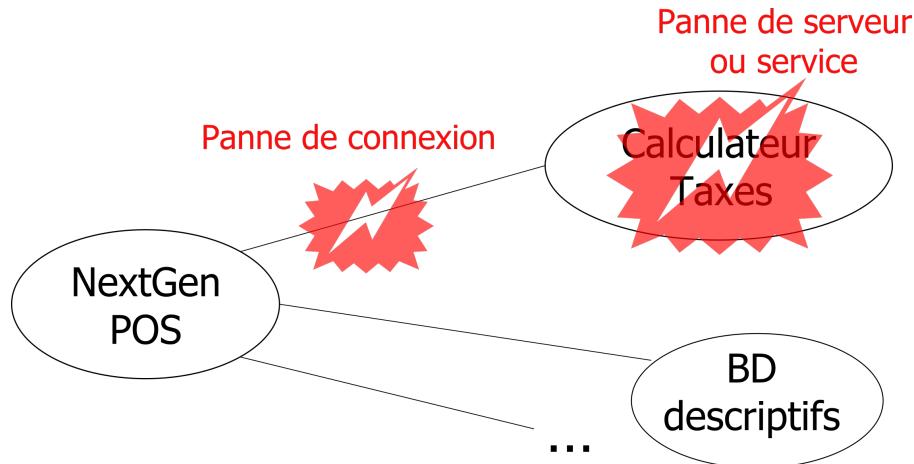


Figure 15.1: Comment tolérer une panne de connexion ou de service?

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 .
 - Utilisation de *Procuration (Proxy) de redirection [GoF]* pour basculer sur un service local en cas de panne (écriture d'information), A35.4/F30.4 

15 Fiabilité

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.

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 .

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
- noeud d'objet
- débranchement et jointure (parallélisme)
- décision et fusion (exclusion mutuelle)

La figure 16.1 présente un exemple de diagramme d'activité décrivant de manière générale une partie du processus de travail d'une personne utilisant `git` pour la gestion de code source.

16.1 Diagrammes de flux de données (DFD)

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

Un exemple de diagramme d'activité dans le cadre d'un cours de programmation utilisant GitHub Classrooms est dans la figure 16.2. 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 téléchargé la liste de classe *avant* d'envoyer les invitations aux étudiants.

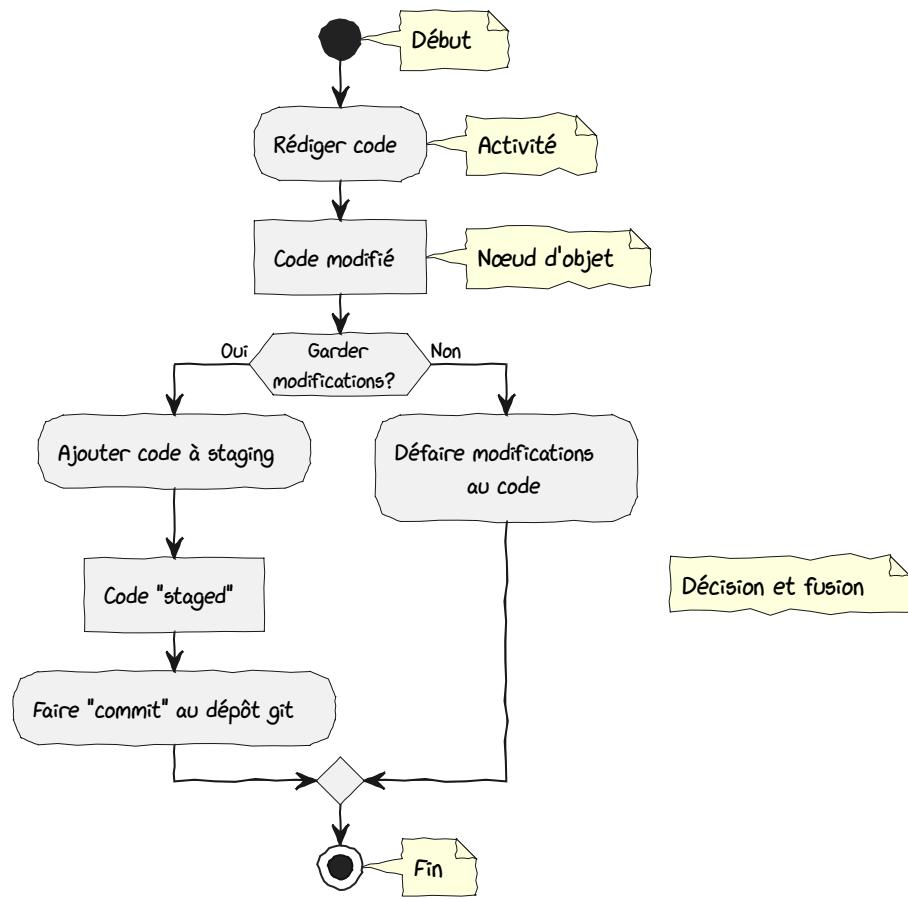


Figure 16.1: Diagramme d'activités pour un processus simple avec git. ([PlantUML](#))

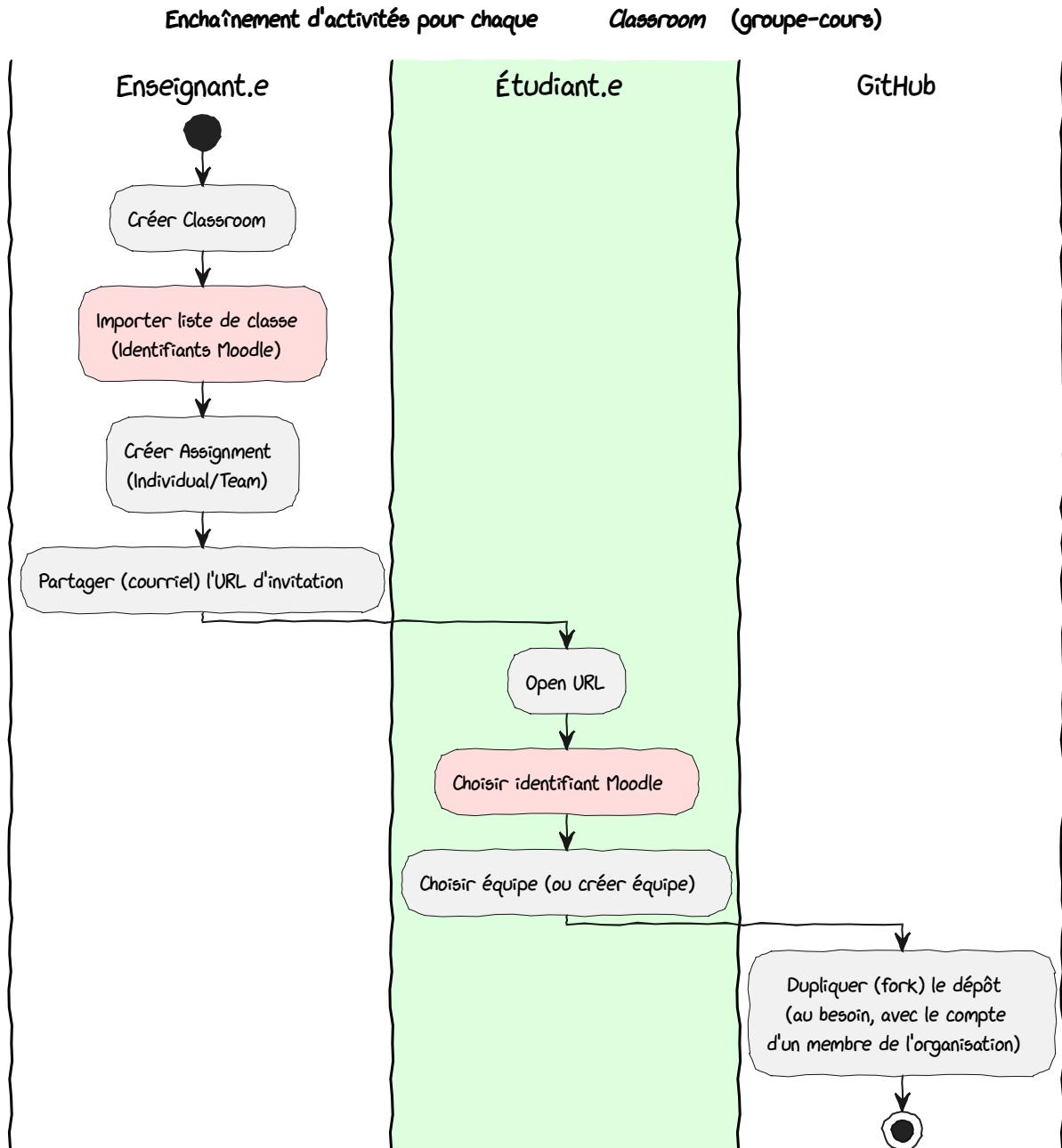


Figure 16.2: Diagramme d'activités pour les activités séquentielles de GitHub Classrooms. ([PlantUML](#))

17 Diagrammes d'état

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

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 W](#)
- 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 17.1 est un exemple tiré du livre de Larman (2005) et fait en PlantUML.

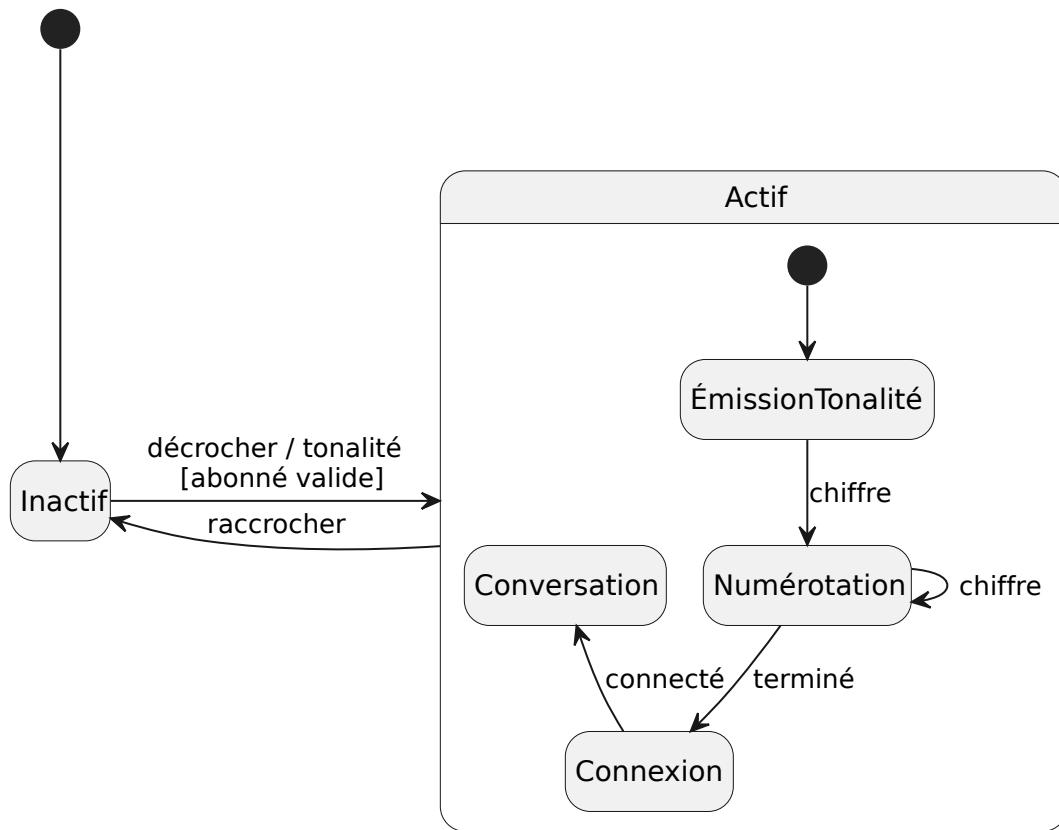


Figure 17.1: Diagramme d'états (figure A29.3/F25.10). ([PlantUML](#))

18 Conception de packages

Le chapitre A36/F29 [█](#) contient des directives pour la conception de packages. Notez que la notion de package n'existe pas en TypeScript (et Javascript), mais le principe de *namespace* existe. La Section [18.1](#) explique quelques pratiques pour la gestion des namespaces en TypeScript.

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 [18.1](#) est un exemple d'un diagramme de package.

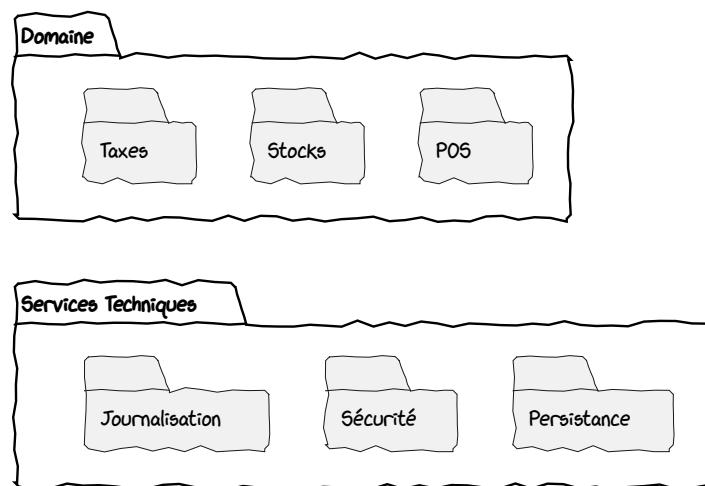


Figure 18.1: Diagramme de packages (tiré de la figure F12.6 [█](#)). ([PlantUML](#))

18.1 Absence de package dans TypeScript

En effet TypeScript n'a pas la notion de package comme dans C# ou Java. Cependant, il y a des pratiques pour organiser logiquement le code et pour éviter les conflits (les collisions) de noms. Rappelons que la notion de package existe dans Java pour:

1. **Organiser logiquement le code:** le type interface `java.util.List` est disponible dans la *bibliothèque java.util*)
2. **Éviter les conflits de nom:** les classes `java.util.List` et `ca.etsmtl.log121.fuhrman.projet2.List` ont le même nom de base, mais puisqu'elles sont dans deux packages différents, elles peuvent être utilisées dans le même programme (leur « fully qualified name » est différent)

En TypeScript, on peut atteindre les mêmes objectifs.

18.1.1 Organisation des éléments du code

L'organisation peut être réalisée grâce aux modules avec les mots-clés `export` et `import`. Par exemple:

```
// maClasse.ts
export class MaClasse {
    // définition
}

// client.ts
import { MaClasse } from './maClasse'
```

On organise les fichiers, e.g., `maClasse.ts` dans les répertoires.

18.1.2 Noms sans conflit

Dans l'exemple plus haut, il ne serait pas possible d'avoir deux fichiers nommés `maClasse.ts` dans le même répertoire, alors il est impossible d'avoir une collision avec le nom du fichier. Donc, on pourrait importer la classe `MaClasse` de `maClasse.ts` et la classe `MaClasse` de `lib/projet2/maClasse.ts` dans le même programme. Cependant, pour éviter un conflit de nom, on emploie le mot-clé `as` pour renommer la classe (`MaClasseP2`) lorsqu'on l'importe:

```
// client.ts
import { MaClasse } from './maClasse'
import { MaClasse as MaClasseP2 } from './lib/projet2/maClasse'
```

18.1.3 Namespaces

TypeScript offre une autre manière d'organiser et d'éviter les conflits de noms avec [namespaces](#) (anciennement les *modules internes*). L'[exemple de Validators](#) est intéressant puisqu'il s'agit d'un namespace commun réparti dans plusieurs fichiers. C'est à utiliser surtout lorsqu'on ne veut pas centraliser tout le code dans un seul (gros) fichier (avec `export`). Mais comme vous voyez dans l'exemple avec les commentaires dans le code de certains fichiers, e.g., `/// <reference path="Validation.ts" />`, il est plus compliqué à maintenir.

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 .

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.

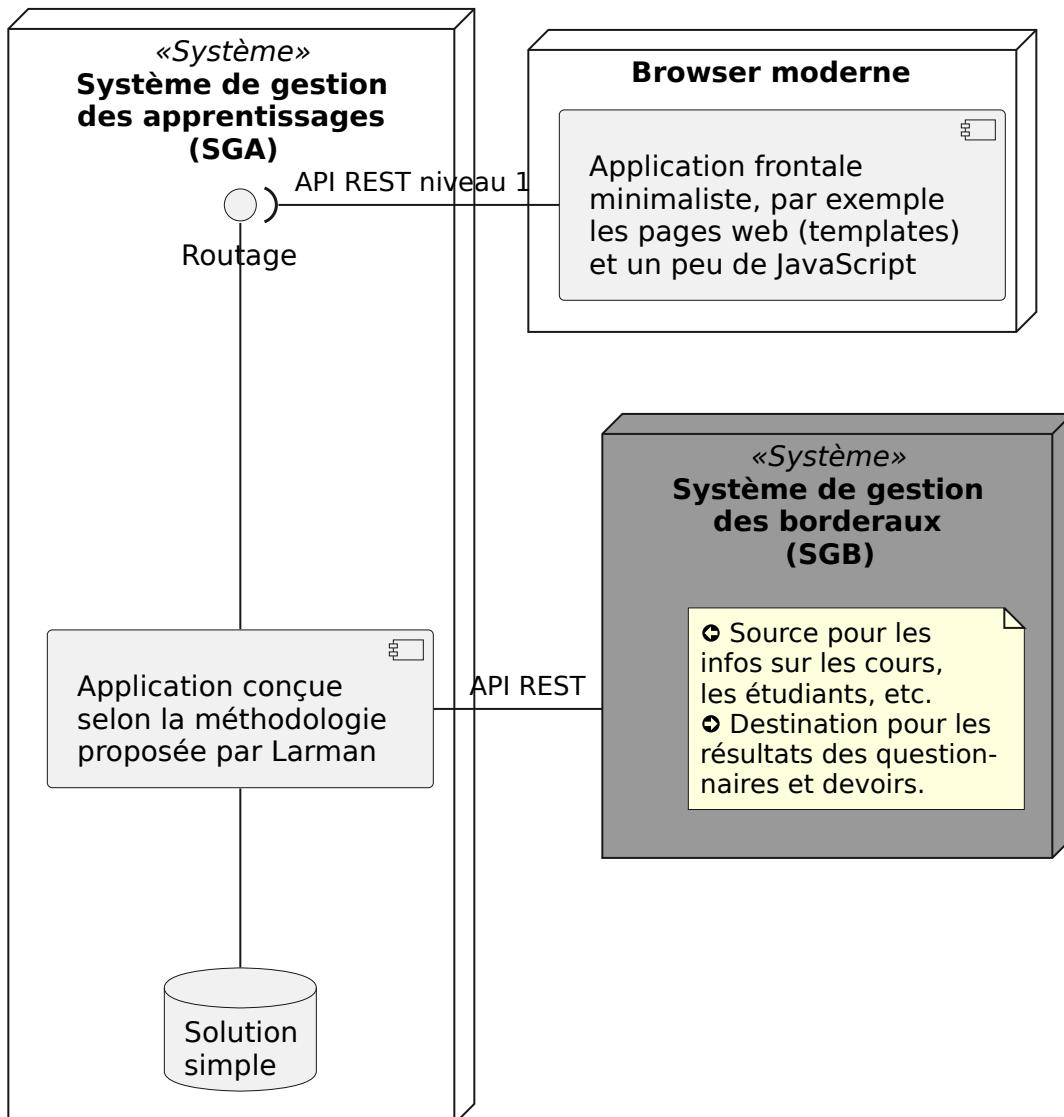


Figure 19.1: Diagramme de déploiement du système à développer pour le laboratoire. ([PlantUML](#))

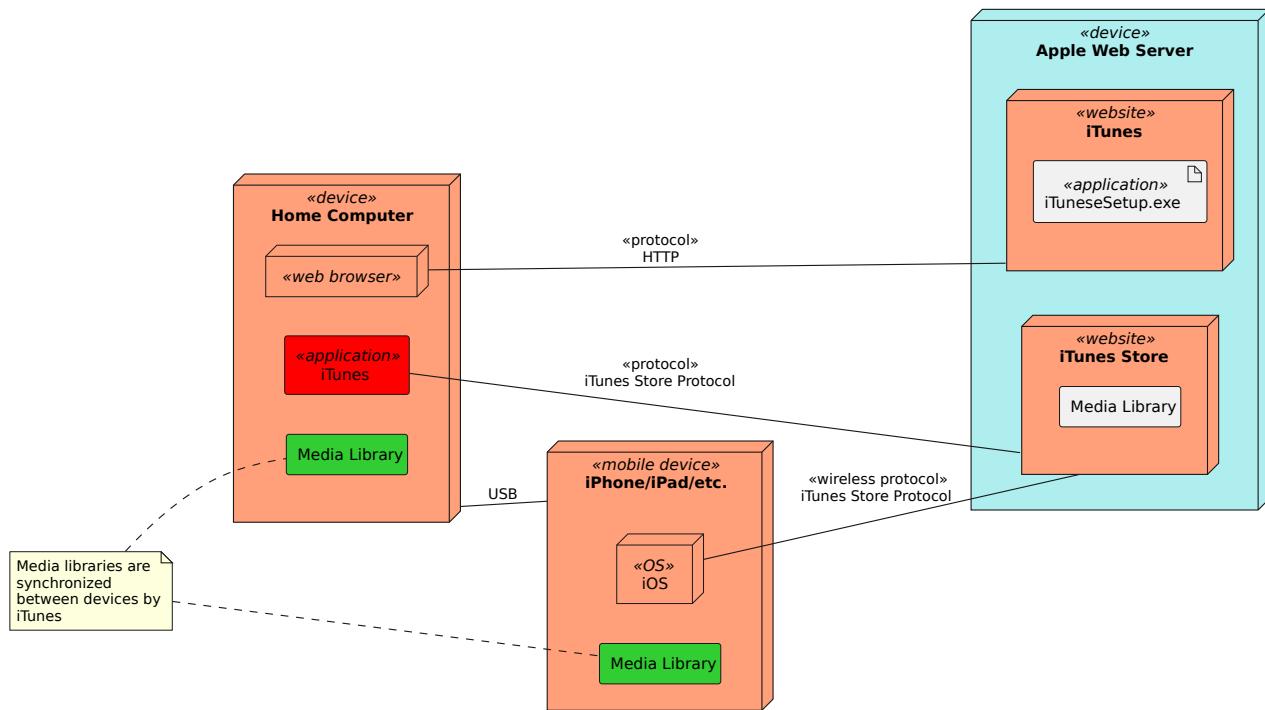


Figure 19.2: Diagramme de déploiement pour iTunes d'Apple, inspiré de [ceci](#). (PlantUML)

20 Laboratoires

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

20.1 JavaScript/TypeScript

Pour la personne ayant déjà des connaissances Java (de LOG121), il est recommandé d'apprendre les choses dans cet ordre:

- **JavaScript** - un tutoriel intéressant (et libre) est sur fr.javascript.info. Je vous recommande de contribuer à des traductions en français sur GitHub.
- **TypeScript** - ce tutoriel est en anglais, mais il est adapté à des personnes ayant déjà une expérience en Java/C#.

Voici des points importants pour le projet de laboratoire, organisés pour quelqu'un ayant déjà des connaissances en Java:

- TypeScript se traduit (« emit ») en JavaScript, alors il faut comprendre le moteur d'exécution JavaScript.
- Pour convertir une chaîne en nombre, pour lire ou écrire un fichier sur disque, etc., on utilise des opérations en JavaScript.
- Un *type* en TypeScript est comme *un ensemble de valeurs* plutôt qu'une définition hiérarchique. En Java, il n'est pas possible d'avoir un type pouvant être soit une chaîne soit un nombre. Mais en TypeScript, c'est facile de déclarer un type comme une *union* de plusieurs types, p. ex. `string | number`.
- JavaScript a des notions de « *truthy* » et « *falsy* » (conversion d'une expression à une valeur booléenne) permettant de vérifier avec moins de code si une variable est définie ou initialisée, etc.
- L'opérateur d'égalité stricte (`==`) (sans conversion de type)
- Les fonctions fléchées (*fat arrow functions* en anglais)
- Le traitement asynchrone en JavaScript
 - Promesses et `async/await`
- Les services REST (GET vs PUT)
- Environnement de test (Jest)
- Les gabarits (templates) PUG (anciennement Jade) : [Tutoriel \(court\)](#), [Tutoriel \(plus complet\)](#)
- Bootstrap (mise en page avec CSS) : [Tutoriel \(attention, il faut appliquer les éléments dans les gabarits PUG\)](#)

Le **Lab 0** aborde plusieurs de ces aspects, mais certaines notions sont plus complexes et nécessitent une étude approfondie. Le but de cette section est de donner des tutoriels plus spécifiques. Enseigner la syntaxe ou les principes du langage TypeScript n'est pas le but de ce manuel, mais apprendre à trouver l'information soi-même est essentiel pour une personne travaillant dans les technologies de l'information.

Il y a un [dépôt d'exemples avec TypeScript \(utilisant ts-node pour les voir facilement\)](#) sur GitHub. Il y a un exemple qui montre comment faire des REST à partir de TypeScript avec le système SGB.

20.2 JavaScript: Truthy et Falsy (conversion en valeur booléenne)

JavaScript offre un mécanisme simple pour vérifier des valeurs dans une expression `if`. Imaginez l'exemple suivant:

```
let maVariable;

// d'autres instructions...

if (maVariable != undefined
    && maVariable != null
    && maVariable != '') {
    // on peut faire quelque chose avec maVariable ...
}
```

On vérifie trois possibilités pour `maVariable` avant de l'utiliser. Ce genre de situation arrive souvent en JavaScript, puisque les objets peuvent prendre les valeurs différentes selon le contexte. Il serait bien de pouvoir réduire la quantité de code dans ces cas.

Grâce à la notion de conversion de valeur selon la règle de « truthy » et « falsy », JavaScript permet de simplifier les instructions en une seule condition, sans ET (`&&`), en convertissant la valeur de `maVariable` en booléenne `true` ou `false`:

```
// conversion booléenne selon la règle de "truthy" et "falsy"
if (maVariable) {
    // on peut faire quelque chose avec maVariable ...
}
```

Il faut comprendre la règle de conversion en valeur booléenne selon « truthy » et « falsy ». En fait, il est plus simple de commencer par les valeurs se traduisant en `false` (« falsy »), car tout ce qui ne l'est pas est donc `true` (« truthy »).

20.2.1 Falsy

Les valeurs suivantes se convertissent en `false` dans une condition:

- `false`
- `null`
- `undefined`
- `0` (attention, c'est parfois un piège)
- `NaN` (not a number)
- `''` ou `""` (chaîne vide)

20.2.2 Truthy

Tout ce qui n'est pas converti en `false` (expliqué ci-dessus) est converti en `true`. En voici quelques exemples:

- `{}` (objet vide)
- `[]` (tableau vide)
- `-20`
- etc.

N'oubliez pas que la valeur de `0` est « falsy » dans une condition. C'est souvent un piège en JavaScript quand on considère les variables qui peuvent avoir une valeur numérique. Par exemple, si on fait `if (maVariable)` pour tester si une variable est définie, si la variable est définie et a sa valeur est `0`, la condition sera `false`.

20.3 Évaluer les contributions des membres 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 les laboratoires de ce manuel 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/b5b098470e7ec4536c35ca1ce3592853 \
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 les 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.

20.3.1 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 doit indiquer explicitement ce fait, même avec des pourcentages**.

Une phrase vague comme « certains ont travaillé plus que d'autres » est une formulation diplomatique, mais elle n'est pas assez explicite et n'est pas une résolution proactive du problème le cas échéant.

20.3 Évaluer les contributions des membres de l'équipe

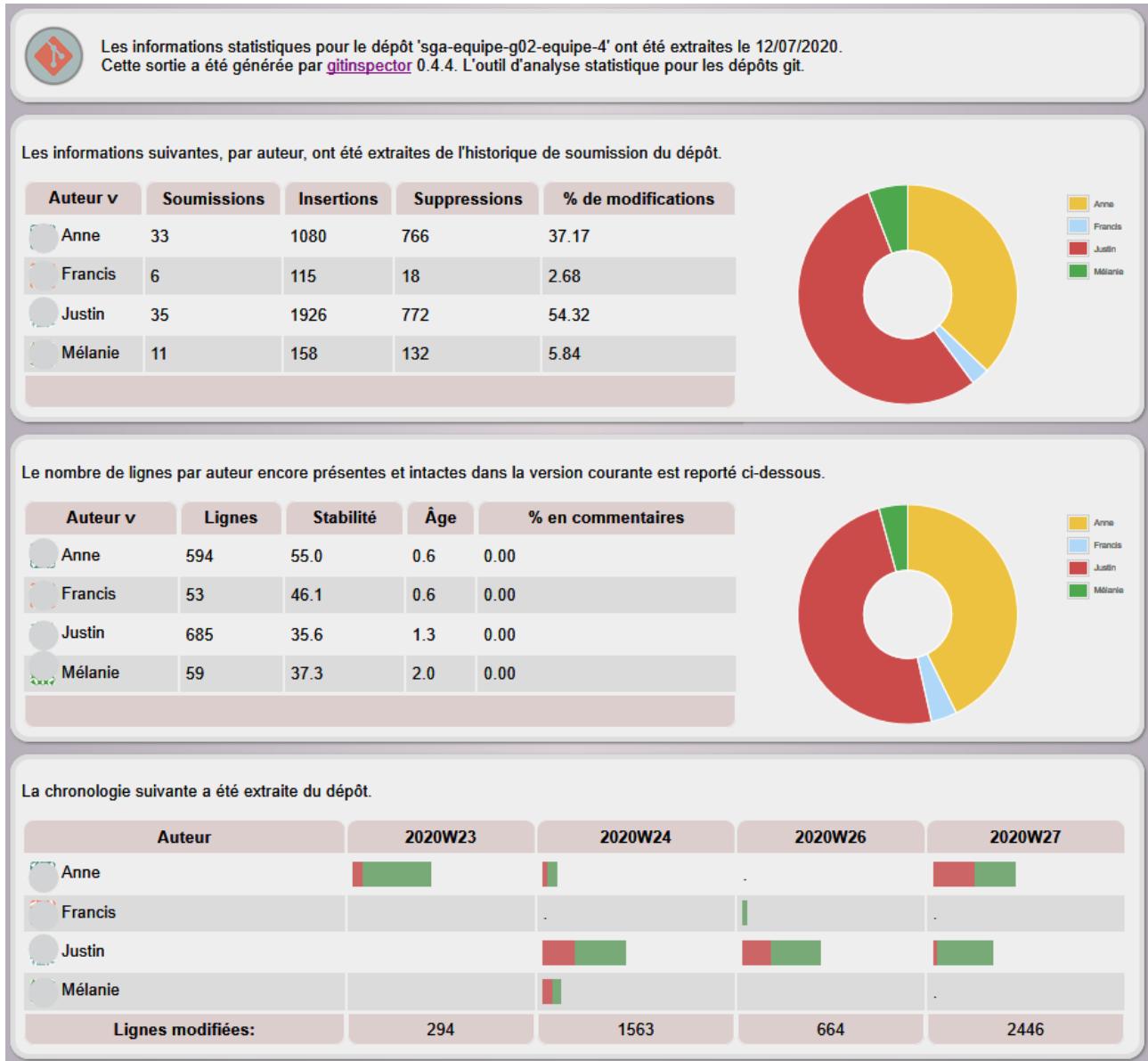


Figure 20.1: Exemple de rapport généré par [gitinspector](#).

20.3.2 Proposer des solutions si besoin

Une inégalité importante dans les contributions est un signal d'alarme. On doit agir, mais on commence par poser des questions, par exemple:

- Est-ce que Francis et Mélanie sont à l'aise avec les technologies utilisées dans le lab, ont-ils besoin de coaching?
- Sont-ils des « parasites » ou « mollassons » ([Oakley et al. 2004](#)) ([traduction française de l'article](#))? À certaines universités, le plan de cours vous permet d'exclure leurs noms du rapport (et ils auront une note de zéro pour la remise), mais **seulement s'ils n'ont rien fait du tout** (ce qui n'est pas le cas dans l'exemple ci-dessus). Une personne exclue de cette manière va probablement abandonner le cours et vous perdrez définitivement un coéquipier.
- Est-ce que Anne et Justin ont laissé suffisamment de liberté aux autres pour faire une contribution importante? Font-ils assez confiance aux autres?
- Avez-vous fait un plan d'itération assez détaillé pour que chacun puisse contribuer adéquatement? Dans l'exemple ci-dessus, peut-être Francis et Mélanie ont-ils trouvé ça difficile de savoir où contribuer?
- Est-ce que tout le monde assiste aux séances de laboratoire?
- Est-ce que tout le monde travaille *au moins 6 heures en dehors des séances encadrées*?
- Est-ce que certains coéquipiers travaillent excessivement, sans donner la chance aux autres de contribuer? N'oubliez pas que les laboratoires sont une manière d'apprendre à pratiquer la matière de ce manuel. Laisser un ou deux coéquipiers faire plus de travail peut nuire à la valeur pédagogique des laboratoires (ça peut faire mal à l'examen final pour ceux qui ont moins contribué). Il y a aussi un risque sur le plan de la **Redondance des compétences dans l'équipe** (**Bus Factor**), surtout si un coéquipier qui travaille beaucoup plus que les autres éprouve un problème d'épuisement à cause du fait qu'il travaille trop.
- Est-ce que tout le monde utilise un moyen de communiquer de manière synchrone et asynchrone (Slack, Discord, Teams, etc.)? Le courriel n'est pas l'outil idéal pour coordonner un travail en équipe.
- etc.

Dans le bilan il faut *constater les faits et proposer des solutions* pour éviter des inégalités importantes sur le plan de la contribution dans les prochaines itérations. Ainsi, vous gérez les problèmes de manière plus proactive.

20.3.3 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 shortlog -se | sed "s/^.*@\t//" > .mailmap
```

Ensuite, modifiez le fichier `.mailmap` pour respecter ce format:

```
Prénom Nom Désirés <courriel> Prénom Nom Non-Désirés <courriel>
```

20.3 Évaluer les contributions des membres de l'équipe

Par exemple, soit le `.mailmap` initial qui contient quatres entrées pour le même auteur:

```
C. Fuhrman <christopher.fuhrman@etsmtl.ca>
Christopher (Cris) Fuhrman <christopher.fuhrman@etsmtl.ca>
Christopher Fuhrman <christopher.fuhrman@etsmtl.ca>
Cris Fuhrman <fuhrmanator+git@gmail.com>
```

On décide de garder l'alias `C. Fuhrman <christopher.fuhrman@etsmtl.ca>` pour chaque nom:

```
C. Fuhrman <christopher.fuhrman@etsmtl.ca>
C. Fuhrman <christopher.fuhrman@etsmtl.ca> Christopher (Cris) Fuhrman <christopher.fuhrman@etsmtl.ca>
C. Fuhrman <christopher.fuhrman@etsmtl.ca> Christopher Fuhrman <christopher.fuhrman@etsmtl.ca>
C. Fuhrman <christopher.fuhrman@etsmtl.ca> Cris Fuhrman <fuhrmanator+git@gmail.com>
```

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=( "C. Fuhrman" "Benjamin Le" "Yvan Ross" )
```

Q: J'ai une autre question...

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

Bibliographie

- Avgeriou, Paris, Philippe Kruchten, Ipek Ozkaya, et Carolyn Seaman. 2016. « Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162) ». Édité par Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, et Carolyn Seaman. *Dagstuhl Reports* 6 (4): 110-38. <https://doi.org/10.4230/DagRep.6.4.110>.
- Coad, Peter. 1997. *Object Models: Strategies, Patterns, and Applications*. 2nd Revised ed. edition. Upper Saddle River, N.J: Pearson Technology Group.
- Fitzpatrick, Brian W., et Ben Collins-Sussman. 2012. *Team Geek: A Software Developer's Guide to Working Well with Others*. 1 edition. Sebastopol, CA: O'Reilly Media.
- Ford, Neal. 2009. « Evolutionary architecture and emergent design: Investigating architecture and design ». {CT}316. <http://www.ibm.com/developerworks/library/j-eaed1/>.
- Fowler, Martin. 2007. « Bliki: DesignStaminaHypothesis ». martinfowler.com.
- . 2018. *Refactoring: Improving the Design of Existing Code*. 2 edition. Boston: Addison-Wesley Professional.
- Gamma, Erich, Richard Helm, Ralph Johnson, et John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1^{re} éd. Reading, Mass: Addison-Wesley Professional. <http://amazon.com/o/ASIN/0201633612/>.
- Hanmer, Robert. 2007. *Patterns for Fault Tolerant Software*. 1^{re} éd. Chichester, England ; Hoboken, NJ: Wiley.
- Karac, I., et B. Turhan. 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, Craig. 2005. *UML 2 et design patterns*. 3^e éd. Paris: Village Mondial.
- Oakley, Barbara, RM Felder, R Brent, et I Elhajj. 2004. « Coping with Hitchhikers and Couch Potatoes on Teams ». *Journal of Student Centered Learning* 2 (1): 32-34. <https://www.cs.cornell.edu/courses/cs3110/2018fa/teams/hitchhikers.html>.

