



Analyse et conception de logiciels

Christopher Fuhrman

Yvan Ross

05 janvier 2023 à 19:29:07 +00:00

Table des matières

Préface	1
Pourquoi une ressource éducative libre ?	1
Prérequis	3
Livre complémentaire	3
Remerciements	4
Sources du manuel	5
Licence	5
I. Théorie	6
1. Analyse et conception de logiciels	7
1.1. Analyse vs Conception	7
1.2. Décalage des représentations	9
1.3. La complexité et ses sources	9
1.4. Survol de la méthodologie	11
1.5. Développement itératif, évolutif et agile	13
2. Besoins (exigences)	19
2.1. FURPS+	19
3. Cas d'utilisation	22
3.1. Exemple : Le jeu Risk	22
4. Modèle du domaine (MDD, modèle conceptuel)	25
4.1. Classes conceptuelles	25
4.2. Attributs	26
4.3. Associations	27
4.4. Exemple de MDD pour le jeu Risk	29
4.5. Attributs dérivés	29
4.6. Classes de « description » et de catalogues	30
4.7. Classes d'association	32
4.8. Affinement du MDD	33
4.9. FAQ MDD	33

4.10. Exercices	34
5. Diagrammes de séquence système (DSS)	37
5.1. Exemple : DSS pour Attaquer un pays	37
5.2. Les DSS font abstraction de la couche présentation	38
5.3. FAQ DSS	40
5.4. Exercices	42
6. Principes GRASP	43
6.1. Spectre de la conception	43
6.2. Tableau des principes GRASP	46
6.3. GRASP et RDCU	47
6.4. GRASP et patterns GoF	48
6.5. Exercices	48
7. Dette technique	49
7.1. Origine	49
7.2. Nuances de la dette technique	51
7.3. Résumé	53
8. Contrats d'opération	54
8.1. Les contrats en bref	54
8.2. Exemple : Contrats d'opération pour Attaquer un pays	56
8.3. Feuille de référence	58
8.4. Exercices	58
9. Réalisations de cas d'utilisation (RDCU)	59
9.1. Spécifier le contrôleur	60
9.2. Satisfaire les postconditions	60
9.3. Visibilité	62
9.4. Transformer identifiants en objets	63
9.5. Utilisation d'un tableau associatif (<code>Map<clé, objet></code>)	64
9.6. RDCU pour l'initialisation, le scénario Démarrer	65
9.7. Réduire le décalage des représentations	65
9.8. Pattern « Faire soi-même »	67
9.9. Exercices	68
10. Développement piloté par les tests	69
10.1. Kata TDD	70
11. Réusinage (Refactorisation)	74
11.1. Introduction	74
11.2. Symptômes de la mauvaise conception - Code smells	75
11.3. Automatisation du réusinage par les IDE	77

Table des matières

11.4. Impropriété	77
12. Développement de logiciels en équipe	78
12.1. Humilité, respect, confiance	79
12.2. Redondance des compétences dans l'équipe (bus factor)	81
12.3. Mentorat	83
12.4. Scénarios	83
13. Outils pour la modélisation UML	85
13.1. Exemples de diagrammes avec PlantUML	87
13.2. Astuces PlantUML	87
14. Décortiquer les patterns GoF avec GRASP	91
14.1. Exemple avec Adaptateur	91
14.2. Imaginer le code sans le pattern GoF	91
14.3. Identifier les GRASP dans les GoF	92
14.4. GRASP et réusinage	95
14.5. Exercices	96
15. Fiabilité	98
15.1. Exercices	99
16. Diagrammes d'activités	100
16.1. Diagrammes de flots de données (DFD)	100
16.2. Exercices	103
17. Diagrammes d'états	105
17.1. Exercices	107
18. Conception de packages	109
18.1. Absence de packages dans TypeScript	110
19. Diagrammes de déploiement et de composants	112
19.1. Diagrammes de déploiement	112
20. Laboratoires	115
20.1. TypeScript	115
20.2. JavaScript/TypeScript	115
20.3. JavaScript: Truthy et Falsy (conversion en valeur booléenne)	116
20.4. Git	118
20.5. Évaluer les contributions des membres de l'équipe	119
Bibliographie	124

II. Exercices	125
21. Critique d'une conception	126
22. Coder des méthodes à partir des diagrammes de séquence	130
 Annexes	 135
A. Cas d'utilisation - Réserver un livre de la bibliothèque	135
B. Cas d'utilisation - Traiter une vente	137
B.1. DSS	137
B.2. MDD partiel	138
C. Cas d'utilisation - Ouvrir la caisse	139
C.1. Terminologie	139
C.2. Cas d'utilisation : Ouvrir la caisse	140
C.3. Modèle du domaine partiel	141
C.4. Diagramme de séquence système (DSS)	143
C.5. Contrats d'opération	144

Liste des figures

1.1.	Diagramme de <i>classes conceptuelles</i> décrivant le <i>problème</i> d'un jeu de dés (adapté du Jeu de dés de Larman, 2005, Chapitre 1). Ceci est élaboré lors d'une activité d'analyse.	8
1.2.	Diagramme de <i>classes logicielles</i> décrivant une <i>solution</i> au problème du jeu de dés. La conception s'inspire du modèle du problème, afin de faciliter sa compréhension.	8
1.3.	« Complexity » (CC BY-SA 2.0) par lytfyre.	10
1.4.	Sources de complexité.	12
1.5.	Survol de la méthodologie.	14
1.6.	Un processus itératif permet de gérer les complexités, car la planification d'une itération peut viser une partie du système et le système évolue à mesure que les itérations avancent. (Cette œuvre, « Modèle du processus itératif », est un dérivé de « Iterative development model » de Krupadeluxe, utilisé sous CC BY-SA 4.0. « Modèle de processus itératif » est sous licence CC BY-SA 4.0 par Christopher Fuhrman.)	15
1.7.	Processus itératif et incrémental (évolutif).	16
1.8.	La rétroaction et la rectification itératives font diminuer l'instabilité des exigences à mesure que le projet avance, car le système à développer converge vers les spécifications et la conception les plus appropriées (adaptation de Larman, 2005, fig. 2.2).	16
1.9.	Pour respecter la méthode du temps limité, on peut modifier les objectifs d'une itération si le travail est trop important (adaptation de Larman, 2005, fig. 2.4).	18
2.1.	Ramasser les besoins non fonctionnels ? « Dog Clean Up » de Luis Prado, utilisé selon CCo	20
3.1.	Cinq dés utilisés dans le jeu Risk. Par Val42 - https://en.wikipedia.org/wiki/Image:Risk-dice-example.jpg , CC BY-SA 3.0 Link.	23
3.2.	Diagramme de cas d'utilisation. (PlantUML)	24
4.1.	Modèle du domaine du jeu Risk. (PlantUML)	29
4.2.	<code>nbPaysOccupés</code> est un attribut dérivé et sa valeur sera calculée selon le nombre de pays de l'association. (PlantUML)	30
4.3.	Cours joue le rôle de description d'entités (les groupes-cours).	31
4.4.	Erreur fréquente: utiliser une classe de description sans justification.	31
4.5.	Classe d'association dans le MDD Jeu Risk. (PlantUML)	32
5.1.	Diagramme de séquence système pour <i>Attaquer un pays</i> . (PlantUML)	38
5.2.	Une opération système dans un DSS. C'est une abstraction. (PlantUML)	38

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)	39
6.1.	Spectre de la conception, adapté de Ford (2009). (PlantUML)	44
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)	50
8.1.	Pendant l'opération système <code>créerNouvelleVente</code> , une instance de Vente doit être créée. Le contrat d'opération le spécifie dans une postcondition.	55
8.2.	Les postconditions décrivent la manipulation d'objets dans un MDD (la partie inférieure ici est un diagramme d'objets).	57
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)	61
9.2.	Combiner la création d'une instance et une modification de son attribut dans un constructeur. (PlantUML)	62
9.3.	L'objet <i>b</i> doit être visible à l'objet <i>a</i> si <i>a</i> veut lui envoyer un message. (PlantUML) . .	63
9.4.	Un identifiant <code>idClient:String</code> est transformé en objet <code>c:Client</code> , qui est ensuite envoyé à la Vente en cours. (PlantUML)	64
9.5.	Exemple de l'utilisation d'un tableau associatif pour trouver une Case Monopoly à partir de son nom. (PlantUML)	64
9.6.	Exemple de l'initialisation partielle du jeu Risk. (PlantUML)	66
9.7.	Faire soi-même : « Moi, objet logiciel, je fais moi-même ce qu'on fait normalement à l'objet réel dont je suis une abstraction » de Coad (1997).	67
10.1.	États du développement piloté par les tests. (PlantUML)	70
10.2.	Étudiante de karaté faisant le kata <i>Bassai Dai</i> (photo « Karate » (CC BY-SA 2.0) par The Consortium).	71
12.1.	Pratiquement tout conflit social est dû à un manque d'humilité, de respect ou de confiance.	79
12.2.	Éviter d'être le « Centre de l'univers » (CC BY-NC-ND 2.0) par Diamonddust. . . .	80
12.3.	« Missing » (CC BY-SA 2.0) par smkybear.	80
12.4.	Un(e) membre de l'équipe humble va accepter une décision prise par l'équipe, même s'il ou elle n'est pas en accord à 100%. (PlantUML)	80
12.5.	Savoir encadrer les membres de l'équipe est une habileté à mettre sur son CV. « Cultu-reTECH BT Monster Dojo » (CC BY 2.0) par connor2nz.	84

Liste des figures

13.1. Microsoft Lens peut détecter le cadre d'un dessin sur un tableau blanc ou sur papier et le transformer, même si l'on n'est pas droit devant le dessin.	86
13.2. L'extension PlantUML pour Visual Studio Code.	87
13.3. PlantUML Gizmo pour Google Docs et Google Slides.	88
13.4. PlantUML Gizmo offre plusieurs exemples de diagrammes UML.	89
13.5. Exemple de tentative de créer un diagramme de séquence système (DSS) avec Lucidchart. 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 « message » UML dans un diagramme de séquence dans Lucidchart 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 Lucidchart ne connaît pas cette sémantique, vous risquez de perdre beaucoup de temps à faire des diagrammes UML avec ce genre d'outil.	90
14.1. Le pattern Adaptateur.	91
14.2. Adaptateur et principes GRASP (voir Larman, 2005, fig. A26.3/F23.3)	93
15.1. Comment tolérer une panne de connexion ou de service ?	98
16.1. Diagramme d'activité pour un processus simple avec Git. (PlantUML)	101
16.2. Diagramme d'activités pour les activités séquentielles de GitHub Classroom (contexte de l'ÉTS avec Moodle). (PlantUML)	102
17.1. Diagramme d'états (figure A29.3/F25.10). (PlantUML)	106
18.1. Diagramme de packages (tiré de la figure F12.6). (PlantUML)	109
19.1. Diagramme de déploiement du système à développer pour le laboratoire. (PlantUML)	113
19.2. Diagramme de déploiement pour iTunes d'Apple, inspiré de ceci. (PlantUML)	114
20.1. Concepts et opérations de base de Git. (Cette œuvre, « Concepts et opérations de base de Git », est un dérivé de « Basic git concepts and operations » de Costa Shulyupin, utilisé sous EPL.	118
20.2. Exemple de rapport généré par <code>gitinspector</code>	120
21.1. Diagramme de classes logicielles (TypeScript) pour le projet [Emojiopoly].(https://github.com/Chuzzy/Emojiopoly)	124
21.2. MDD (version française) de Monopoly proposé par Larman (2005)	128
21.3. MDD (version anglaise) de Monopoly proposé par Larman (2005)	129
22.1. Exemple de diagramme de séquence.	130
B.1. DSS pour le scénario Traiter une vente de Larman (2005).	137

Liste des figures

C.1.	Diagramme de cas d'utilisation pour le système NextGen.	139
C.2.	Plateau-billets et tiroir-caisse. Capture d'écran de la vidéo sur YouTube.	140
C.3.	Modèle du domaine partiel du système POS NextGen avec deux nouvelles classes conceptuelles MisePlateau et PlateauBillets, qui modélisent des éléments du cas d'utilisation « Ouvrir la caisse ».	142
C.4.	Diagramme de séquence système (DSS) pour le scénario « Ouvrir la caisse ».	143

Liste des tableaux

4.1.	Extrait du tableau 9.1 	25
4.2.	Extrait du tableau 9.2 	(liste d'associations courantes) 28
6.1.	Patterns (principes) GRASP	46
7.1.	Classification de la dette selon Fowler (2009)	52

Préface

Cet ouvrage a commencé comme notes de cours pour le cours *Analyse et conception de logiciel* (*LOG210*) dans les programmes de baccalauréat en génie logiciels (LOG) et en génie des technologies de l'information (GTI) de l'École de technologie supérieure (ÉTS) à Montréal, au Québec. À l'origine, la communauté étudiante dans ces programmes provenait des collèges d'enseignement général et professionnel (cégeps), ayant donc déjà reçu un diplôme d'études collégiales (DEC) dans un programme technique. Ainsi, ces personnes ont déjà appris à programmer dans un langage orienté objet. En effet, cette démarche correspond à l'approche « computer science first » (CS-first) pour enseigner le génie logiciel (Ardis et coll., 2015).

LOG210 a été mis sur pied au début des années 2000, et le livre obligatoire a toujours été le fameux *Applying UML and Patterns* de Craig Larman, qui en était à sa 2^e édition (2001) à l'époque. En 2005, la 3^e édition a été publiée avec plusieurs traductions, notamment celle en français (2005). Pendant plus de dix ans, le livre de Larman a été fort apprécié par les personnes étudiantes.

Mais pour le corps enseignant donnant le cours, le manque d'exercices dans le livre a toujours été un gros inconvénient. En plus, certains sujets (comme les cas d'utilisation, les préconditions des contrats d'opération et les diagrammes de communication) dans le livre de Larman sont moins pertinents en industrie aujourd'hui, puis d'autres sujets (comme le développement piloté par les tests, le travail en équipe et les cadriels Web) sont devenus plus importants. Une 4^e édition n'a jamais été publiée. Finalement, la traduction française du livre est en rupture de stock depuis 2019. Le présent manuel essaie de pallier tous ces problèmes, sous forme de ressource éducative libre (REL).

Pourquoi une ressource éducative libre ?

Aujourd'hui, les logiciels libres sont très répandus. Dans le cadre de l'enseignement, nous utilisons les dépôts de code source libre comme GitHub et GitLab, car ces plateformes permettent d'évaluer et de valider des logiciels ainsi que de collaborer dans les communautés de logiciel libre. Il est donc naturel d'imaginer une forme analogique pour le contenu de ce manuel, soit une **ressource éducative libre (REL)**, qui est également développée à travers un dépôt libre (GitHub dans ce cas).

Pour favoriser la réutilisation, nous avons utilisé plusieurs logiciels libres :

- Quarto (2022) (le langage source du texte est [Markdown](#)) ;
- Pandoc (2022) et LaTeX (2022), qui font partie des composantes exploitées par Quarto ;
- PlantUML (2022) pour les figures sous forme de texte (faciles à actualiser).

Préface

Autant que possible, les figures sont créées dans une forme vectorielle (comme SVG) plutôt que matricielle (comme JPEG) afin que le texte dans les figures soit « indexable » par les moteurs de recherche.

Finalement, il y a une tendance à faire des REL pour favoriser l'accès à l'information et pour permettre une meilleure inclusivité des personnes apprenantes et enseignantes.

Prérequis

Le contenu de ce manuel est organisé pour les personnes ayant déjà une base et une expérience avec :

- la programmation dans un langage orienté objet (Java, C#, C++, Python, TypeScript, etc.) ;
- les concepts de modélisation orientée objet (les classes, les interfaces, les instances, l'héritage, la composition, le polymorphisme, etc.) ;
- les tests unitaires (avec un cadriel comme JUnit) ;
- l'application des patrons de conception de la « bande des quatre » (Gang of Four) : Gamma, Helm, Johnson et Vlissides Gamma, Helm, Johnson, & Vlissides ([1994](#)).

La notation UML (*Unified Modeling Language*) est utilisée partout dans ce manuel. Nous faisons l'hypothèse que les personnes suivant ce manuel ont déjà vu cette notation avant, mais la familiarité avec UML n'est pas un préalable.

Livre complémentaire

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](#)). Le livre est encore populaire et pertinent, mais, malheureusement, il n'a pas été actualisé depuis sa 3^e édition (en 2005). De plus, la traduction française du livre n'est plus en stock au Québec depuis plusieurs années, et beaucoup de choses ont évolué depuis bientôt vingt ans !

Note

Au besoin, des références au livre de Larman sont indiquées par l'icône du livre . Puisqu'il est disponible en français et en anglais, et qu'il y a des différences avec les numéros de chapitres, nous indiquons une référence avec F et A pour signifier la langue du livre. Par exemple, la matière sur les principes GRASP est dans le chapitre 17 dans la version en anglais. Cependant, à cause des fusions de contenu lors de la traduction, la matière sur GRASP est dans le chapitre 16 dans la version en français. Dans ce cas, à cause des différences de chapitres, **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.

Prérequis

Mise en garde

Si vous avez une autre édition, comme la 2^e du livre en anglais ou même une des premières impressions de la 3^e du livre en anglais, les chapitres ne sont pas toujours les mêmes, et vous devrez chercher le sujet dans la table des matières.

Cependant, dans ce manuel, 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.

Remerciements

Comme beaucoup de ressources libres, ce manuel existe grâce à une collaboration avec de nombreuses personnes. La [fabriqueREL](#) (ressources éducatives libres) a soutenu financièrement et pédagogiquement l'élaboration du manuel lors de la période 2022-2023. Les conseils avec la dimension *Creative Commons* de la licence du contenu ont été fort utiles. Nous remercions les membres du suivi du projet dans le cadre de ce travail pris en charge par la fabriqueREL de leur soutien :

- Marianne Dubé, conseillère pédagogique à l'Université de Sherbrooke et coordonnatrice de la fabriqueREL ;
- Mouna Moumene, bibliothécaire à l'École de technologie supérieure (ÉTS) ;
- Marjolaine Lewis, conseillère pédagogique et technopédagogique à l'École de technologie supérieure (ÉTS) ;
- Claude Potvin, conseiller en formation à l'Université Laval ;
- Stéphane Roux, directeur général du service de soutien à la formation à l'Université de Sherbrooke ;
- Serge Allary, vice-recteur adjoint aux études de l'Université de Sherbrooke.

La fabriqueREL est également une vitrine de rayonnement importante pour les REL au Québec. Elle permet de valoriser le travail réalisé par un membre du corps professoral dans la rédaction d'un manuel libre pour un cours universitaire. Cela favorise une évolution positive dans la culture universitaire au Québec.

Nous tenons aussi à remercier chaleureusement les auxiliaires d'enseignement et les personnes ayant suivi le cours *Analyse et conception de logiciels (LOG210)* des programmes de baccalauréat en génie logiciel et en génie des technologies de l'information à l'ÉTS. Leurs suggestions et leurs commentaires constructifs ont permis d'améliorer les versions antérieures de ce manuel qui ont été utilisées dans le cadre de ce cours.

Nous remercions les membres du comité de révision (2023) pour leurs rétroactions constructives :

- Taki Eddine Seghiri, étudiant à la maîtrise en génie logiciel à l'ÉTS ;
- Mouna Moumene, bibliothécaire à l'École de technologie supérieure (ÉTS) ;

— Roberto Erick Lopez-Herrejon, professeur au département de génie logiciel et des TI à l'ÉTS.

Finalement, nous remercions Katerine Robert, réviseure linguistique à K. R. Révision, pour la révision du manuel.

Sources du manuel

Ce manuel est écrit en Markdown, et les sources sont sur GitHub à <https://github.com/fuhrmanator/1og210-ndc-quarto>. Les versions Web, PDF et EPUB ont été générées par le logiciel Quarto. Pour en savoir plus sur Quarto, visitez <https://quarto.org/docs/books>.

Licence

À déterminer (le texte est toujours un brouillon)

Première partie. Théorie

1. Analyse et conception de logiciels

Comment développer un logiciel dans un domaine où l'équipe de développement n'a pas forcément une expérience importante ? Comment faire une bonne conception d'un tel logiciel ? Ce manuel vise à décrire une méthodologie de développement de logiciels qui répond à ces questions.

Nous suivrons une approche systématique permettant de modéliser un domaine d'application et de concevoir un logiciel avec des abstractions (des classes) plus intuitives. Au fond, il s'agit de faire une *analyse* d'un domaine d'application et proposer une *conception* du logiciel à développer dans le domaine visé.

Puisqu'une conception doit être réalisée (codée) pour être validée, nous appliquerons une approche de programmation pilotée par les tests (*test-driven development* ou *TDD* en anglais). Le TDD est un élément essentiel dans le [Devops](#), une approche permettant de livrer et de faire évoluer rapidement les logiciels de manière robuste.

Parallèlement, surtout pour faire face à la complexité, la méthodologie propose des activités en itération ayant une durée relativement courte (1 à 3 semaines), dans lesquelles on réalise un sous-ensemble du système à développer et on le présente au client pour avoir sa rétroaction. C'est une approche dite « agile », ayant une base pédagogique neutre, nommée le « processus unifié ».

Dans ce chapitre, nous présentons en détail ces dimensions du développement de logiciels.

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 des 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 la 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.

1. Analyse et conception de logiciels

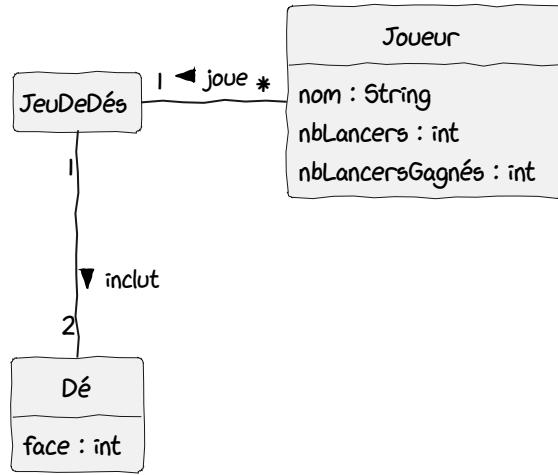


FIGURE 1.1. – Diagramme de *classes conceptuelles* décrivant le *problème* d'un jeu de dés (adapté du Jeu de dés de Larman, 2005, Chapitre 1). Ceci est élaboré lors d'une activité d'analyse.

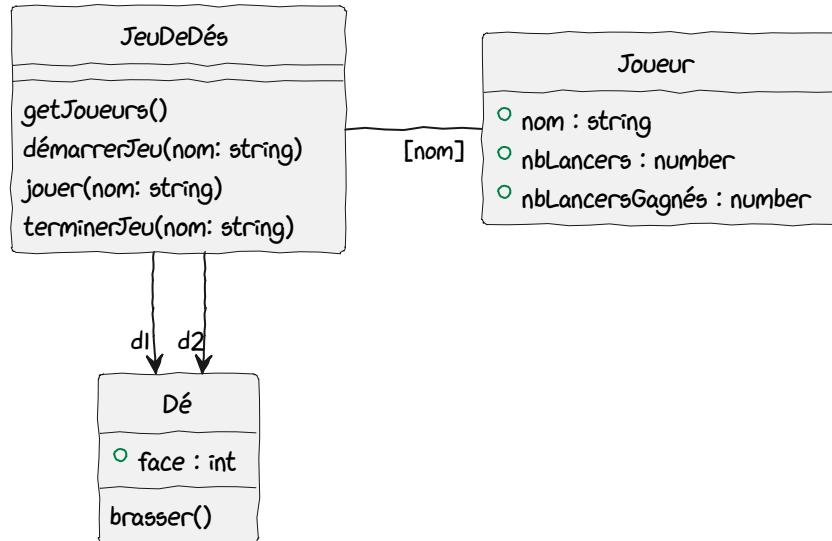


FIGURE 1.2. – Diagramme de *classes logicielles* décrivant une *solution* au problème du jeu de dés. La conception s'inspire du modèle du problème, afin de faciliter sa compréhension.

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 la représentation est proche de la modélisation du problème comme à la figure 1.2).

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 donc 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 il 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 le nom est difficile à relier 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 une 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 :

1. Analyse et conception de logiciels

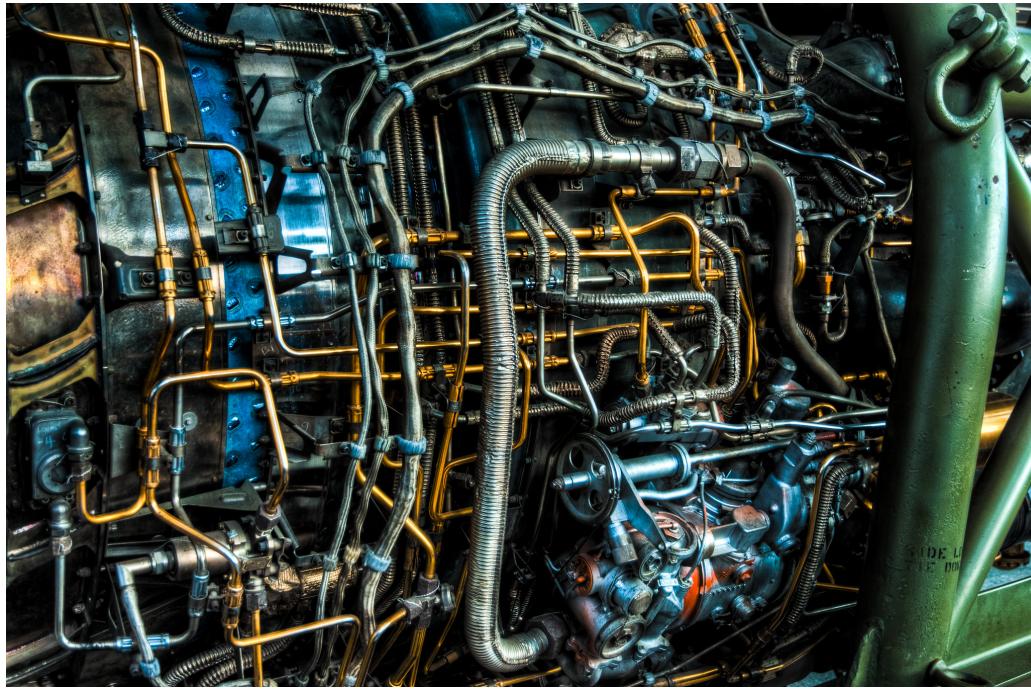


FIGURE 1.3. – « Complexity » (CC BY-SA 2.0) par lytfyre.

- 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.
- 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 le codage, le débogage et la maintenance d'une implémentation d'un algorithme complexe seront plus coûteux.
- Un *patron de conception* peut être complexe, par exemple les patrons Visiteur, Décorateur, Médiateur, etc., des GoF (1994). Un patron définit des rôles et parfois du code et des classes supplémentaires à créer. Le tout doit s'intégrer dans un design existant (qui a son propre niveau de complexité).
- 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 de tailles différentes, de l'alimentation 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 choisissant soigneusement un cadriel Web ou une architecture logicielle. La complexité circonstancielle peut aussi être due à des contraintes imposées sur la conception, comme l'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 *conteneurs* (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.

1.4. Survol de la méthodologie

La méthodologie d'analyse et de conception proposée dans ce manuel se base sur celle présentée par Larman (2005).

Voici les éléments importants documentés dans ce manuel (voir la figure 1.5, qui est une adaptation de plusieurs figures de Larman (2005)):

- Il y a une spécification explicite des besoins (Chapitre 2) dans le modèle de cas d'utilisation ;

1. Analyse et conception de logiciels

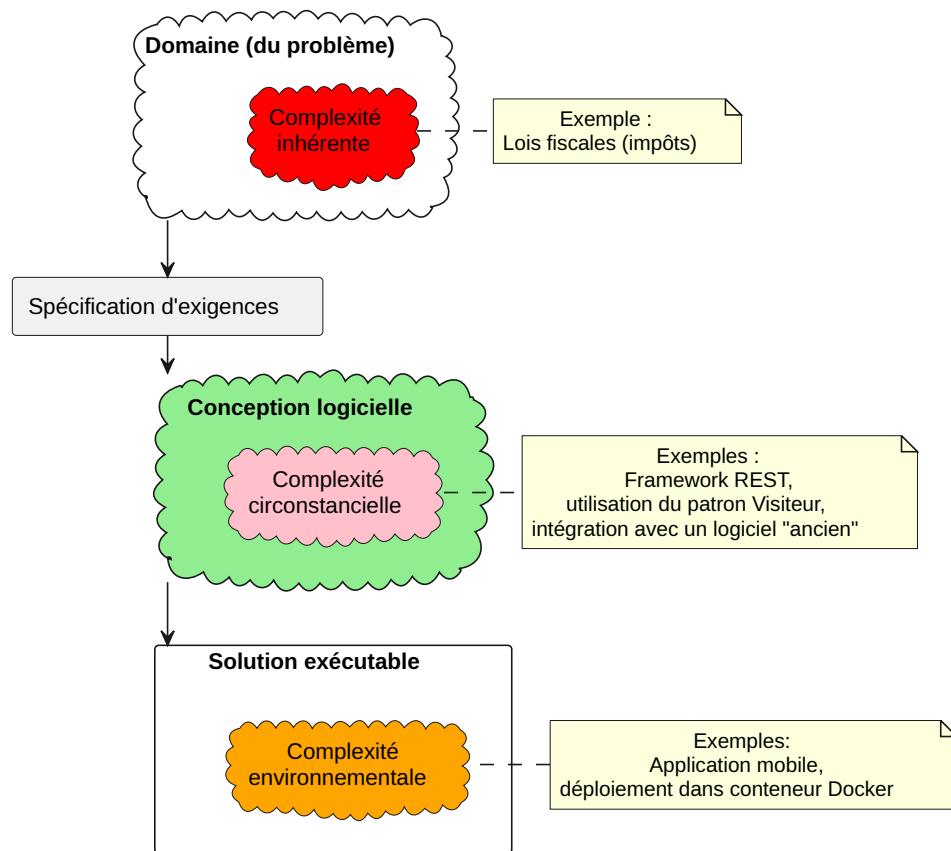


FIGURE 1.4. – Sources de complexité.

- À partir de chaque cas d'utilisation (Chapitre 3), il y a une conception de haut niveau (l'API du système à développer) documentée sous forme de diagramme de séquence système (DSS) (Chapitre 5) ;
- À partir de chaque DSS, on peut définir un ensemble de contrats d'opération (Chapitre 8), surtout pour les opérations complexes ;
- À partir de l'ensemble des besoins, on construit un modèle du domaine (MDD) (Chapitre 4) ;
- Pour faire une conception intuitive et facile à adapter, on propose un modèle de conception sous forme de plusieurs RDCU (Chapitre 9) qui sont cohérentes avec le MDD (pour diminuer le décalage des représentations) et avec les contrats ;
- Pour implémenter les conceptions, on développe du code à partir des diagrammes dans le modèle de conception, ainsi que du code pour tester tout ça selon le développement piloté par les tests (Chapitre 10) ;
- Pour gérer la dette technique (Chapitre 7) on fait du réusinage (Chapitre 11) au besoin ;
- Le tout se fait de manière évolutive, en itérations courtes selon le *Processus unifié*.

1.5. Développement itératif, évolutif et agile

Nous adoptons 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 concepts fondamentaux 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 1.7).
- 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 1.8). 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 1.9). Selon le contexte du projet (voir le **Spectre de la conception**), on peut décider de

1. Analyse et conception de logiciels

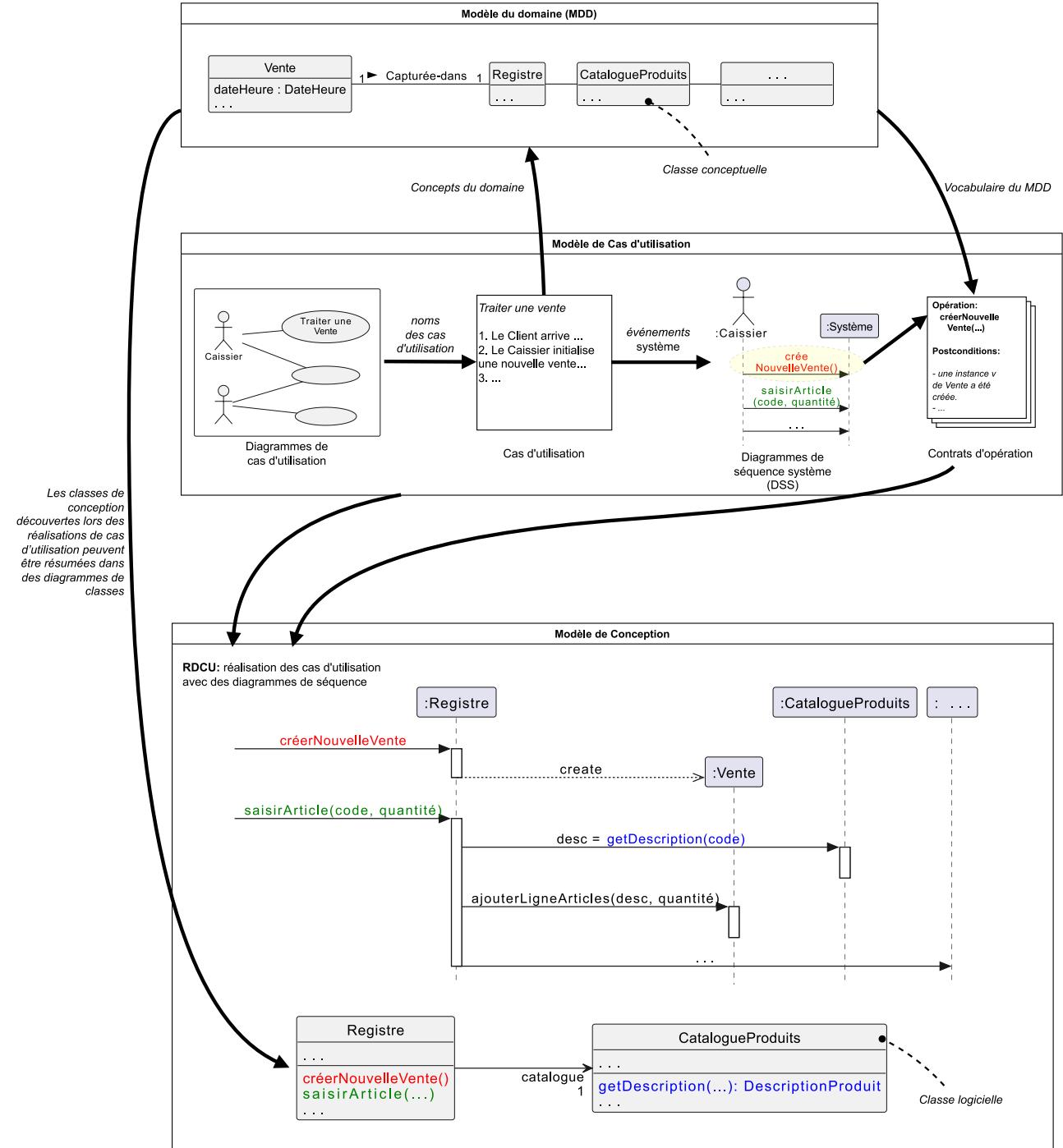


FIGURE 1.5. – Survol de la méthodologie.

ne pas faire de modélisation. Cependant, en fonction de la complexité du projet à réaliser, cela peut amener des risques, ce que l'on appelle la **dette technique**.



FIGURE 1.6. – Un processus itératif permet de gérer les complexités, car la planification d'une itération peut viser une partie du système et le système évolue à mesure que les itérations avancent. (Cette œuvre, « Modèle du processus itératif », est un dérivé de « Iterative development model » de Krupadeluxe, utilisé sous CC BY-SA 4.0. « Modèle de processus itératif » est sous licence CC BY-SA 4.0 par Christopher Fuhrman.)

1. Analyse et conception de logiciels

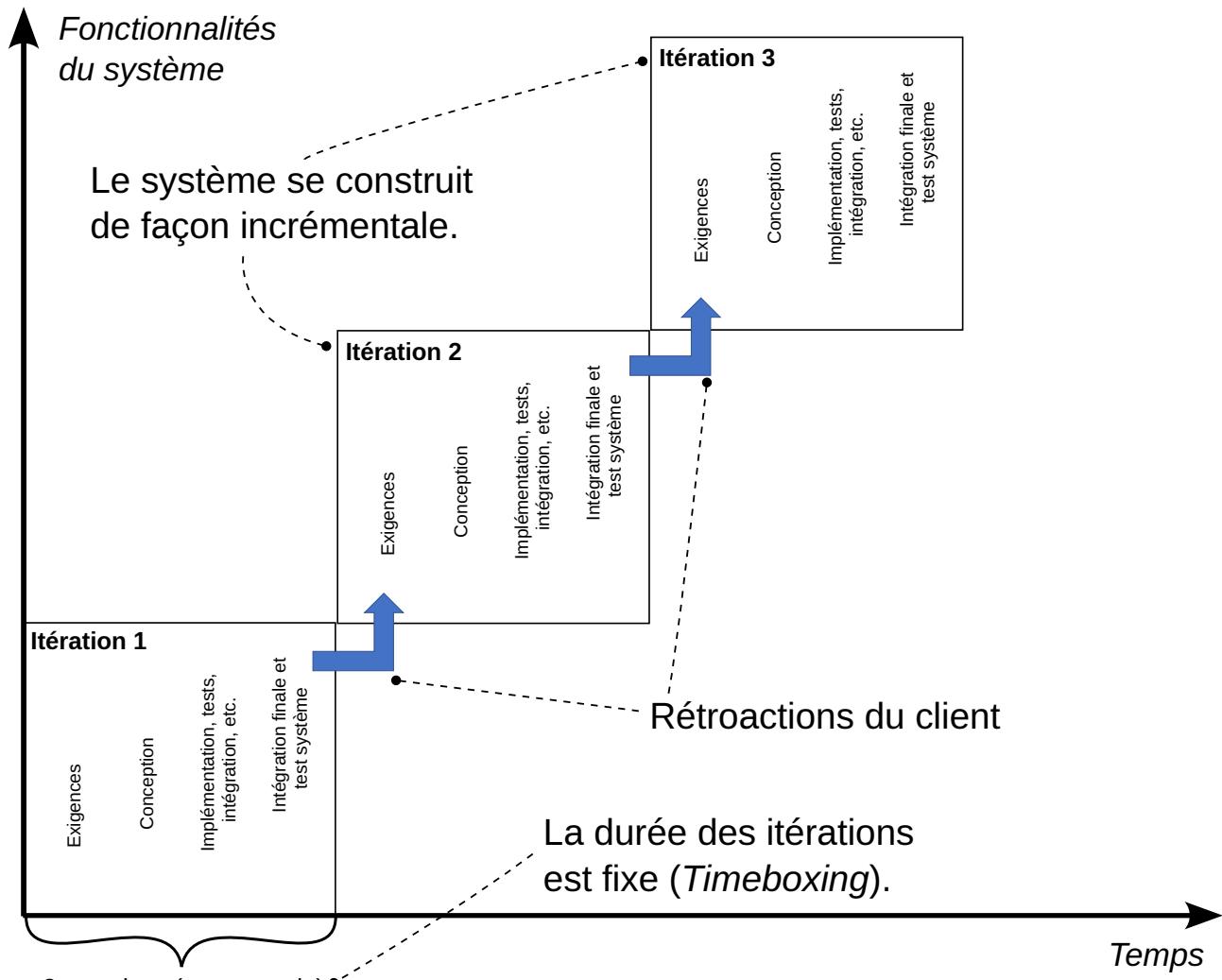


FIGURE 1.7. – Processus itératif et incrémental (évolutif).



FIGURE 1.8. – La rétroaction et la rectification itératives font diminuer l'instabilité des exigences à mesure que le projet avance, car le système à développer converge vers les spécifications et la conception les plus appropriées (adaptation de Larman, 2005, fig. 2.2).

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 précoces ;
- la complexité est gérée (restreinte à une itération) ;
- la possibilité d'exploiter méthodiquement les leçons tirées d'une itération.

Cependant, il y a des défis associés à ce genre de développement :

- **Instabilité apparente au début** : Dans les itérations initiales, puisqu'on n'a pas beaucoup de temps pour comprendre les exigences, le domaine du client et les contraintes du projet, la compréhension des spécifications et la conception sont loin de la « bonne voie ». La conséquence est que les évaluations et les rétroactions peuvent sembler rudes, et cela peut être déstabilisant pour des personnes qui ne sont pas familières avec le processus. La bonne nouvelle est que, normalement, cette instabilité diminue au fur et à mesure que le projet avance (voir la figure 1.8).
- **Modifications des objectifs de l'itération en cours au besoin** : Il arrive souvent que, dans une itération, les choses ne se passent pas comme nous l'avons imaginé. Par exemple, l'ensemble des récits utilisateur ou des scénarios de cas d'utilisation visés pour l'itération nécessite plus de travail que prévu. La tendance dans ce cas est de nous donner plus de temps pour terminer. Mais cela voudrait dire que la rétroaction de toute l'itération sera retardée. Il serait nécessaire de changer la planification de démonstration avec le client (qui a souvent peu de disponibilités). Donc, le processus nous impose de toujours respecter le délai des itérations. Il s'agit d'une **gestion par blocs de temps** (en anglais *timeboxing*). Que faire alors si, dans une itération, nous n'arriverons pas à tout faire ? La résolution est de demander à l'équipe après la moitié de l'itération si les objectifs d'origine peuvent être atteints. Si la réponse est non, nous priorisons les objectifs en plaçant les objectifs secondaires dans la catégorie des « choses à faire » (qui seront éventuellement faites à une itération ultérieure). Voir la figure 1.9. Selon une étude menée par Blincoe et coll. (2019) sur trois gros projets itératifs d'IBM, jusqu'à 54 % des exigences de haut niveau ont été déplacées de cette manière. Le but ultime de cette stratégie est de pouvoir faire une démonstration à la fin de l'itération, même si elle ne comprend pas toutes les fonctionnalités visées au début, car la rétroaction régulière sur des choses qui fonctionnent est essentielle.

1. Analyse et conception de logiciels

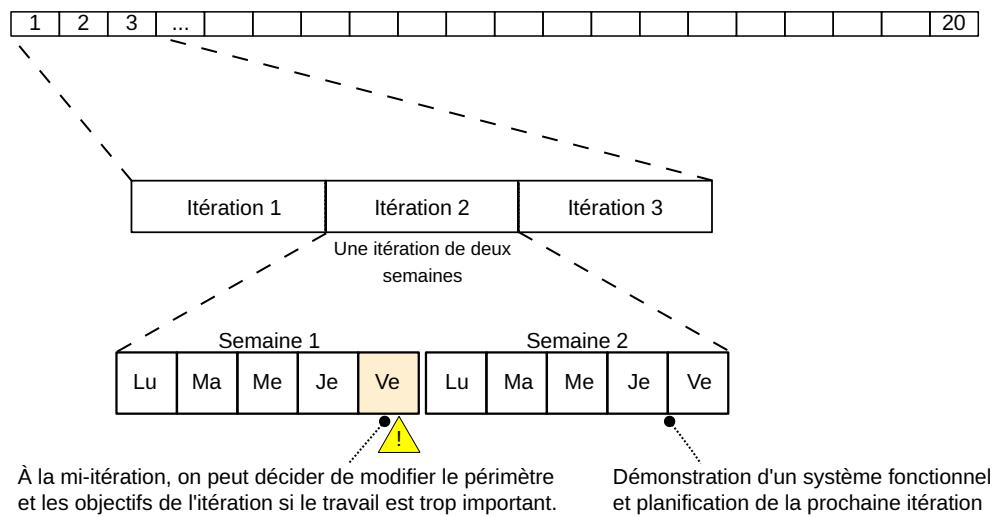


FIGURE 1.9. – Pour respecter la méthode du temps limité, on peut modifier les objectifs d'une itération si le travail est trop important (adaptation de Larman, 2005, fig. 2.4).

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 socioéconomique.

Voici un exemple d'une exigence d'un système de messagerie instantanée (comme Discord ou Slack) :

Il doit permettre à des utilisateurs d'écrire des messages qui seront 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 personnes dans le canal en moins de 0,5 seconde, 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 des fonctionnalités. Il y a beaucoup d'exemples d'exigences non fonctionnelles, par exemple sur [Wikipedia](#) W. Le nom porte à confusion, car une chose qui ne fonctionne pas est « non fonctionnelle ». Pour cette raison, elles 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) section 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 une personne, etc.

2. Besoins (exigences)



FIGURE 2.1. – Ramasser les besoins non fonctionnels ? « Dog Clean Up » de Luis Prado, utilisé selon CCo

- **Fiabilité** (*Reliability*). Comment le logiciel doit se comporter lorsqu'il y a des problèmes ou des pannes. Par exemple un traitement de 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.
- **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 bibliothèques et des langages qui ne sont pas payants (logiciel libre).
 - **Interface**. Par exemple, les contraintes d'interfaçage avec un système externe.
 - **Exploitation**. Par exemple, l'utilisation d'un 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).

i Note

D'autres méthodologies de développement peuvent déterminer les besoins avec les récits utilisateur (*user stories*), qui sont généralement plus courts et moins prescriptifs 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 en savoir plus sur les différences.

La théorie sur *comment écrire* les cas d'utilisation ne fait pas partie de ce manuel (voir [Larman, 2005, Chapitre 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 : Le jeu Risk

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



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

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 :

3.1.1. 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, 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'étant qu'une sorte de *table des matières* des fonctionnalités, il 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.

3. Cas d'utilisation

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.

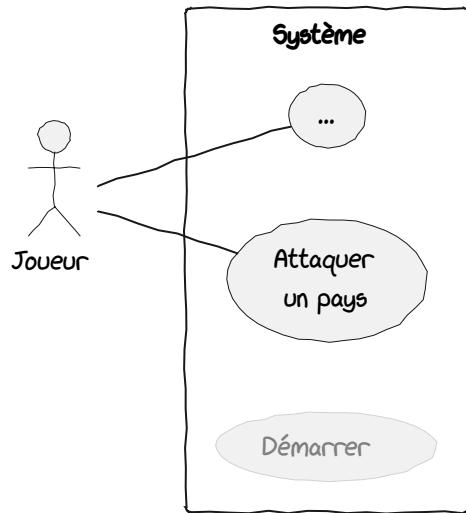


FIGURE 3.2. – Diagramme de cas d'utilisation. (PlantUML)

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'affaires : 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 à 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>

4. Modèle du domaine (MDD, modèle conceptuel)

Catégorie	Exemples
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>
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 (en opposition avec Exemplaire), Cours (en opposition avec 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, SystèmeGestionBordereaux</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 le 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 l'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 on évite 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 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 mots 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 mots 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.

4. Modèle du domaine (MDD, modèle conceptuel)

TABLEAU 4.2. – Extrait du tableau 9.2  (liste d'associations courantes)

Catégorie	Exemples
A est une transaction liée à une transaction B	<i>PaiementEnEspèces – Vente Réservation – Annulation</i>
A est un élément d'une transaction B	<i>LigneArticles – Vente</i>
A est un produit pour une transaction (ou un élément de transaction) B	<i>Article – LigneArticles (ou Vente) 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>

Catégorie	Exemples
A est voisin de B	<i>Article – Article</i> <i>Case – Case</i> <i>Ville – Ville</i>

4.4. Exemple de MDD pour le jeu Risk

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

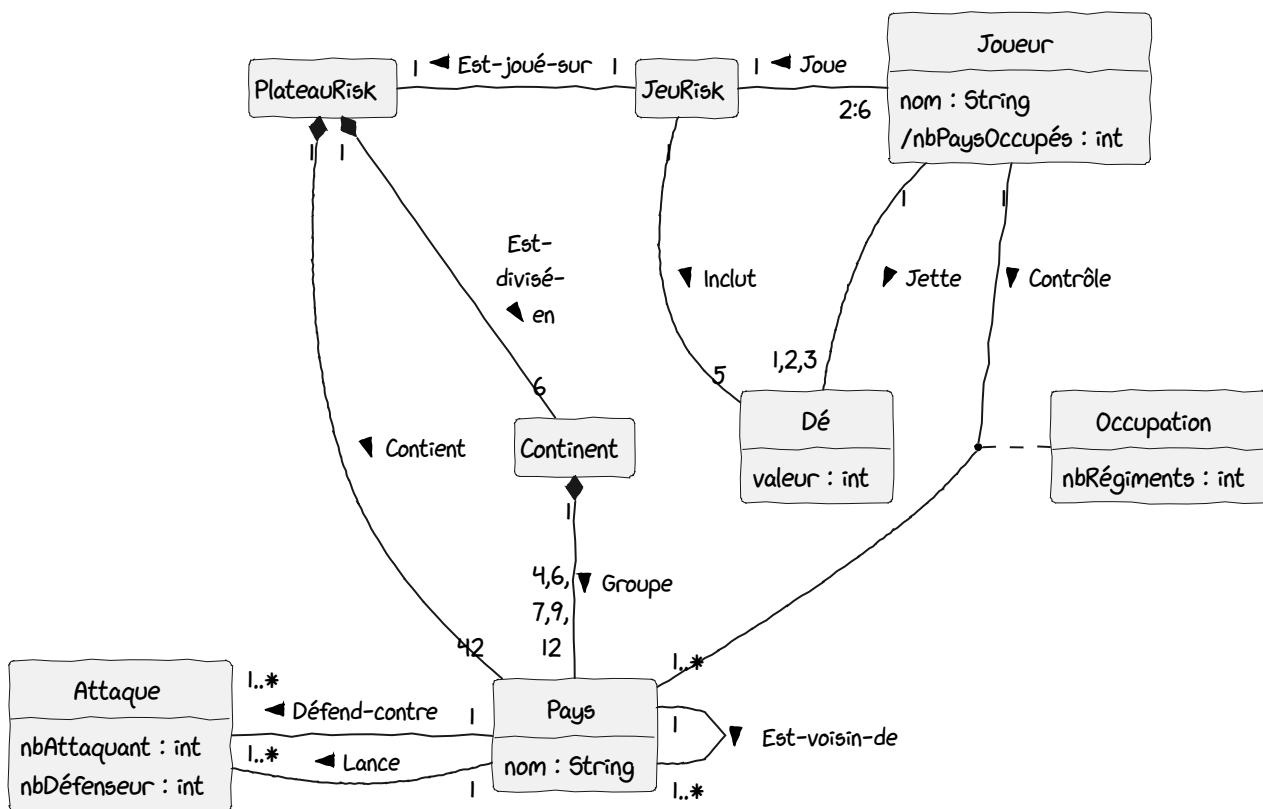


FIGURE 4.1. – Modèle du domaine du jeu Risk. (PlantUML)

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

4. Modèle du domaine (MDD, modèle conceptuel)

leur nom. L'exemple à la figure 4.2 s'applique à la règle du jeu 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.

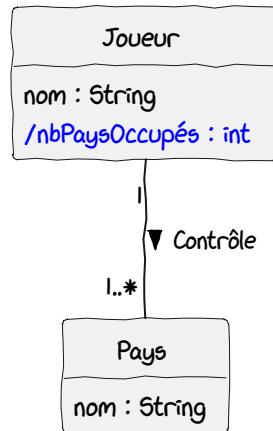


FIGURE 4.2. – nbPaysOccupés est un attribut dérivé et sa valeur sera calculée selon le nombre de pays de l'association. (PlantUML)

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.

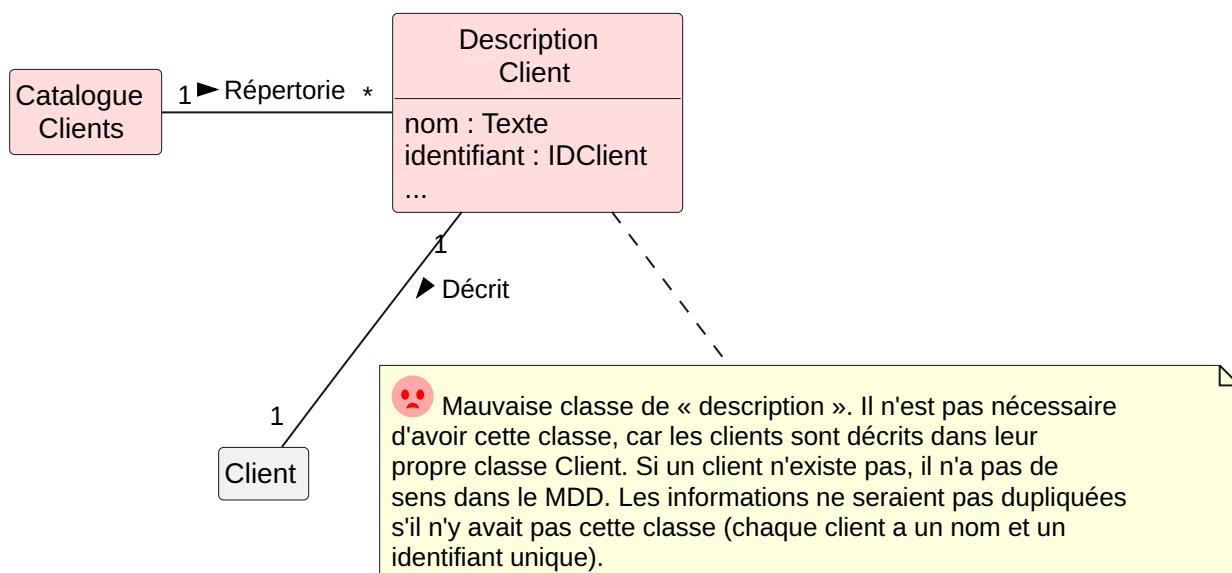


FIGURE 4.4. – Erreur fréquente: utiliser une classe de description sans justification.

4. Modèle du domaine (MDD, modèle conceptuel)

4.7. Classes d'association

Les classes d'association dans un MDD sont le sujet de la section A31.10/F26.10 .

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 ;
- si la durée de vie des instances de la classe d'association dépend de l'association ;
- s'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.

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

Voir le livre obligatoire pour plus d'exemples.

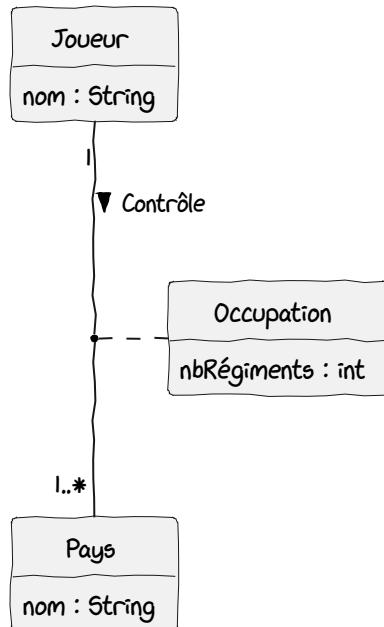


FIGURE 4.5. – Classe d'association dans le MDD Jeu Risk. (PlantUML)

4.8. Affinement du MDD

Lorsqu'on modélise un domaine, il est normal de commencer avec un modèle simple (à partir d'un ou de deux cas d'utilisation) et ensuite de l'affiner 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/A31 . 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 Risk) ;
 - 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/spécialisation, 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.

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

4. Modèle du domaine (MDD, modèle conceptuel)

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 paiement 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 (2005) 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 comportemental dans le domaine.

Vous pouvez aussi lire [cette question](#).

4.10. Exercices

Exercice 4.1 (Classes conceptuelles trouvées par catégorie). À partir du [cas d'utilisation Réserver un livre de la bibliothèque](#), trouvez des classes conceptuelles candidates en utilisant une liste de catégories de classes. Vous pouvez remplir un tableau comme ceci :

Catégorie de classe conceptuelle	Classes candidates selon le cas d'utilisation
Transaction d'affaires (métier) (Continuez avec d'autres catégories) (Certaines catégories ne s'appliqueront pas.)	<i>Réservation</i> (ceci est un exemple) (Certaines classes candidates seront découvertes par plusieurs catégories)

Exercice 4.2 (Classes conceptuelles trouvées à l'aide de groupes nominaux). Cette fois-ci, utilisez les groupes nominaux pour trouver des classes conceptuelles candidates. Commencez par souligner ou par mettre en surbrillance les noms et les groupes nominaux dans le *cas d'utilisation Réserver un livre de la bibliothèque*. Les groupes nominaux peuvent être des classes ou des attributs, ou peuvent ne pas s'appliquer du tout. Faites une liste de classes conceptuelles candidates (sans doublons).

Exercice 4.3 (Comparaison des approches). Comparez la liste des classes trouvées dans l'Exercice 4.2 avec les classes trouvées dans l'Exercice 4.1. Quelles sont les différences ?

Note

Vous pouvez dessiner les diagrammes à la main et en prendre une photo avec une application comme Microsoft Lens ([Android](#), [iOS](#)).

Vous pouvez également utiliser PlantUML. Voici des ressources à ce propos :

- [tutoriel VS Code sur YouTube](#) ;
- [extension PlantUML pour VS Code](#) ;
- [PlantUML Gizmo](#), module supplémentaire Google Docs ;
- [PlantText.com](#).

Méfiez-vous des outils comme Lucidchart ayant seulement des profils superficiels pour UML (voir la figure 13.5).

4. Modèle du domaine (MDD, modèle conceptuel)

Exercice 4.4 (Diagramme de classes conceptuelles). À partir du **cas d'utilisation Réserver un livre de la bibliothèque**, esquissez le modèle du domaine correspondant au domaine de l'application (cela comprend des **classes**, des **attributs** et des **associations**).

1. Considérez les classes candidates provenant de l'Exercice [4.1](#) et de l'Exercice [4.2](#).
2. Notez que les classes candidates dénichées ne sont pas toujours importantes. Certains concepts sont des attributs. D'autres (surtout avec l'approche linguistique) n'ont rien à voir avec le problème du domaine. Vous devez appliquer les directives vues dans le chapitre 9 de Larman ([2005](#)).
3. Faites attention à bien modéliser la classe de « description » dans ce problème.
4. Tout attribut doit avoir un type.
5. Limitez-vous à des associations « mémorisables » dans le contexte de l'application du logiciel (ne pas faire des associations hors de la portée du cas d'utilisation).
6. Vérifiez les cardinalités.
7. Vérifiez les verbes sur les associations ainsi que le sens de lecture.

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 au 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 « Dis 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. Diagrammes de séquence système (DSS)

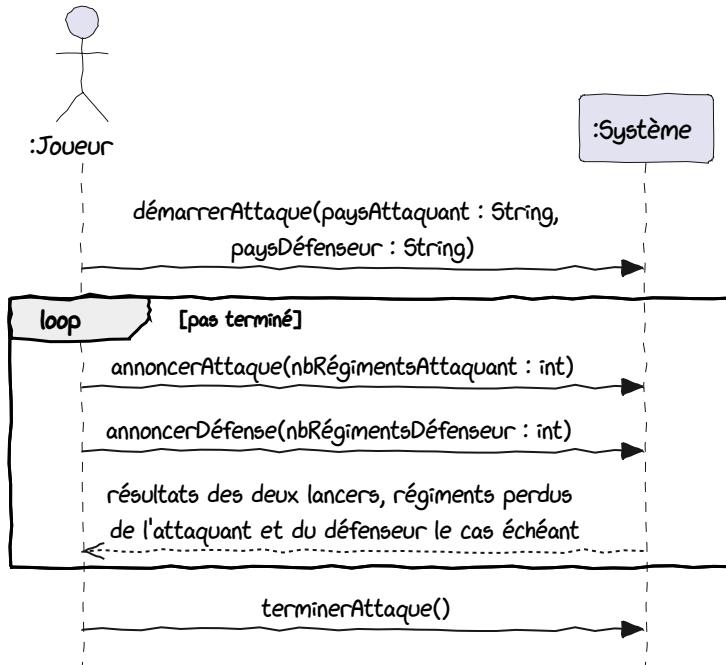


FIGURE 5.1. – Diagramme de séquence système pour *Attaquer un pays*. (PlantUML)

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, il y a l'acteur, le Système et une opération système. On ne rentre pas dans les détails, bien qu'ils existent et soient importants.

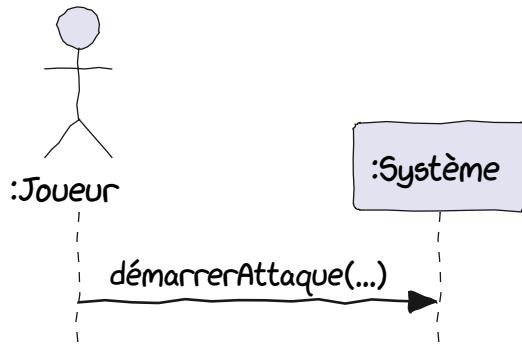


FIGURE 5.2. – Une opération système dans un DSS. C'est une abstraction. (PlantUML)

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,

5.2. Les DSS font abstraction de la couche présentation

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).

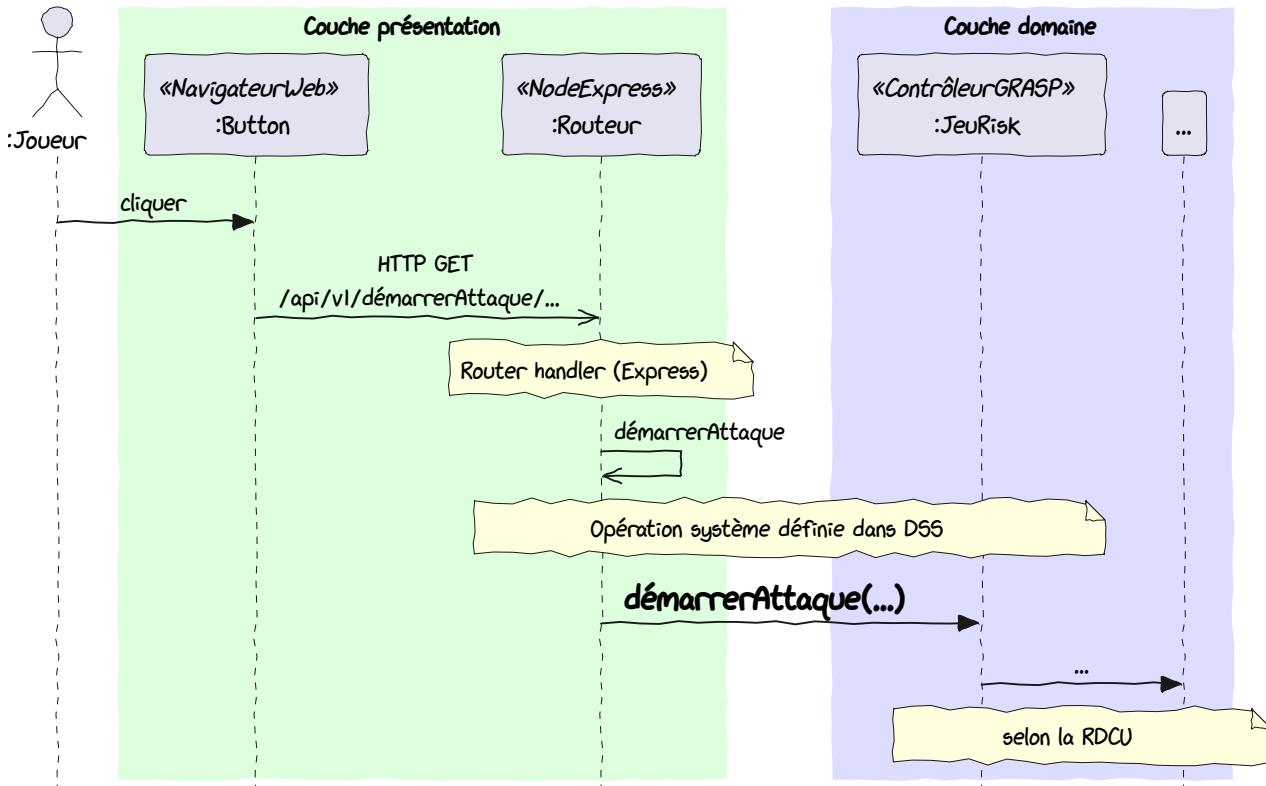


FIGURE 5.3. – Une opération système est envoyée par la couche présentation et elle est reçue dans la couche domaine par son contrôleur GRASP. Ceci est un exemple avec un navigateur Web, mais d'autres possibilités existent pour la couche présentation. ([PlantUML](#))

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

5. Diagrammes de séquence système (DSS)

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 à 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, par exemple pour déterminer qui contrôle quel pays après les attaques.

Cependant, si vous avez une boucle pour indiquer la possibilité de répéter une action (par exemple ajouter des produits dans un système d'inventaire) et que 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 derrière le 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 des `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 n'est-il pas permis de passer des objets comme arguments ?

Il n'est pas conseillé de passer des *objets du domaine* comme arguments, 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 paramètre 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: 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 de le laisser faire le décorticage ?

5. Diagrammes de séquence système (DSS)

Dans un sens, ça serait plus simple (pour le code de la couche présentation). Cependant, on veut 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'applications frontales (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 alors obligé de connaître toutes les trois formes (Web, iOS, Android) et, ainsi, sa cohésion serait 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.

5.4. Exercices

Note

Vous pouvez dessiner les diagrammes à la main et en prendre une photo avec une application comme Microsoft Lens ([Android](#), [iOS](#)).

Vous pouvez également utiliser PlantUML. Voici des ressources à ce propos :

- [tutoriel VS Code sur YouTube](#) ;
- [extension PlantUML pour VS Code](#) ;
- [PlantUML Gizmo](#), module supplémentaire Google Docs ;
- [PlantText.com](#).

Méfiez-vous des outils comme Lucidchart ayant seulement des profils superficiels pour UML (voir la figure 13-5).

Exercice 5.1 (Dessiner un DSS à partir d'un cas d'utilisation). Esquissez le DSS pour le cas d'utilisation **Réserver un livre de la bibliothèque**. Toutes les opérations système doivent définir le type de chaque argument, le cas échéant.

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(e) 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 avec 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 est 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 aurez l'habitude d'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

6. Principes GRASP

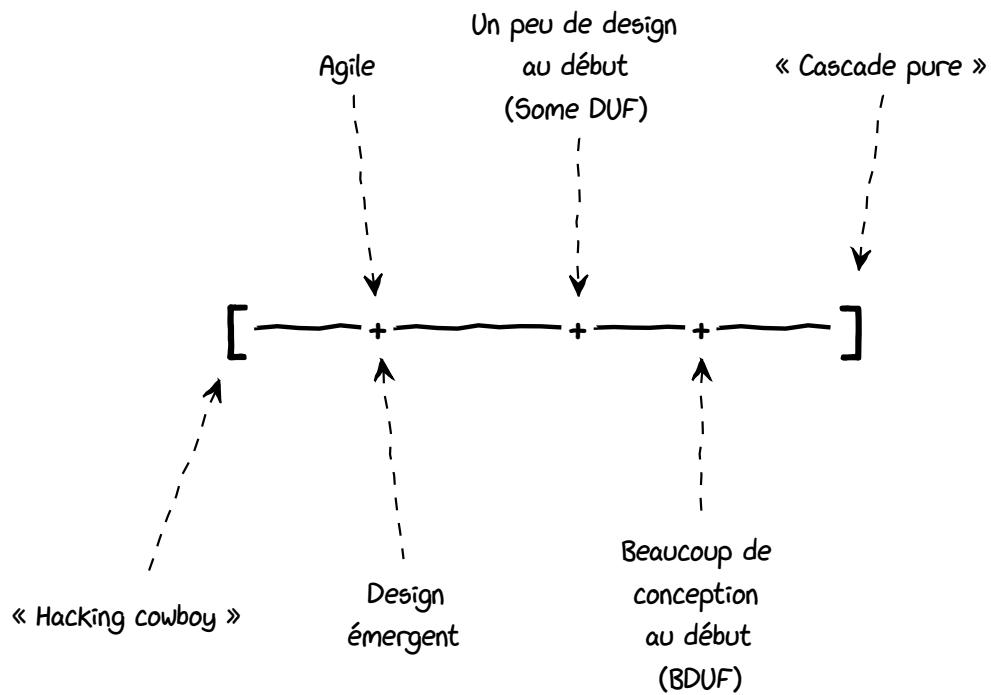


FIGURE 6.1. – Spectre de la conception, adapté de Ford (2009). (PlantUML)

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.

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 seconde période de financement. Si l'entreprise est financée pour une seconde période, la conception du code pourrait avoir besoin de beaucoup de soin, car elle aura 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'avions ou d'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 à valider la conception. Puisque les exigences sont plus stables (et les développeurs(euses) 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.

6.1. Spectre de la conception

Le spectre de la conception est très important pour le monde réel, parce qu'une ingénierie ou un ingénieur devrait pouvoir s'adapter selon 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 l'on vous dit que c'est « agile ».

6. Principes GRASP

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 F16.11/A17.11 	<p>Un principe général de conception d'objets et d'affectation des responsabilités.</p> <p>Affectez une responsabilité à l'expert – la classe qui possède les informations nécessaires pour s'en acquitter.</p>
Créateur F16.10/A17.10 	<p>Qui crée ? (Notez que Fabrique Concète est une solution de rechange courante.)</p> <p>Affectez à la classe B la responsabilité de créer une instance de la classe A si l'une des assertions suivantes est vraie :</p> <ol style="list-style-type: none"> 1. B contient A 2. B agrège A 3. B a les données pour initialiser A 4. B enregistre A 5. B utilise étroitement A
Contrôleur F16.13/A17.13 	<p>Quel est le premier objet en dehors de la couche présentation qui reçoit et coordonne (« contrôle ») les opérations système ?</p> <p>Affectez une responsabilité à la classe qui correspond à l'une de ces définitions :</p> <ol style="list-style-type: none"> 1. Elle représente le système global, un « objet racine », un équipement ou un sous-système (contrôleur de façade). 2. Elle représente un scénario de cas d'utilisation dans lequel l'opération système se produit (<i>contrôleur de session</i> ou contrôleur de cas d'utilisation). On la nomme GestionnaireX, où X est le nom du cas d'utilisation.
Faible Couplage (évaluation) F16.12/A17.12 	<p>Comment minimiser les dépendances ?</p> <p>Affectez les responsabilités de sorte que le couplage (inutile) demeure faible. Employez ce principe pour évaluer les alternatives.</p>

Pattern	Description
Forte Cohésion (évaluation) F16.14/A17.14 	Comment conserver les objets cohésifs, compréhensibles, gérables et, en conséquence, obtenir un Faible Couplage ? Affectez les responsabilités de sorte que les classes demeurent cohésives. Employez ce principe pour évaluer les différentes solutions.
Polymorphisme F22.1/A25.1 	Qui est responsable quand le comportement varie selon le type ? Lorsqu'un comportement varie selon le type (classe), affectez la responsabilité de ce comportement – avec des opérations polymorphes – aux types selon lesquels le comportement varie.
Fabrication Pure F22.2/A25.2 	En cas de situation désespérée, que faire quand vous ne voulez pas transgresser les principes de faible couplage et de forte cohésion ? Affectez un ensemble très cohésif de responsabilités à une classe « comportementale » artificielle qui ne représente pas un concept du domaine – une entité fabriquée pour augmenter la cohésion, diminuer le couplage et faciliter la réutilisation.
Indirection F22.3/A25.3 	Comment affecter les responsabilités pour éviter le couplage direct ? Affectez la responsabilité à un objet qui sert d'intermédiaire avec les autres composants ou services.
Protection des variations F22.4/A25.4 	Comment affecter les responsabilités aux objets, sous-systèmes et systèmes de sorte que les variations ou l'instabilité de ces éléments n'aient pas d'impact négatif sur les autres ? Identifiez les points de variation ou d'instabilité prévisibles et affectez les responsabilités afin de créer une « interface » stable autour d'eux.

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.

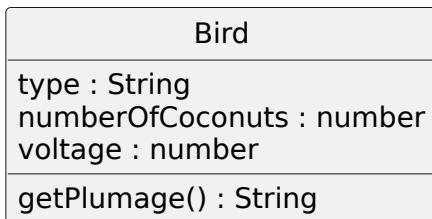
6. Principes GRASP

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.

6.5. Exercices

Exercice 6.1 (GRASP Polymorphisme). Soit le diagramme de classe modélisant l'exemple de Fowler (2018) à *Replace Conditional with Polymorphism*:



Appliquez GRASP Polymorphisme pour le code suivant :

```
get plumage() {
    switch (this.type) {
        case 'EuropeanSwallow':
            return "average";

        case 'AfricanSwallow':
            return (this.numberOfCoconuts > 2) ? "tired" : "average";

        case 'NorwegianBlueParrot':
            return (this.voltage > 100) ? "scorched" : "beautiful";

        default:
            return "unknown";
    }
}
```

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 ajouter une nouvelle fonctionnalité à un système, les développeurs(euses) 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 de 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 que, à un certain moment, ignorer une conception va nuire à la performance des développeurs(euses) en ce qui concerne les nouvelles fonctionnalités.

Voici une courte définition complémentaire de la dette technique (Avgeriou, Kruchten, Ozkaya, & Seaman, 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 (1992) :

7. Dette technique

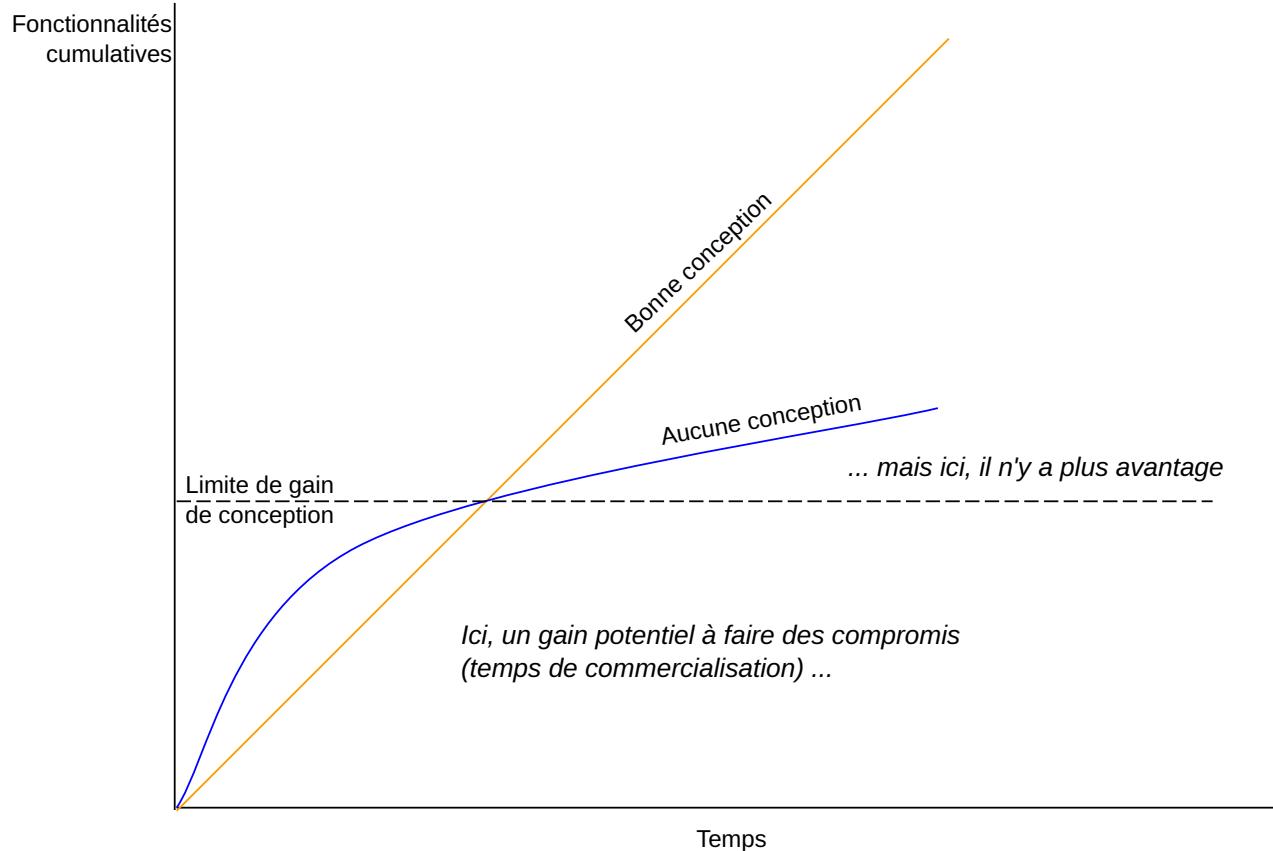


FIGURE 7.1. – « Le pseudographique affiche des fonctionnalités (cumulatives) en fonction du temps pour deux projets stéréotypés imaginaires : l'un avec une bonne conception et l'autre sans conception. Le projet qui ne fait aucune conception ne consacre aucun effort aux activités de conception, qu'il 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)

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 (la) client(e), des quantités excessives de ce genre de code rendront le programme impossible à maîtriser, ce qui entraînera une surspécialisation des programmeurs(euses) et, finalement, un produit inflexible. Livrer du code non raffiné équivaut à s'endetter. Une petite dette accélère le développement tant qu'elle est remboursée rapidement avec une réécriture. [Le paradigme des] objets 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.

Comme c'est une métaphore puissante, beaucoup de développeurs(euses) l'utilisent, et c'est un terme avec une certaine popularité. 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(euses)) 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 par 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

7. Dette technique

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, nous savons comment nous aurions 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 dans 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

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(es) qui ont acheté des choses dans un magasin dans une journée, on imagine 72 instances de Vente, une pour chaque client(e).

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(a) caissier(e) démarre une nouvelle vente pour un(e) client(e). 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 la voix passive au passé composé en français) : « une instance *v* de Vente a été créée ».

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

8.1. Les contrats en bref

Les contrats d'opération sont le sujet du chapitre 11 (2005). Voici les points importants pour la méthodologie enseignée dans le présent manuel :

- Un contrat d'opération correspond à une opération système provenant d'un DSS.
- On fait des contrats surtout pour les opérations système ayant une certaine complexité.
- 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.

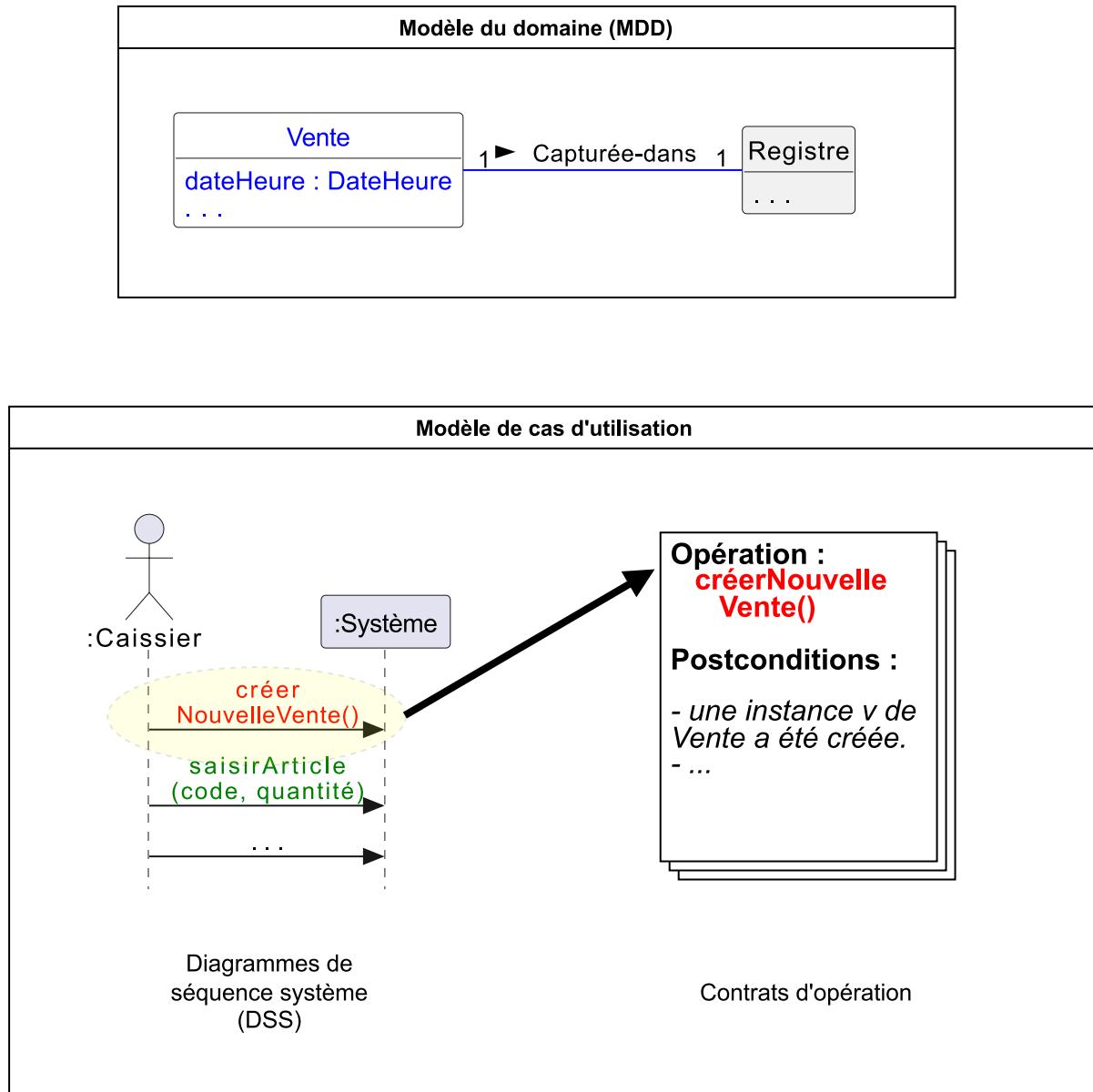


FIGURE 8.1. – Pendant l’opération système **créerNouvelleVente**, une instance de *Vente* doit être créée. Le contrat d’opération le spécifie dans une postcondition.

8. Contrats d'opération

- On ne spécifie pas les préconditions dans les contrats (Larman ne donne pas beaucoup de directives claires à ce propos).
- Il ne faut pas confondre les postconditions d'un contrat d'opération et d'un cas d'utilisation. Ce sont deux choses différentes.
- Quand on rédige 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.

8.2. 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.

Attaquer un pays

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

Postconditions :

- Une nouvelle instance *a* de Attaque a été créée.
- *a* a été associée au Pays correspondant à *paysAttaquant*.
- *a* a été associée au Pays correspondant à *paysDéfenseur*.

Opération : `annoncerAttaque(nbRégimentsAttaquant:int)`

Postcondition :

- *a.nbAttaquant* est devenu *nbRégimentsAttaquant*.

Opération : `annoncerDéfense(nbRégimentsDéfendant:int)`

Postconditions :

- *a.nbDéfendant* est devenu *nbRégimentsDéfendant*.
- L'attribut valeur des *d1* à *d5* est devenu un nombre entier aléatoire entre 1 et 6.
- *Occupation.nbRégiments* est ajusté selon le résultat des valeurs correspondant à *paysAttaquant*.
- *Occupation.nbRégiments* est ajusté selon le résultat des valeurs correspondant à *paysDéfenseur*.

8.2. Exemple : Contrats d'opération pour Attaquer un pays

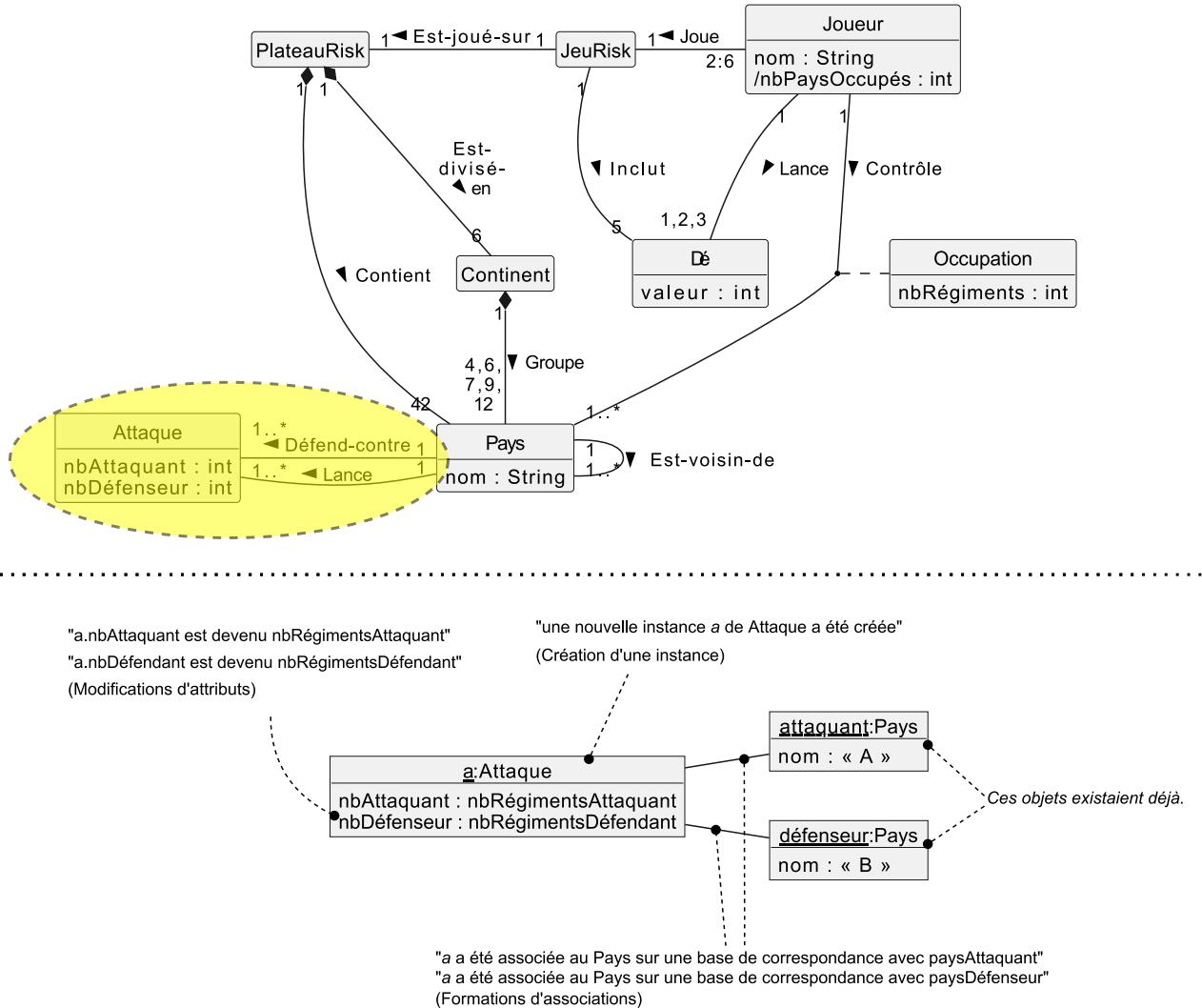


FIGURE 8.2. – Les postconditions décrivent la manipulation d'objets dans un MDD (la partie inférieure ici est un diagramme d'objets).

8. Contrats d'opération

8.3. Feuille de référence

Pour faire des contrats, voici une démarche générale :

1. Faire un contrat pour chaque opération système.
2. Porter une attention à sa signature (les arguments et leur type).
3. Rappeler les formes de postconditions :
 - a) créer/supprimer instances ;
 - b) former/briser associations ;
 - c) modifier attributs.
4. Utiliser *le vocabulaire du modèle du domaine* dans les postconditions. Ça veut dire qu'il faut parler d'instances de classes conceptuelles, de leurs attributs et des associations entre ces classes.
5. Ne pas créer une instance de classe qui existe déjà, par exemple un produit (connu) dans un magasin, un acteur (connu) qui se connecte au système, ou (dans l'exemple de Risk) un Pays (voir la partie inférieure de la figure 8.2).
6. Ne rien oublier. Marquer le MDD ou dessiner un diagramme d'objets, comme à la partie inférieure de la figure 8.2 si nécessaire.

8.4. Exercices

Exercice 8.1 (`terminerAttaque`). Rédigez le contrat d'opération pour `terminerAttaque()`. Il faut considérer le cas où une attaque a réussi, c'est-à-dire que le `paysDéfenseur` change de Joueur (celui du `paysAttaquant`). Suivez les exemples de contrats d'opération à la Section 8.2.

Exercice 8.2 (Contrats d'opération pour Traiter une vente). Rédigez **un contrat d'opération pour chacune des opérations système** dans le DSS (et qui est cohérent avec le MDD) de la Section B.1. Suivez les exemples de contrats d'opération à la Section 8.2.

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 vous sera difficile de réussir les RDCU. Il est normal de ne pas tout comprendre au début, alors posez des 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 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 sur 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 de 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*

9. Réalisations de cas d'utilisation (RDCU)

seule manière de valider une RDCU. Évidemment, la programmation prend beaucoup plus de temps et n'est pas insignifiante. 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 connaître lors de la construction. Alors, votre RDCU sera incertaine 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](#).

Mise en garde

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.



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)

- 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](#) .

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.

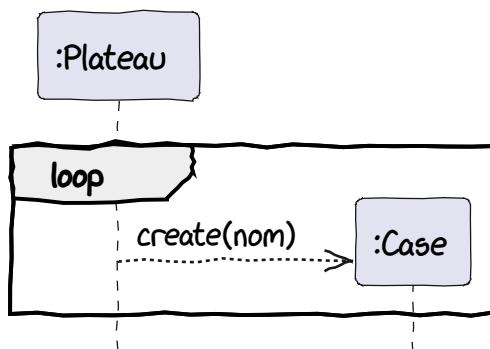


FIGURE 9.2. – Combiner la création d'une instance et une modification de son attribut dans un constructeur. ([PlantUML](#))

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 d'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 Stack Overflow](#).

- 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 précédent, 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).



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

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

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). Un autre exemple du livre de Larman (2005) est l'identifiant `codeArticle` transformé en objet `DescriptionProduit` par la méthode `CatalogueProduits.getDescProduit(codeArticle:String):DescriptionProduit`.

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

9. Réalisations de cas d'utilisation (RDCU)



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

9.5. Utilisation d'un 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.

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 d'un 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) 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 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** ses 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 que 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 des bases 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

9. Réalisations de cas d'utilisation (RDCU)

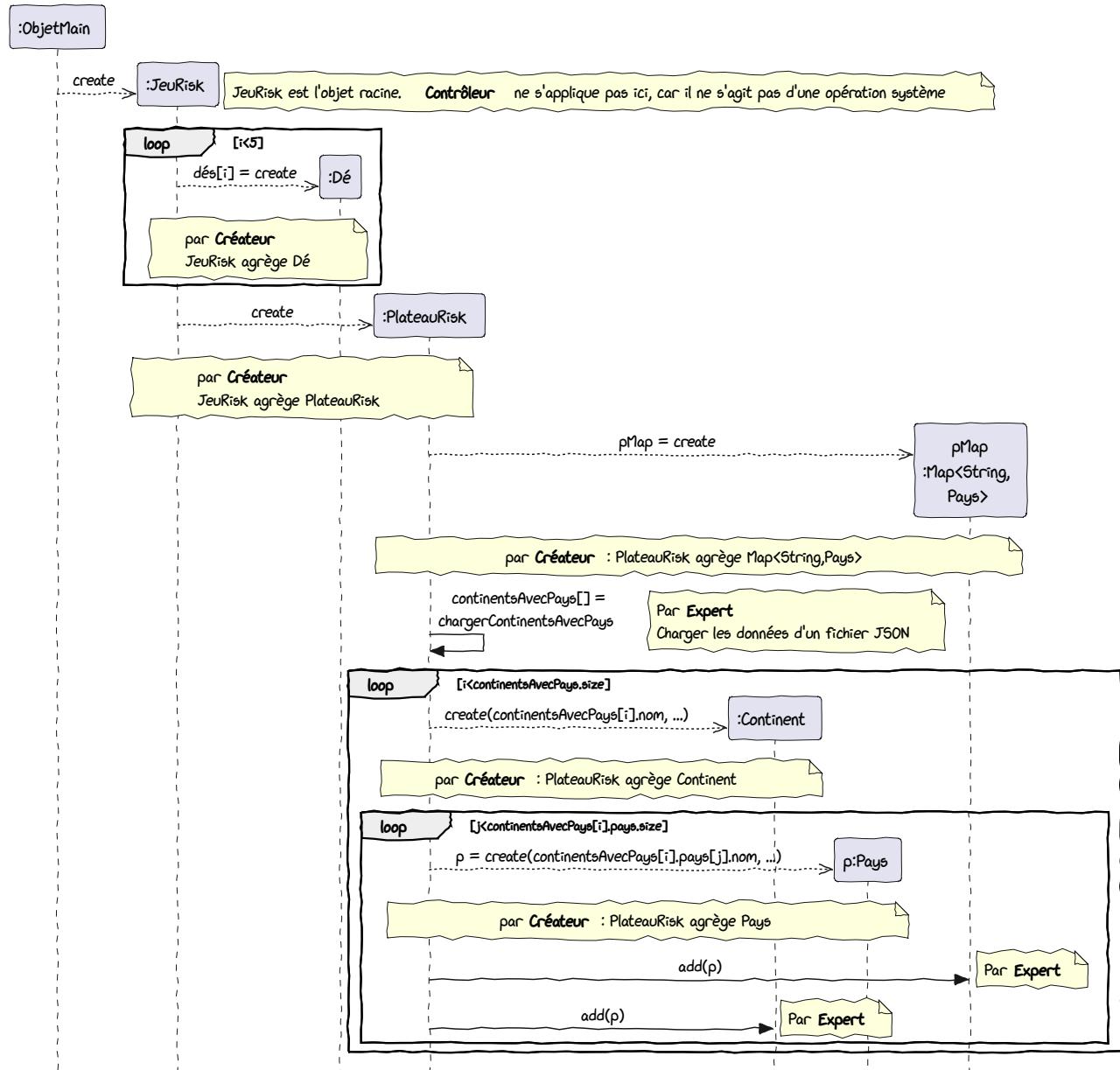


FIGURE 9.6. – Exemple de l'initialisation partielle du jeu Risk. (PlantUML)

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.

🔥 Mise en garde

Une application de patterns GoF à la solution peut nuire à ce principe, car ces patterns ajoutent 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 9.7a) :



(a) Dés dans la vraie vie (« Hand of chance » (CC BY 2.0) par Alexandra E Rust).



(b) Dé dans un logiciel selon *Faire soi-même*.

FIGURE 9.7. – **Faire soi-même** : « Moi, objet logiciel, je fais moi-même ce qu'on fait normalement à l'objet réel dont je suis une abstraction » de Coad (1997).

9. Réalisations de cas d'utilisation (RDCU)

9.9. Exercices

Exercice 9.1 (RDCU pour le cas d'utilisation *Ouvrir la caisse*). Faites les RDCU pour **le cas d'utilisation *Ouvrir la caisse***. Vous y trouverez également des artefacts tels que le DSS, les contrats d'opération et le modèle du domaine. Ils sont essentiels pour faire les RDCU selon la méthodologie.

10. Développement piloté par les tests

Si l'on écrivait des logiciels pouvant se tester automatiquement ? Le développement piloté par les tests (en 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 développeur(euses). Historiquement, si l'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(euses).

Ainsi, il y a toujours des tests pour les fonctionnalités, et le développement se fait en petits incrémentations 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. Des puristes du TDD insisteront sur le fait que le test soit écrit toujours en premier ! Cette discipline est parfois culturelle.

Plusieurs chercheurs(euses) ont mené des expériences, par exemple Karac & Turhan (2018), pour voir si *tester en premier* avait un vrai bénéfice. Les résultats de leurs analyses n'ont pas toujours montré que c'est le cas (ce genre d'expérience est difficile à faire, en partie parce qu'il n'y a pas beaucoup de développeurs(euses) en industrie qui le pratiquent). Les chercheurs(euses) 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(euses) 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 (en 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**.

10. Développement piloté par les tests

L'exécution de tests peut même être 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).

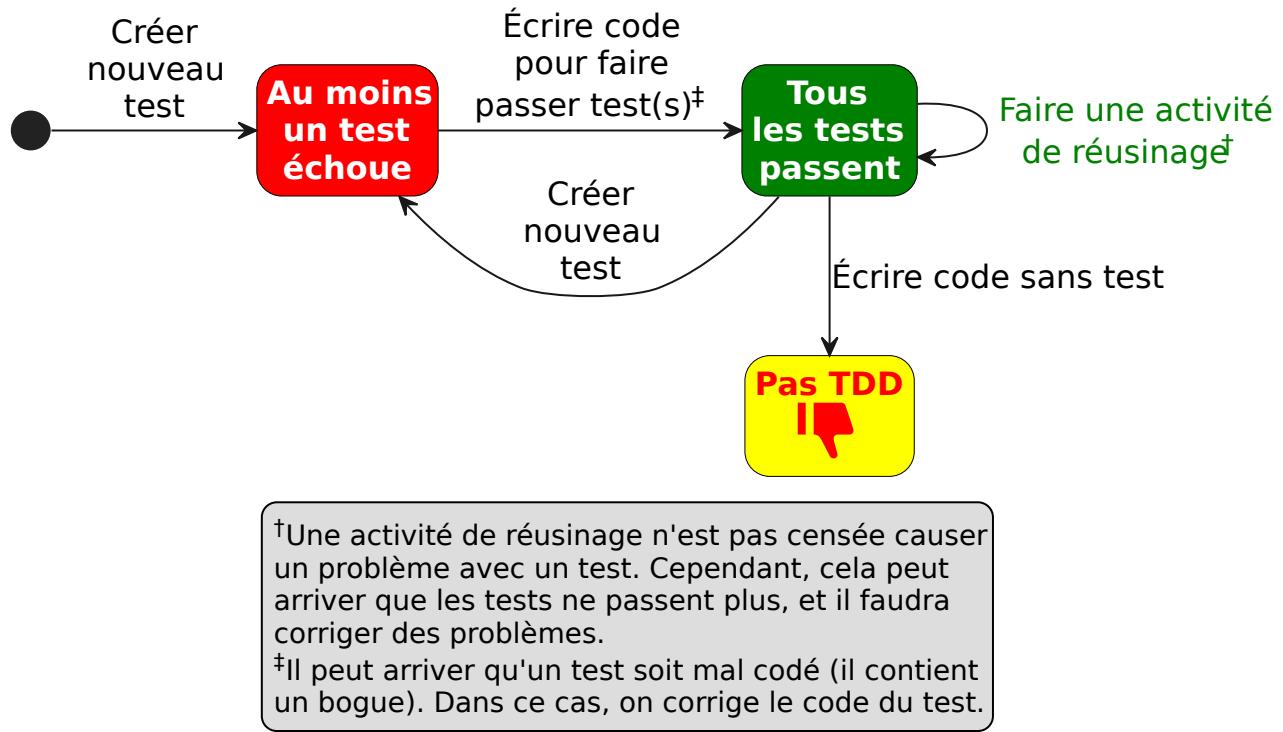


FIGURE 10.1. – États du développement piloté par les tests. ([PlantUML](#))

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

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.

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 un environnement de test différents. Le kata vous permet d'avoir une



FIGURE 10.2. – Étudiante de karaté faisant le kata *Bassai Dai* (photo « Karate » (CC BY-SA 2.0) par The Consortium).

10. Développement piloté par les tests

facilité avec les aspects techniques de développement dans plusieurs dimensions (complétion de code pour le 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 TDD 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 Visual Studio 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). Chacune 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(euse), codeur(euse)) 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 décrites 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 2^e édition du livre *Refactoring* de Fowler (2018) :

Il était une fois un consultant qui a rendu visite à l'équipe d'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 avaient 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(euses) travaillant sur la hiérarchie. Les programmeurs(euses) 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(euses) ont donc passé un jour ou deux à nettoyer la hiérarchie. Une fois qu'ils eurent terminé, 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(euses) 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 des universitaires. Le consultant a suggéré qu'un nettoyage similaire soit effectué sur d'autres parties centrales du système, ce qui pourrait interrompre le projet pendant une

semaine ou deux. Tout cela était pour rendre le code plus beau, pas pour lui faire faire ce qu'il ne faisait pas déjà.

Cette histoire est un bon exemple des deux forces constamment en jeu lors d'un développement de logiciel. D'un côté, on veut que le code fonctionne (pour satisfaire les fonctionnalités) et, d'un autre côté, on veut que la conception soit acceptable puisqu'il y a d'autres exigences sur un logiciel telles que la maintenabilité, l'extensibilité, etc. La section sur le [Spectre de la conception](#) aborde cette dynamique.

Le réusinage (en 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 intégrante 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 l'on est en train d'écrire du code. Plutôt que d'être bloqué sur le choix, on met un nom arbitraire (ou peut-être que, 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 (celles) qui aiment le bon design. Voici la liste complète des « smells » ainsi que des activités de réusinage à appliquer (voir le catalogue sur le site Web <https://refactoring.com/catalog/> pour les détails).

11. Réusinage (Refactorisation)

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

Symptôme de mauvaise conception (« Smell »)	Activités de réusinage à appliquer éventuellement
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 du 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.

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 *restructuration* dans ce cas.

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

12. Développement de logiciels 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 (ancien employé de Google) et Ben Collins-Sussman (développeur fondateur du système de contrôle de version Subversion, employé de 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 encore du code pour Flash W ? Cette technologie est maintenant désuète, et ça ne vaut pas beaucoup 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(euses) qui ont également des compétences générales (*soft skills*).

Fitzpatrick & Collins-Sussman (2012) abordent des problèmes dus aux tendances comportementales chez les développeurs(euses). Par exemple, une personne n'a pas toujours envie de montrer son code source aux autres membres de l'équipe 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 d'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.

Fitzpatrick & Collins-Sussman (2012) le disent, et c'est un fait : si nous sommes, dans l'ensemble, plus ou moins compétent(e)s sur le plan technique, ce qui fera la différence importante dans une carrière est notre habileté à collaborer avec les autres.

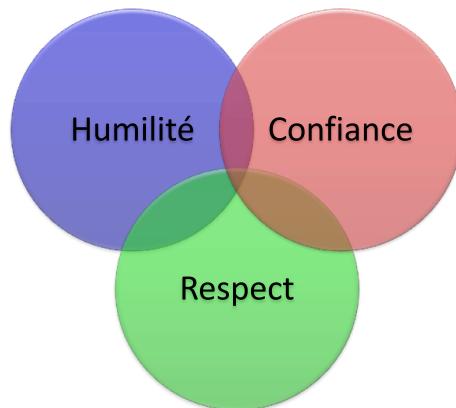


FIGURE 12.1. – Pratiquement tout conflit social est dû à un manque d’humilité, de respect ou de confiance.

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 (Fitzpatrick & Collins-Sussman, 2012) :

- Je ne suis pas le centre de l’univers.
- Je ne suis ni omnisciente ni infaillible.
- Je suis ouverte à m’améliorer.

! Important

L’humilité ne veut pas dire « je n’ai pas de valeur » ou « j’accepte les attaques de la part des autres ». Voir la section **Proposer des solutions au besoin**.

Quelques exemples concrets d’humilité dans le développement :

- Les membres de l’équipe qui débutent en JavaScript, Git, etc. vont le reconnaître et vont même faire des exercices sur Internet pour s’améliorer.
- Les membres de l’équipe qui ont pris une mauvaise décision (technique ou autre) vont l’avouer. Elles et ils savent que les autres ne sont pas là pour les attaquer (il y a du respect).

12. Développement de logiciels en équipe



FIGURE 12.2. – Éviter d'être le « Centre de l'univers » (CC BY-NC-ND 2.0) par Diamondduste.



FIGURE 12.3. – « Missing » (CC BY-SA 2.0) par smkybear.

Équipe > Moi

FIGURE 12.4. – Un(e) membre de l'équipe humble va accepter une décision prise par l'équipe, même s'il ou elle n'est pas en accord à 100%. (PlantUML)

12.2. Redondance des compétences dans l'équipe (*bus factor*)

- Un(e) membre de l'équipe va travailler fort pour que *son équipe* réussisse.
- Un(e) membre de l'équipe qui reçoit une critique ne va pas la prendre personnellement. Il ou elle 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 (Fitzpatrick & Collins-Sussman, 2012) :

- 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 de la confiance pense ainsi (Fitzpatrick & Collins-Sussman, 2012) :

- Je crois que les autres membres de l'équipe font preuve de compétence et de bon jugement.
- Je suis à l'aise lorsque les autres (membres de l'équipe) prennent le volant, le cas échéant.

Le dernier point peut être extrêmement difficile si, par le passé, une personne (incompétente) à qui vous aviez délégué une tâche n'a pas répondu à vos attentes.

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(e) membre de l'équipe (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 personnes à perdre (par exemple, elles ont été heurtées 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 là, le projet s'arrête !

12. Développement de logiciels en équipe



Dans le cas d'une équipe, un(e) membre de l'équipe peut s'absenter ou être moins disponible pour beaucoup de raisons. Par exemple, cette personne part en vacances, tombe malade, prend un congé parental, change d'emploi, abandonne le cours (contexte de projet universitaire), etc. 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. L'automatisation des tests dans un processus de construction de logiciels (*build process*) à la devops (Chapitre 10) aide aussi, car l'équipe ne dépend pas d'une personne pour construire la solution, rouler les tests, etc. Ces pratiques vont également faciliter l'intégration de nouvelles personnes dans l'équipe.

⚠ Avertissement

Si un(e) membre de l'équipe quitte en cours de trimestre, il n'est pas facile de maintenir le même rythme. Cependant, les enseignant(e)s et les auxiliaires de laboratoire s'attendront à ce que vous ayez pensé à un « plan B » avant cette perte. Au moins une autre personne dans l'équipe doit être au courant de ce que faisait l'ancien(ne) membre de l'équipe, pour que le projet ne soit pas complètement arrêté.

12.3. Mentorat

Pour des raisons pédagogiques (Oakley, Felder, Brent, & Elhajj, 2004a), c'est l'enseignant(e) qui décide la composition des équipes. Ça veut dire que, forcément certaines personnes de l'équipe ont plus d'expérience et de facilité à faire certaines tâches que d'autres. Les équipes doivent composer avec cette diversité.

Selon Fitzpatrick & Collins-Sussman (2012) :

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

Si les personnes plus fortes n'aident pas les autres, ils risquent de les éloigner de l'équipe et de se retrouver seules sur le plan des contributions techniques. Voir la section sur la [Redondance des compétences dans l'équipe \(*bus factor*\)](#).

Encadrer un(e) membre de l'équipe 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 facteur de bus.

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 donner à la personne encadrée.

Le dernier point est important parce que, si vous donnez trop d'informations, la personne peut vous ignorer plutôt que de vous dire gentiment qu'elle a compris (Fitzpatrick & Collins-Sussman, 2012). En plus, donner un faible niveau d'orientation à une personne ayant déjà de l'expérience est plus efficace que de donner une orientation explicite (Chen, Kalyuga, & Sweller, 2017 ; Oakley & Sejnowski, 2021). Un bon(ne) mentor(e) doit pouvoir estimer le niveau de la personne et lui donner une orientation appropriée, ce qui n'est pas toujours facile. Mais sachez que « moins est plus » dans certains cas.

12.4. Scénarios

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

- Une personne à l'équipe se trouve à être la seule à faire de la programmation.
- Elle ne fait plus confiance aux autres membres de l'équipe, car leur code est trop bogué.
- Elle n'a pas la patience pour accommoder les membres de l'équipe avec moins d'expérience.

12. Développement de logiciels en équipe

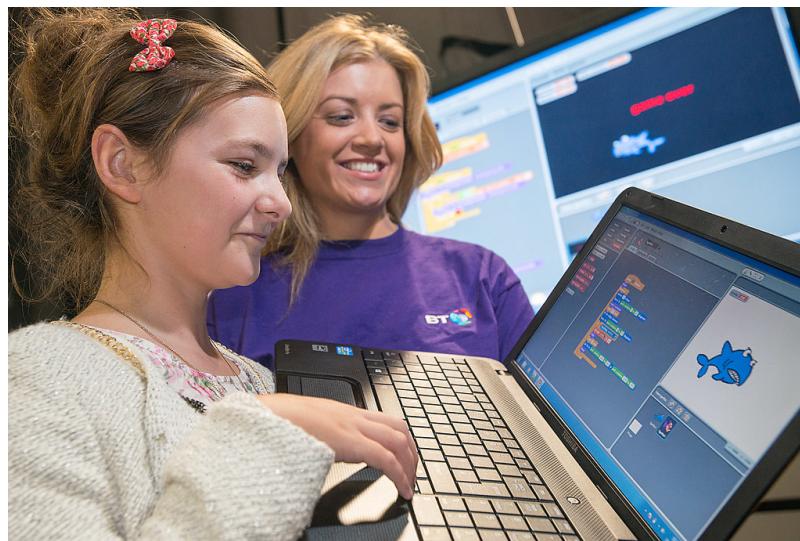


FIGURE 12.5. – Savoir encadrer les membres de l'équipe est une habileté à mettre sur son CV. « Cultu-reTECH BT Monster Dojo » (CC BY 2.0) par connor2nz.

- Elle croit que les autres auraient dû apprendre à mieux programmer dans les cours préalables.
- Une personne dans l'équipe dit qu'elle a « fait ses trois heures de contribution » chaque dimanche chez elle et que ça devrait suffire pour sa partie (elle a un emploi et n'a pas beaucoup de temps pour l'équipe du projet universitaire).
- Un ou deux membres d'une équipe abandonnent le cours après les évaluations de mi-trimestre, par crainte d'échec.
- Un membre de l'équipe suit cinq (!) cours en même temps et n'a pas le temps suffisant pour travailler correctement dans les laboratoires de cette matière.
- Plusieurs membres de l'équipe ont de l'expérience, mais ont de la difficulté à s'entendre sur l'orientation du projet.
- L'équipe n'est pas cohésive : chaque membre 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 une personne ressource, comme votre superviseur(e) (en stage), les auxiliaires de laboratoire ou l'enseignant(e).

Des conseils pour mieux évaluer le travail de chacun(e) dans l'équipe au laboratoire sont présentés dans la section **Évaluer les contributions des membres de l'équipe**.

13. Outils pour la modélisation UML

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

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

Mise en garde

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 à dessiner les modèles en mode esquisse (à la main).

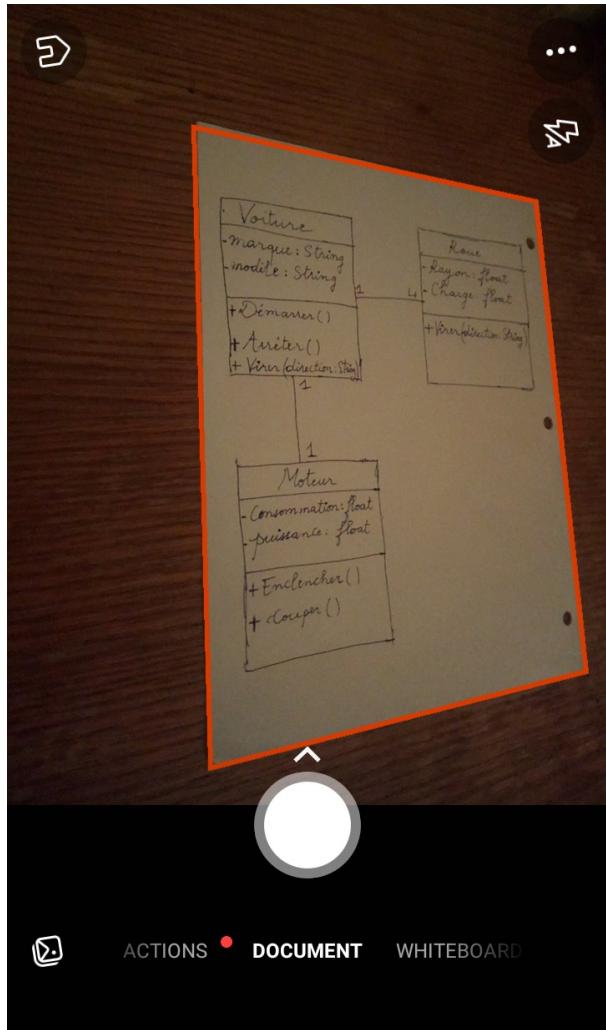
Dans la méthodologie de ce manuel, on exploite l'outil PlantUML pour faire beaucoup de modèles. C'est un outil qui a plusieurs avantages :

- Il est basé sur un langage spécifique à un domaine (en anglais *domain-specific language* ou *DSL*), dont les fichiers peuvent être facilement mis sur un contrôle de version (Git) ;
- Il est basé sur du code libre ;
- Il s'occupe de la mise en page des diagrammes (cela est parfois un inconvénient si un modèle est complexe) ;
- Il est populaire (utilisé par des ingénieur(e)s chez Google pour documenter Android, Pay, etc.) ;
- Il existe plusieurs supports pour les outils de documentation :
 - extension [PlantUML pour Visual Studio 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).

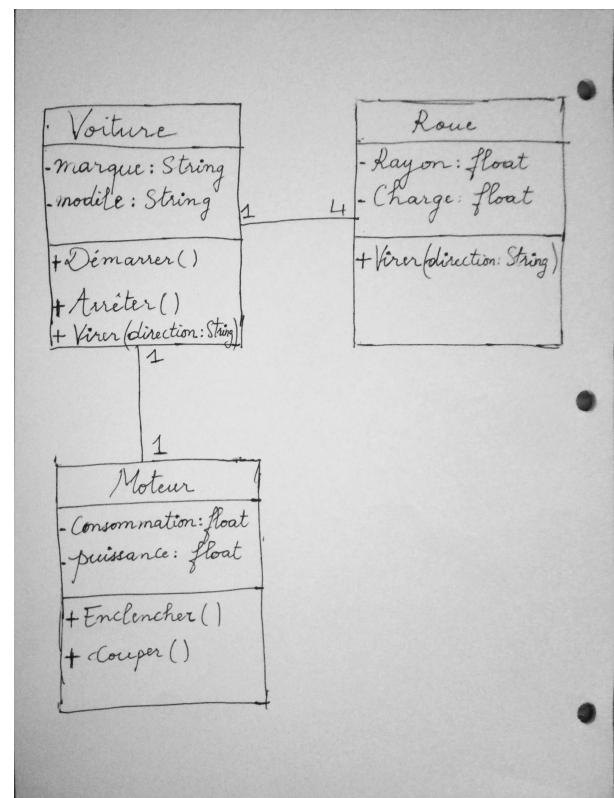
Pour une personne débutante, le langage PlantUML peut sembler plus compliqué que d'utiliser un outil graphique comme Lucidchart. Cependant, pour beaucoup de diagrammes (comme les diagrammes de séquence), ça peut être plus long à créer ou à modifier. Bien que ces outils aient des gabarits ou des modes « UML », ceux-ci ne sont pas toujours conviviaux ou complets. Ce sont 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

13. Outils pour la modélisation UML



(a) Reconnaissance du cadre d'image sur une feuille



(b) Transformation vers un résultat plus lisible

FIGURE 13.1. – Microsoft Lens peut détecter le cadre d'un dessin sur un tableau blanc ou sur papier et le transformer, même si l'on n'est pas droit devant le dessin.

13.1. Exemples de diagrammes avec PlantUML

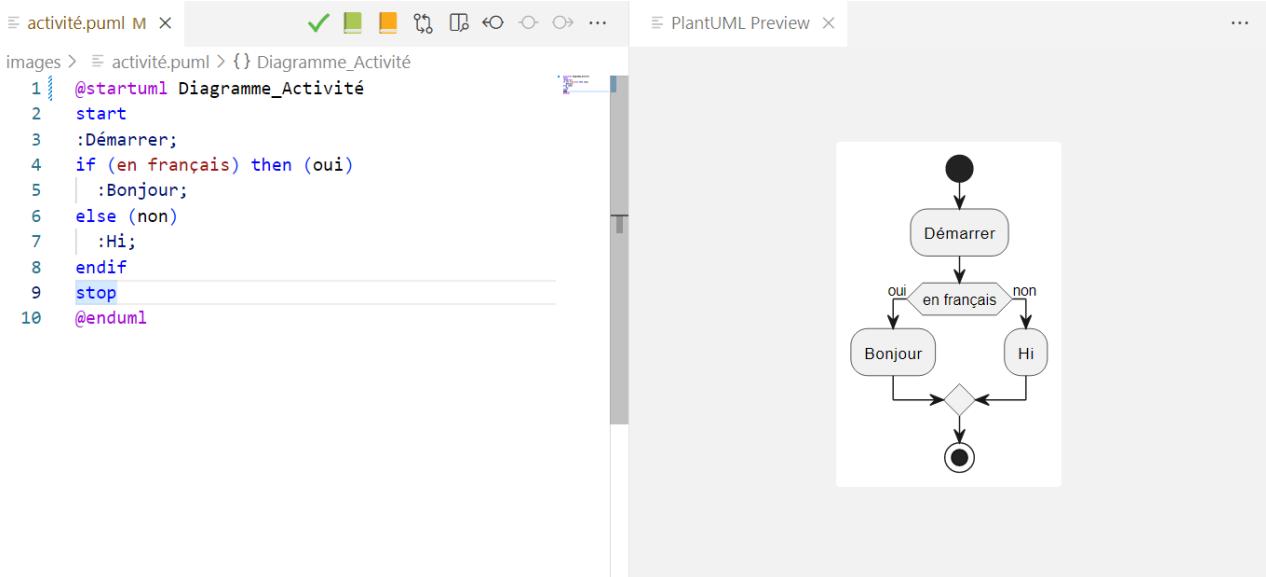


FIGURE 13.2. – L’extension PlantUML pour Visual Studio Code.

sens en UML (voir la figure 13.5). Il est possible de corriger le diagramme, mais en combien de clics ? C’est très vite agaçant.

13.1. Exemples de diagrammes 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 du 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’une 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’une URL avec l’outil PlantUML localement avec l’option `-decodeurl {clé}` de la ligne de commande :

13. Outils pour la modélisation UML

```
$ java -jar plantuml.jar -decodeurl Syp9J4vLqBLJSCffib9mB2t9ICqhoKnEBCdCprC8IYqiJIqkuGBAAUW2rJ
@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 métadonnées 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
```

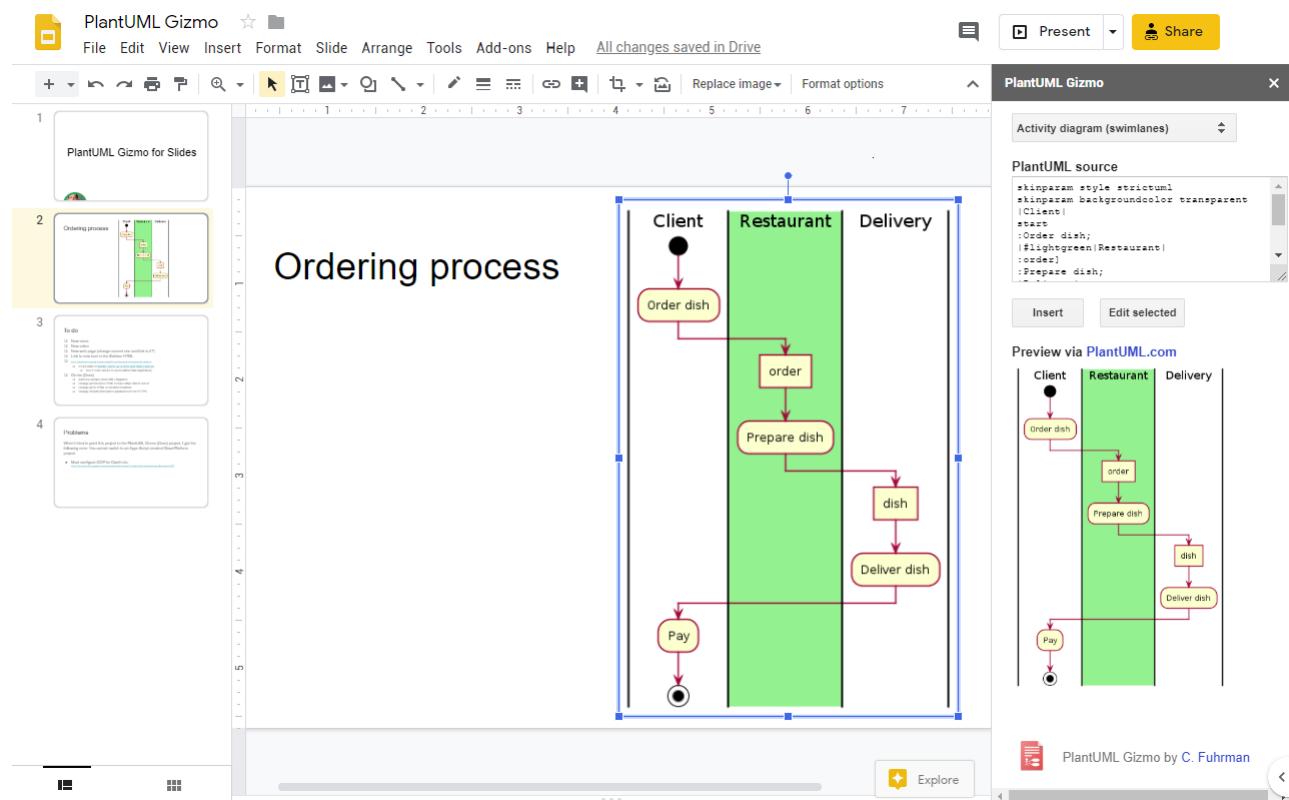


FIGURE 13.3. – PlantUML Gizmo pour Google Docs et Google Slides.

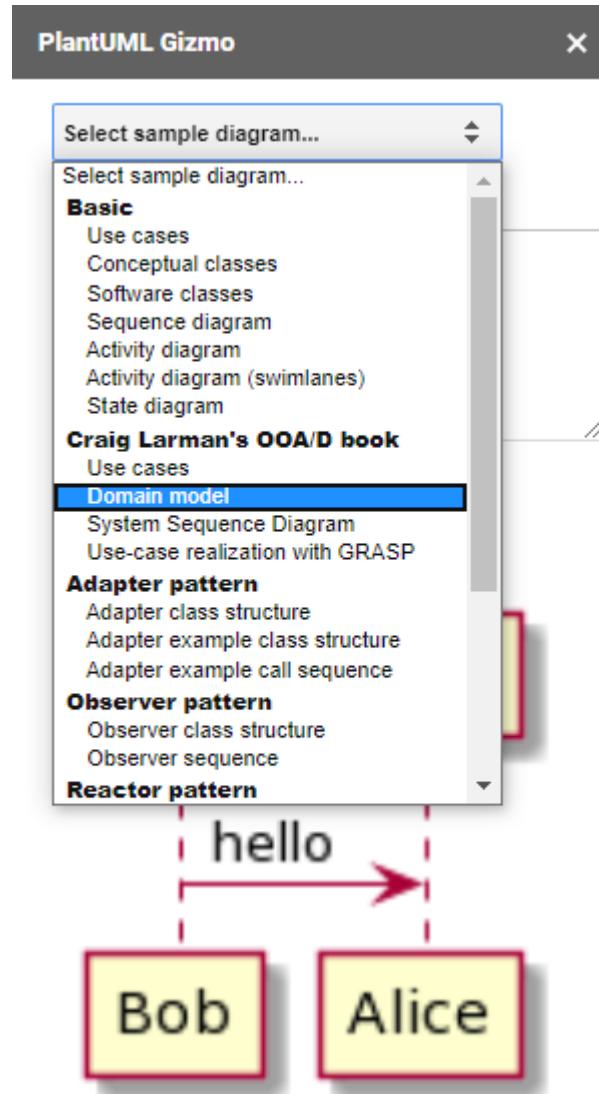


FIGURE 13.4. – PlantUML Gizmo offre plusieurs exemples de diagrammes UML.

13. Outils pour la modélisation UML

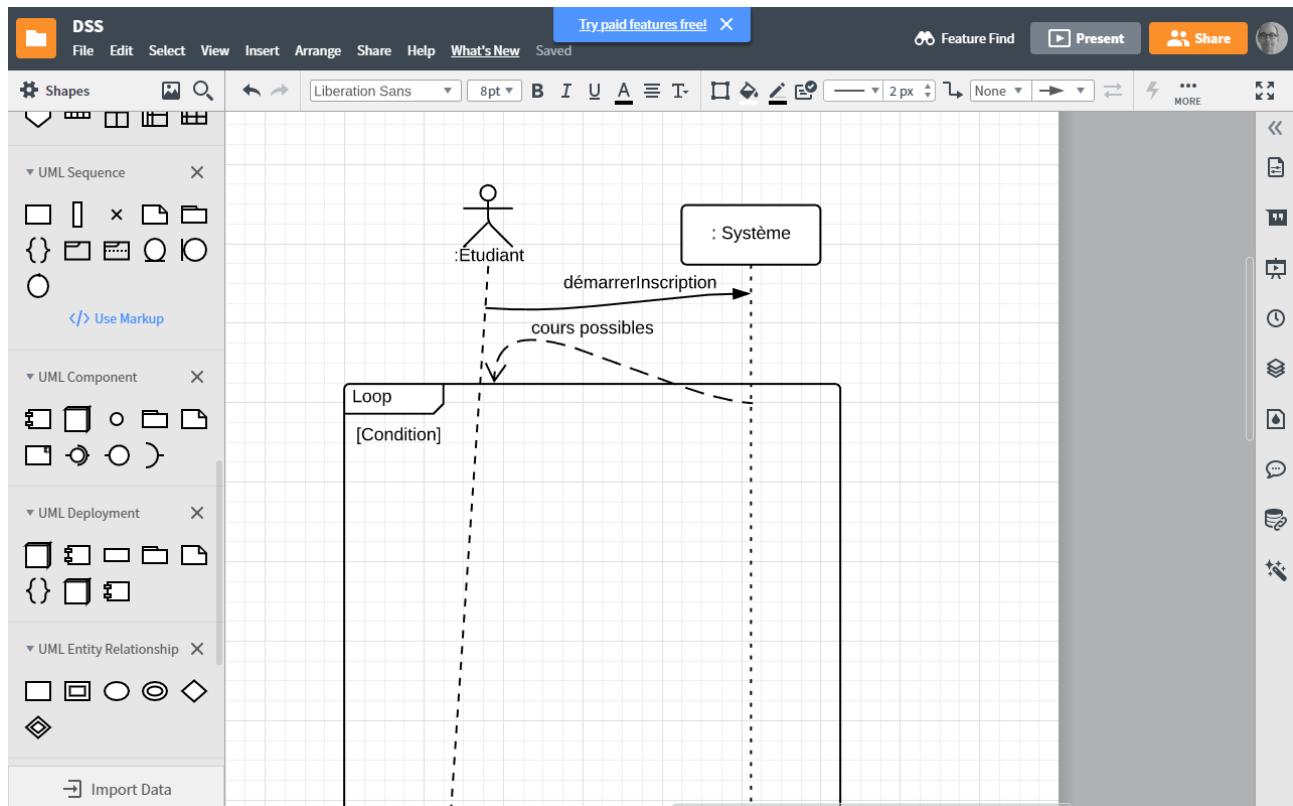


FIGURE 13.5. – Exemple de tentative de créer un diagramme de séquence système (DSS) avec Lucidchart. C'est principalement un éditeur graphique avec les éléments graphiques UML qui sont essentiellement des éléments graphiques composés. Il n'y a pas de sémantique UML dans l'outil. Par exemple, un « message » UML dans un diagramme de séquence dans Lucidchart 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 Lucidchart 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 [2] présente l'exemple du pattern Adaptateur pour les calculateurs de taxes (figure 14.1 tirée du livre de Larman, figure A26.1/F23.1).

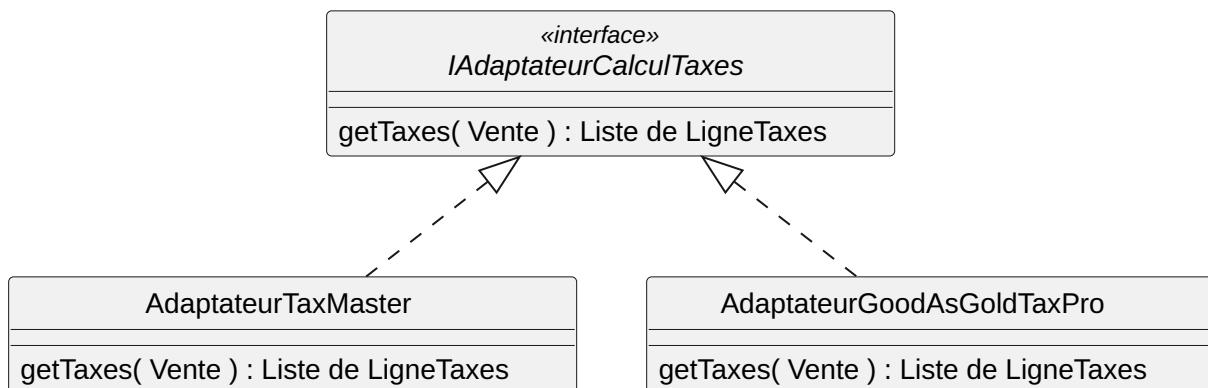


FIGURE 14.1. – Le pattern Adaptateur.

14.2. Imaginer le code sans le pattern GoF

Chaque principe GRASP est défini avec un énoncé d'un problème de conception et avec 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).

14. Décortiquer les patterns GoF avec GRASP

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 l'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 interagir avec le calculateur */
} else if(calculateurTaxes == "TaxMaster") {
    /* série d'instructions pour interagir 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 l'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 distincte et immuable) de chaque calculateur de taxes 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.

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 ? 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 éventuellement.

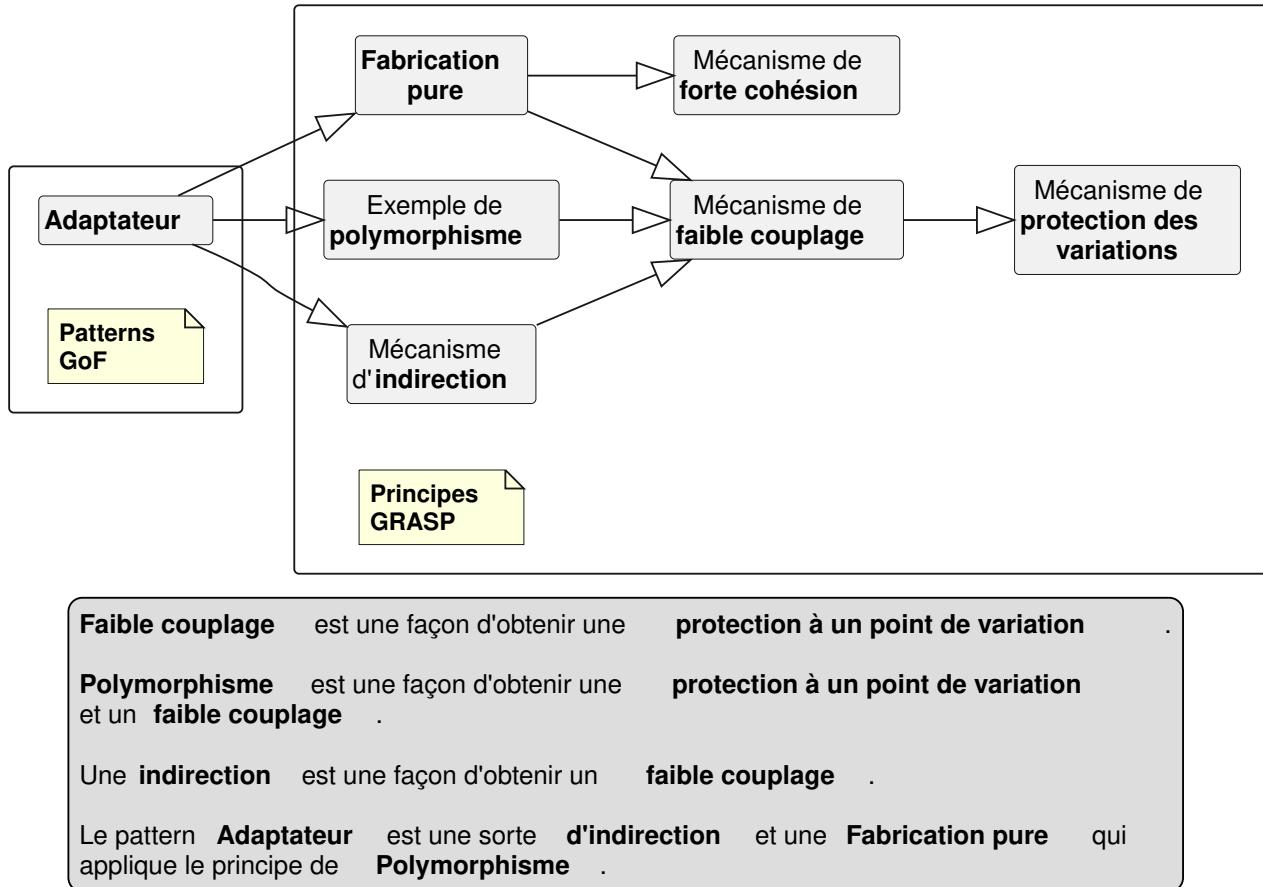


FIGURE 14.2. – Adaptateur et principes GRASP (voir Larman, 2005, fig. A26.3/F23.3)

14. Décortiquer les patterns GoF avec GRASP

Ensuite, on explique comment le principe (et le pattern GoF) résout le problème.

Consultez la figure 14.1 du pattern Adaptateur pour les sections suivantes.

14.3.1. Polymorphisme

Selon Larman (2005) :

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

14.3.2. Fabrication pure

Selon Larman (2005) :

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

Selon Larman (2005) :

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

Selon Larman (2005) :

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(euses) 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(euses) 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 (probablement) 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.

14. Décortiquer les patterns GoF avec GRASP

- GRASP Polymorphisme est relié à *Replace Type Code with Subclasses* et à *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 à *Move Function*.

14.5. Exercices

Pour ces exercices, suivez le modèle pour décortiquer le patron Adaptateur, illustré à la figure 14.2.

Une bonne ressource pour les patterns GoF est la suivante :

https://fuhrmanator.github.io/oodp-horstmann/htm/index_fr_en.html



Astuce

Il se peut que certains principes GRASP ne s'appliquent pas à un patron GoF !

Exercice 14.1 (Itérateur). Identifiez les 4 principes GRASP dans le patron **Itérateur**, selon les directives à la Section 14.3.

Exercice 14.2 (Observateur). Identifiez les 4 principes GRASP dans le patron **Observateur**, selon les directives à la Section 14.3.

Exercice 14.3 (Stratégie). Identifiez les 4 principes GRASP dans le patron **Stratégie**, selon les directives à la Section 14.3.

Exercice 14.4 (Composite). Identifiez les 4 principes GRASP dans le patron **Composite**, selon les directives à la Section 14.3.

Exercice 14.5 (Décorateur). Identifiez les 4 principes GRASP dans le patron Décorateur, selon les directives à la Section 14.3.

Exercice 14.6 (Méthode Template). Identifiez les 4 principes GRASP dans le patron Méthode Template, selon les directives à la Section 14.3.

Exercice 14.7 (Commande). Identifiez les 4 principes GRASP dans le patron Commande, selon les directives à la Section 14.3.

Exercice 14.8 (Méthode Fabrique). Identifiez les 4 principes GRASP dans le patron Méthode de fabrique, selon les directives à la Section 14.3.

Exercice 14.9 (Proxy). Identifiez les 4 principes GRASP dans le patron Proxy, selon les directives à la Section 14.3.

Exercice 14.10 (Façade). Identifiez les 4 principes GRASP dans le patron Façade, selon les directives à la Section 14.3.

Exercice 14.11 (Adaptateur pour Maps). Proposez une mise en oeuvre du patron GoF Adaptateur pour un système de livraison qui peut être configuré avec trois variantes du service de calcul d'itinéraires :

- Google Maps ;
- Bing Maps ;
- Apple Maps.

Le système veut obtenir une liste d'étapes (des directions) pour se rendre à une destination à partir d'un point de départ. L'utilisateur ou l'utilisatrice du système pourra décider lequel des services lui convient dans les préférences.

Le but de l'exercice est de déterminer l'interface stable (Protection des variations GRASP) étant donné les variantes des services de calcul d'itinéraires. Cela peut être un diagramme de classes réalisé avec PlantUML.

15. Fiabilité

Le chapitre A36/F30 2 présente le problème de la fiabilité pour le système NextGen POS. C'est le basculement vers un service local en cas d'échec d'un service distant.

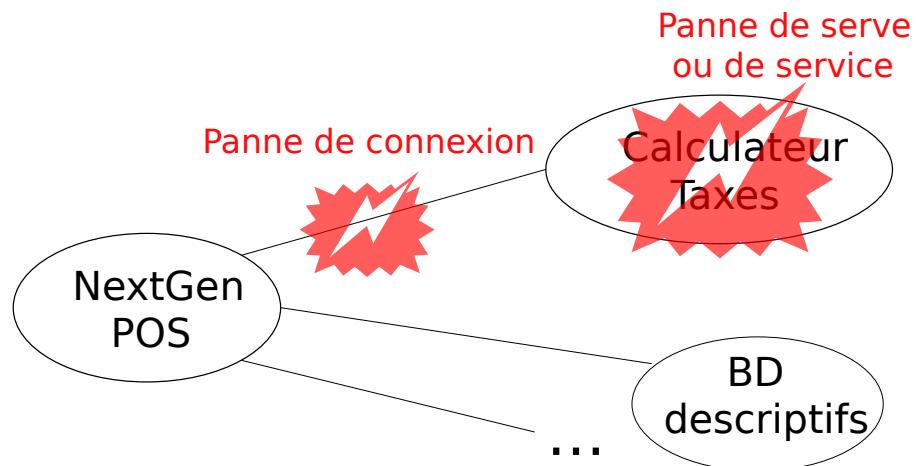


FIGURE 15.1. – Comment tolérer une panne de connexion ou de service ?

Voici les points importants :

- Définition des termes, A36.3/F30.3 2:
 - **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, A36.2/F30.2 2.
 - Utilisation d'*Adaptateur [GoF]* pour réaliser le service redondant (lecture d'information), A36.2/F30.2 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), A36.3/F30.3 2. Dans ce cas, il faut bien gérer les exceptions.
 - Utilisation de *Procuration (Proxy) de redirection [GoF]* pour basculer vers un service local en cas de panne (écriture d'information), A36.4/F30.4 2.

! Important

Faire une conception pour la fiabilité nécessite de l'expérience (ou l'utilisation des patterns). Un bon livre sur le sujet est celui de 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) pourraient affecter toute une grappe de serveurs.

15.1. Exercices

Exercice 15.1 (Faute, Erreur, Échec). Pour chaque scénario, indiquez la faute, l'erreur et l'échec (selon les définitions dans les notes de cours).

Scénario 1 : Guichet automatique

Lors de son contrôle d'un guichet automatique, une technicienne constate qu'il n'y a plus de billets. Alors, elle remet des billets de 20 \$ et de 50 \$. La prochaine personne qui utilise ce guichet automatique indique qu'elle veut retirer 100 \$, mais elle est surprise quand elle reçoit 5 billets de 50 \$ (pour 250 \$).

- Faute :
- Erreur :
- Échec :

Scénario 2 : Vaisseau spatial

Une mise à jour du logiciel de contrôle sur un vaisseau spatial est effectuée par un(e) ingénieur(e) sur Terre. Ce nouveau programme contient une mauvaise référence pour stocker les valeurs de configuration du réacteur. Lorsque le programme effectue ce stockage, il y a une corruption du programme qui contrôle l'orientation de l'antenne pour la communication avec la Terre. Ensuite, l'antenne ne pointe plus vers la Terre, et la communication est coupée à jamais.

- Faute :
- Erreur :
- Échec :

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
- nœud 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'activités dans le cadre d'un cours de programmation utilisant GitHub Classroom est dans la figure 16.2. Ce diagramme explique comment GitHub Classroom permet à l'étudiant(e) qui accepte un devoir (*assignment* en anglais) sur GitHub Classroom de choisir son identité universitaire, mais seulement si l'enseignant(e) a téléchargé la liste de classe *avant* d'envoyer les invitations aux étudiant(e)s.

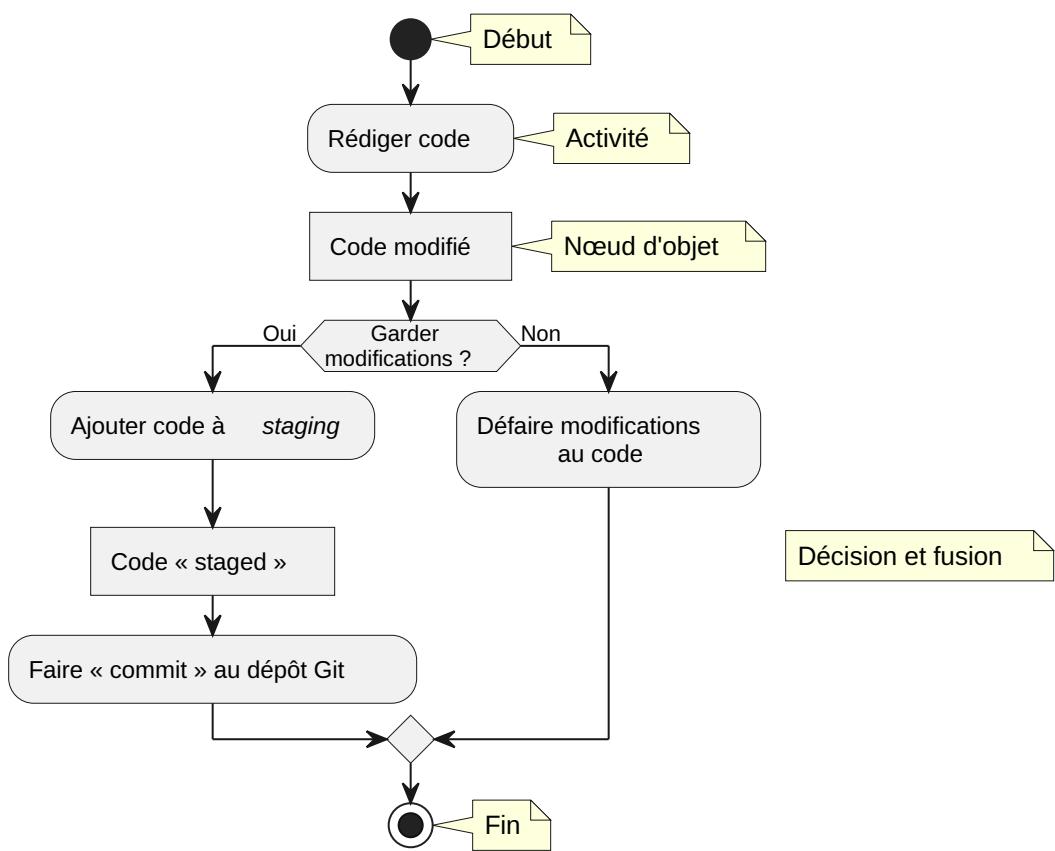


FIGURE 16.1. – Diagramme d’activité pour un processus simple avec Git. ([PlantUML](#))

16. Diagrammes d'activités

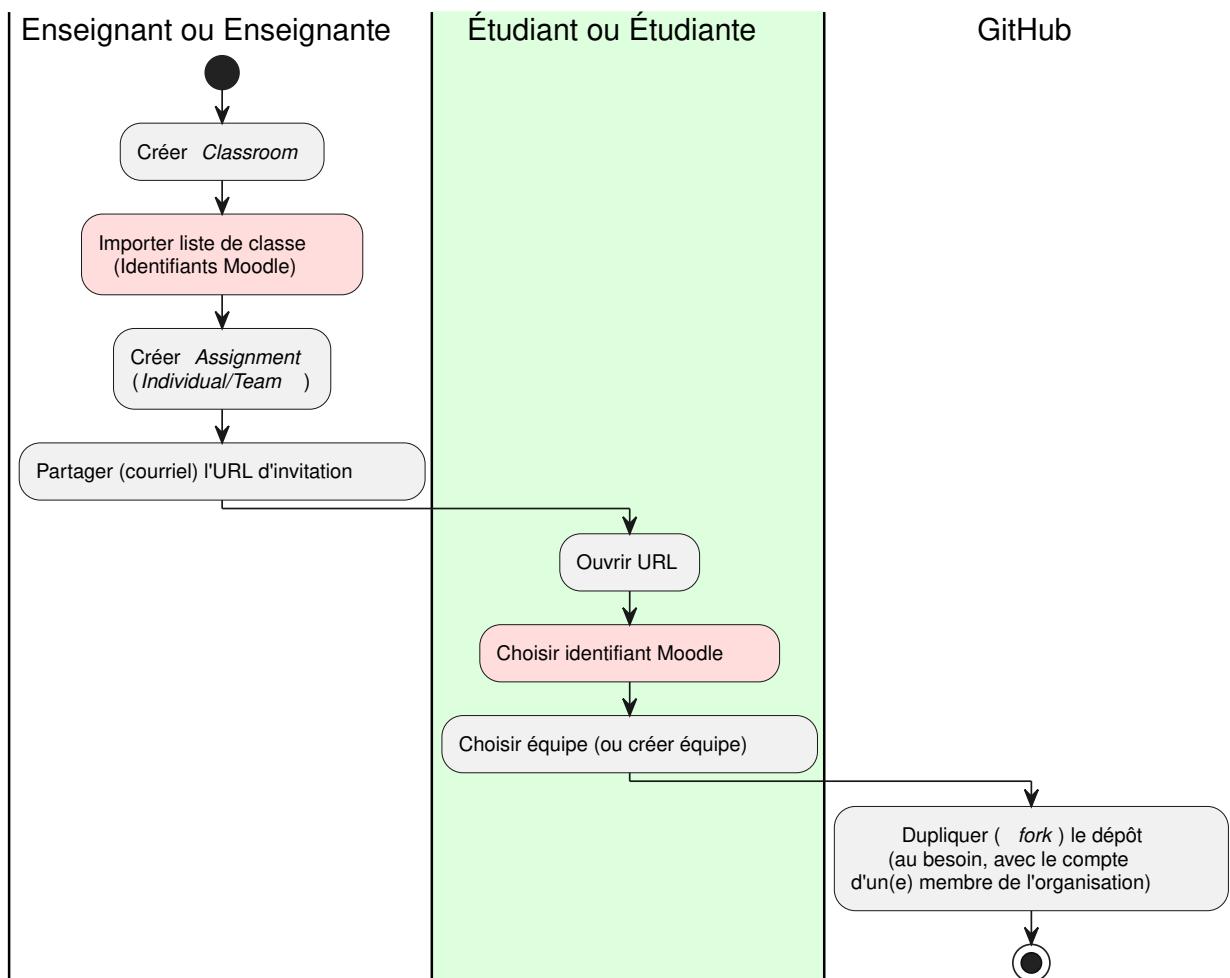


FIGURE 16.2. – Diagramme d'activités pour les activités séquentielles de GitHub Classroom (contexte de l'ÉTS avec Moodle). ([PlantUML](#))

16.2. Exercices

Ces exercices devraient vous aider à comprendre les diagrammes d'activités en UML. Vous devez vous référer au livre du cours pour la bonne notation (chapitre F25/A28) .

Note

Vous pouvez dessiner les diagrammes à la main et en prendre une photo avec une application comme Microsoft Lens ([Android](#), [iOS](#)).

Vous pouvez également utiliser PlantUML. Voici des ressources à ce propos :

- [tutoriel VS Code sur YouTube](#) ;
- [extension PlantUML pour VS Code](#) ;
- [PlantUML Gizmo](#), module supplémentaire Google Docs ;
- [PlantText.com](#).

Méfiez-vous des outils comme Lucidchart ayant seulement des profils superficiels pour UML (voir la figure [13.5](#)).

Exercice 16.1 (Location de voitures). Esquissez le diagramme d'activités lors de la réception de voitures louées (après la location) dans une compagnie. Pour le diagramme, faites attention à la **notation UML** : cela comprend les objets (pour la voiture et pour la facture), le début et la fin de l'activité, les débranchements, les jointures, les décisions et les fusions.

- Les rôles sont le client, le réceptionniste (qui gère la documentation et le paiement de la location) et l'agent (qui gère le traitement des voitures avant la prochaine location).
- Le client rend la voiture et les clés.
- Le réceptionniste note le kilométrage et le niveau d'essence pour calculer la facture.
- Le client paye sa location, selon le montant sur la facture.
- L'agent inspecte la voiture pour la propreté. Si elle n'est pas assez propre, alors l'agent doit laver, rincer et sécher l'extérieur et nettoyer l'intérieur. Ce travail devrait commencer le plus vite possible, après que le réceptionniste a fini de noter les informations pour la facture.

Voir [une solution avec PlantUML](#).

Exercice 16.2 (Soumission de devoir Moodle). Dessinez un diagramme d'activités qui modélise ce qui se passe lorsque vous faites un devoir dans Moodle. Votre diagramme doit comprendre les activités de l'enseignant et de l'étudiant (dans les partitions séparées). Le devoir doit être un *objet* dans le diagramme.

16. Diagrammes d'activités

Exercice 16.3 (Soumission de devoir Google Classroom). Dessinez un diagramme d'activités qui modélise ce qui se passe lorsque vous faites un devoir dans Google Classroom. Votre diagramme doit comprendre les activités de l'enseignant et de l'étudiant (dans les partitions séparées). Le devoir doit être un *objet* dans le diagramme.

17. Diagrammes d'états

Ce chapitre contient des informations sur les diagrammes d'états en UML. Ce 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'états sert à modéliser les comportements. Un concept préalable : **Automate fini W**.
- Un diagramme d'états contient les éléments suivants :
 - Événement :
 - occurrence d'un fait significatif ou remarquable ;
 - État :
 - 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 des *états imbriqués*.

La figure 17.1 est un exemple tiré du livre de Larman (2005) et est fait en PlantUML.

17. Diagrammes d'états

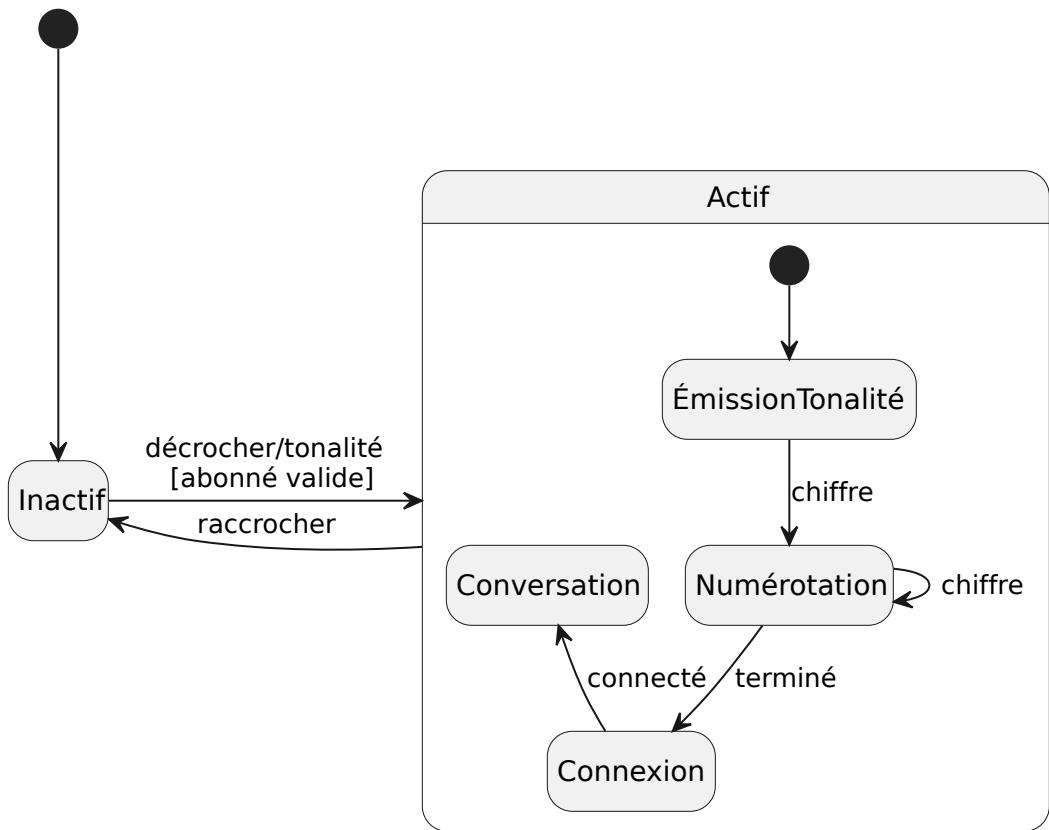


FIGURE 17.1. – Diagramme d'états (figure A29.3/F25.10). (PlantUML)

17.1. Exercices

Note

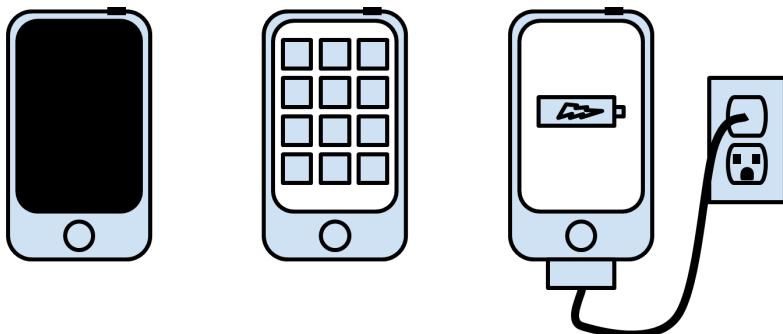
Vous pouvez dessiner les diagrammes à la main et en prendre une photo avec une application comme Microsoft Lens ([Android](#), [iOS](#)).

Vous pouvez également utiliser PlantUML. Voici des ressources à ce propos :

- [tutoriel VS Code sur YouTube](#) ;
- [extension PlantUML pour VS Code](#) ;
- [PlantUML Gizmo](#), module supplémentaire Google Docs ;
- [PlantText.com](#).

Méfiez-vous des outils comme Lucidchart ayant seulement des profils superficiels pour UML (voir la figure [13.5](#)).

Exercice 17.1 (États d'un téléphone). Faites un diagramme d'états en UML modélisant les états d'un téléphone intelligent. Considérez une dynamique simplifiée, avec seulement trois états correspondant aux images suivantes :



Pour simplifier encore le modèle : le bouton en haut à droite sert à éteindre et à allumer l'écran. Le téléphone est initialement éteint. Vous pouvez ignorer l'autre bouton rond au centre en bas. On peut brancher l'alimentation pour charger le téléphone à tout moment, mais le bouton n'a aucun effet sur l'écran lorsque le téléphone est connecté à l'alimentation. Lorsque l'on débranche l'alimentation, l'écran est toujours éteint.

Exercice 17.2 (Guichet automatique). Faites un diagramme d'états en UML qui correspond au système décrit par les cas d'utilisation suivants (format bref) :

S'authentifier. Le Client arrive à un guichet automatique bancaire, car il désire effectuer une transaction sur son compte. Le Client introduit sa carte bancaire, et le système attend qu'il saisisse

17. Diagrammes d'états

le NIP de la carte. Si le NIP est valide pour la carte, alors le système est prêt pour accepter d'autres actions. Sinon, le système enregistre la mauvaise tentative et demande de nouveau au Client de saisir son NIP. À tout moment que le système possède la carte du Client, ce dernier peut annuler la session pour récupérer sa carte.

Gérer guichet. L'Administrateur démarre le système, et le système attend l'introduction d'une carte bancaire du Client. Quand le système est dans cet état, l'Administrateur peut aussi l'éteindre.

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 en 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 packages ;
- Organiser les packages par **cohésion** ;
- Organiser les packages en 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 packages.

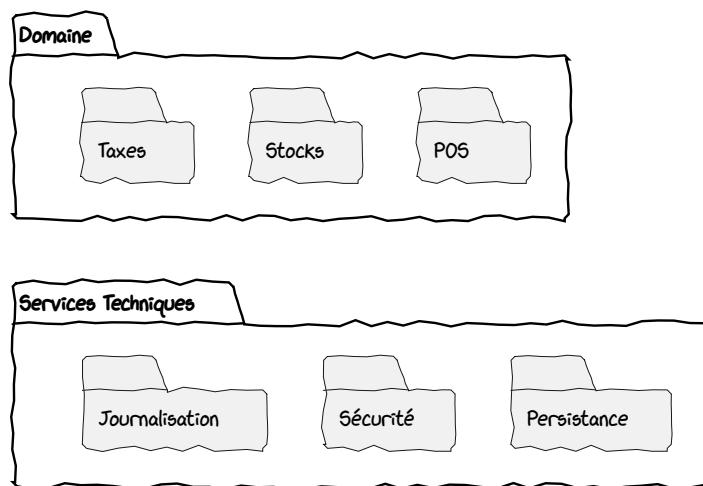


FIGURE 18.1. – Diagramme de packages (tiré de la figure F12.6 ). (PlantUML)

18.1. Absence de packages 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 noms** : 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, par exemple `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 noms, 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 les *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 le voyez dans l'exemple avec les commentaires dans le code de certains fichiers, par exemple `/// <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 (par exemple, 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 elle-même un service pour héberger et exécuter d'autres logiciels, par exemple :
 - un système d'exploitation (OS), est un logiciel qui héberge et qui exécute des programmes ;
 - une machine virtuelle (JVM ou .NET) ;
 - un moteur de base de données (ex. PostgreSQL) exécute les requêtes SQL ;
 - un navigateur Web qui héberge et qui exécute JavaScript, les applets Flash/Java ;
 - un moteur de *workflow* ;
 - un 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.

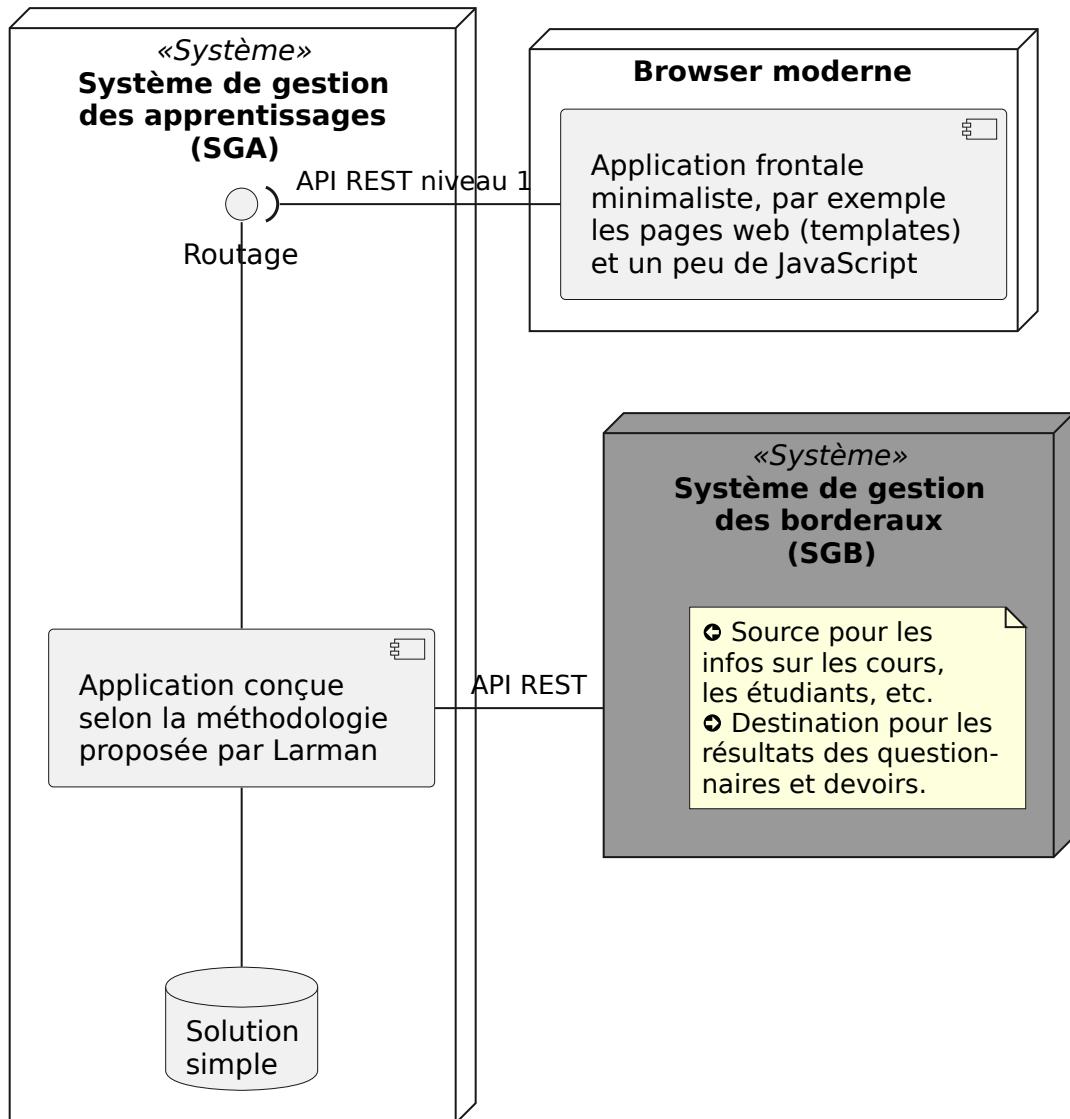


FIGURE 19.1. – Diagramme de déploiement du système à développer pour le laboratoire. ([PlantUML](#))

19. Diagrammes de déploiement et de composants

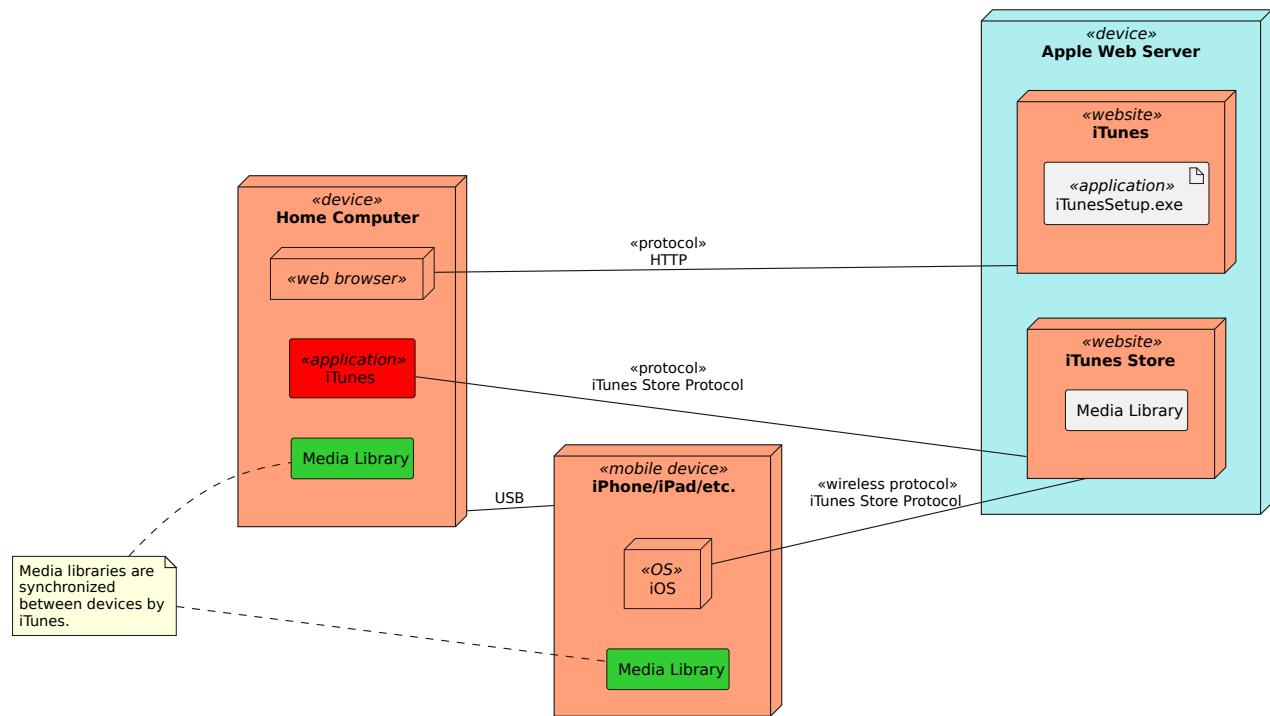


FIGURE 19.2. – Diagramme de déploiement pour iTunes d'Apple, inspiré de [ceci](#). (PlantUML)

20. Laboratoires

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

20.1. TypeScript

Qu'est-ce que TypeScript ? Selon Goldberg ([2022](#)), TypeScript comprend quatre éléments :

- Un **langage de programmation** ayant toute la syntaxe de JavaScript, plus une syntaxe propre à TypeScript pour définir et utiliser les types ;
- Un **vérificateur de type** pouvant décortiquer un ensemble de fichiers en JavaScript et en TypeScript définissant des variables, les fonctions, etc., afin d'indiquer des problèmes éventuels de configuration ;
- Un **compilateur** qui 1) invoque le vérificateur de type, 2) signale des problèmes le cas échéant et 3) finalement génère du code en JavaScript équivalent ;
- Un **service de langage** qui transmet des informations provenant du vérificateur de type à l'IDE pour que ce dernier puisse faciliter le développement.

20.2. JavaScript/TypeScript

Pour la personne ayant déjà des connaissances Java et des patrons de conception de la Bande des quatre Gamma et coll. ([1994](#)), il est recommandé d'apprendre les choses dans cet ordre :

- **JavaScript** : un tutoriel intéressant (et libre) est sur [fr.javascript.info](#). Nous vous recommandons 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, par exemple `string | number`.
- JavaScript a des notions de « *truthy* » et « *falsy* » (conversion d'une expression en 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.

20.3. 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 des 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 »).

20.3.1. 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)

20.3.2. 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.

Mise en garde

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 l'on fait `if (maVariable)` pour tester si une variable est définie, si la variable est définie et que sa valeur est 0, la condition sera `false`.

20.4. Git

Git est un logiciel de gestion des versions permettant de stocker un ensemble de fichiers en conservant la chronologie de tous les changements ayant été effectués dessus. Ce genre de logiciel permet de retrouver les différentes versions d'un lot de fichiers connexes. Depuis 2010, Git est le logiciel de gestion des versions le plus populaire, disponible sur les environnements Windows, Mac et Linux. Il s'agit d'un logiciel libre et gratuit, créé en 2005 par Linus Torvalds, fondateur du noyau Linux. Linus Torvalds prononce Git avec un g dur ([Google, 2007](#)).

Git est particulier parce qu'il est décentralisé, utilisant un système de connexion pair à pair. Les fichiers informatiques sont stockés sur l'ordinateur de chaque personne qui contribue au projet et ils peuvent également l'être sur un serveur dédié comme [GitHub](#), [GitLab](#), [BitBucket](#), etc.

Puisque chaque personne est libre à modifier les fichiers comme elle veut, sans être bloquée par les autres contributeurs, il est nécessaire de synchroniser et de fusionner les contributions de temps en temps. Cette synchronisation peut être plus ou moins compliquée, selon les travaux réalisés par chacun, par exemple lorsque deux personnes ont modifié un même fichier sur son ordinateur. Une des forces de Git est l'ensemble des fonctionnalités permettant de gérer tous les cas synchronisation. Mais cet avantage peut aussi faire en sorte que Git soit compliqué à utiliser. La figure 20.1 présente un survol des concepts et opérations de base de Git.

Ce manuel ne rentre pas dans les détails de chaque opération Git ; il existe plusieurs tutoriels pour Git sur Internet. Il y a beaucoup de scénarios où Git peut être utile, par exemple pour récupérer une version antérieure d'un fichier ou pour savoir qui a apporté une modification à un fichier, etc. Apprendre tous les cas d'utilisation de Git serait long et pas très intéressant. Il n'est pas nécessaire de tout comprendre pour commencer.

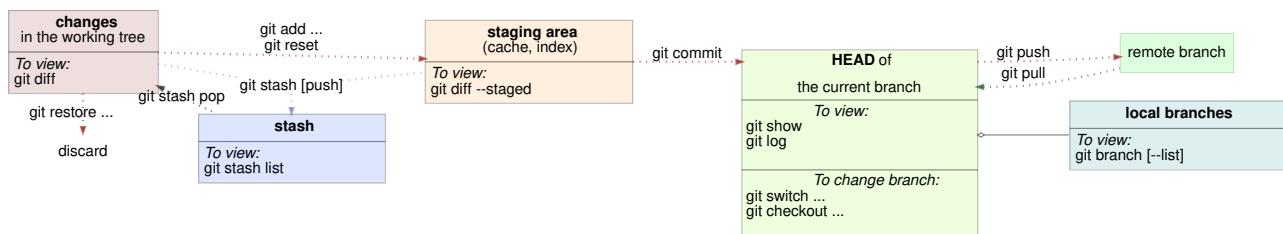


FIGURE 20.1. – Concepts et opérations de base de Git. (Cette œuvre, « Concepts et opérations de base de Git », est un dérivé de « [Basic git concepts and operations](#) » de Costa Shulyupin, utilisé sous [EPL](#).

20.5. 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 utilisent 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(e) (activité Git);
2. Le nombre de lignes par auteur(e) encore présentes et intactes dans la version HEAD.

Vous pouvez voir un exemple de rapport à la figure 20.2.

20. Laboratoires

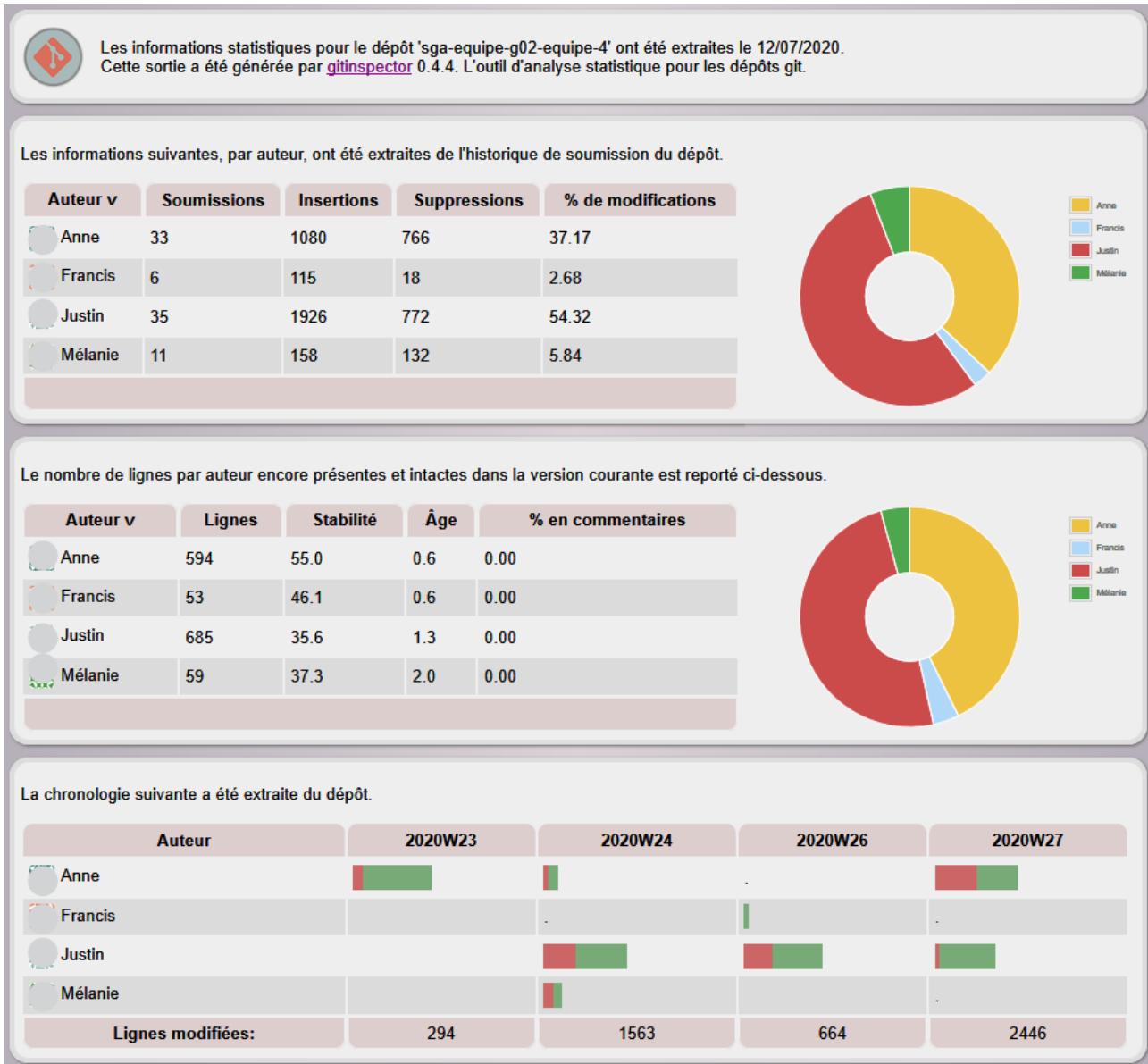


FIGURE 20.2. – Exemple de rapport généré par [gitinspector](#).

20.5.1. Faire le bilan de la contribution de chaque membre

Après l'évaluation à la fin de chaque itération, il est important de considérer combien chaque membre 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 20.2 que Anne et Justin ont fait une contribution beaucoup plus importante que Francis et Mélanie. Dans le bilan de l'itération, **on doit indiquer explicitement ce fait, même avec des pourcentages**.

! Important

Une phrase vague comme « des membres 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.

20.5.2. Proposer des solutions au 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 des « mollassons » ([Oakley, Felder, Brent, & Elhajj, 2004b](#) ([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 membre de l'équipe.
- Est-ce qu'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 chaque membre 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 membres 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 membres de l'équipe faire plus de travail peut nuire à la valeur pédagogique des laboratoires (ça peut faire mal à l'examen final pour les membres 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(e) membre qui travaille beaucoup plus que les autres éprouve un problème d'épuisement à cause du fait qu'il (elle) travaille trop.

- Est-ce que tout le monde utilise un moyen de communication 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 gérerez les problèmes de manière plus proactive.

20.5.3. FAQ pour gitinspector

Q : Comment fusionner le travail réalisé par une même personne, 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>
```

Par exemple, le `.mailmap` initial qui contient quatre 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(e) chargé(e) de laboratoire (par exemple le clone initial dans GitHub Classroom) ?

20.5. Évaluer les contributions des membres de l'équipe

R : La solution est d'ajouter le nom de l'auteur(e) 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(e) auteur(e) contenant 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

Deuxième partie. Exercices

21. Critique d'une conception

Dans cet exercice, l'objectif est de vous sensibiliser à la facilité de comprendre une conception à partir d'un problème. Un objectif secondaire est de considérer les choix de conception sur le plan de la cohésion et du couplage. Ici, il s'agit du jeu Monopoly, qui est un exemple proposé par Larman (2005), pour lequel il a également proposé un modèle du domaine et une conception, selon la méthodologie.

Pour cet exercice, nous examinerons une conception orientée objet réelle du jeu Monopoly disponible sur GitHub, soit [Emojiopoly](#). Voici le travail à faire.

- Considérez les deux artefacts :
 - un [modèle du domaine de Monopoly](#) proposé par Larman (2005) (il y a une version en français et une en anglais, puisque le code TypeScript est en anglais) et
 - un [modèle d'une solution](#) sous forme de diagramme de classes logicielles, créé à partir du code TypeScript dans le dépôt mentionné ci-haut).
- Nous faisons une hypothèse que l'équipe qui a développé Emojiopoly n'a pas commencé avec le MDD de Larman.
- Comparez ces deux artefacts et faites des remarques sur la conception, **surtout par rapport au MDD et au décalage des représentations**.
- Faites des remarques sur la solution concernant la cohésion et le couplage.

Diagramme de classes d'Emojiopoly

Pour visualiser la conception, nous avons généré un diagramme des classes en UML avec l'outil [tplant](#) :

Modèle du domaine de Monopoly

Puisque la solution Emojiopoly est en anglais, vous pouvez regarder le modèle du domaine de Monopoly en français et en anglais pour vous aider à comprendre les termes.

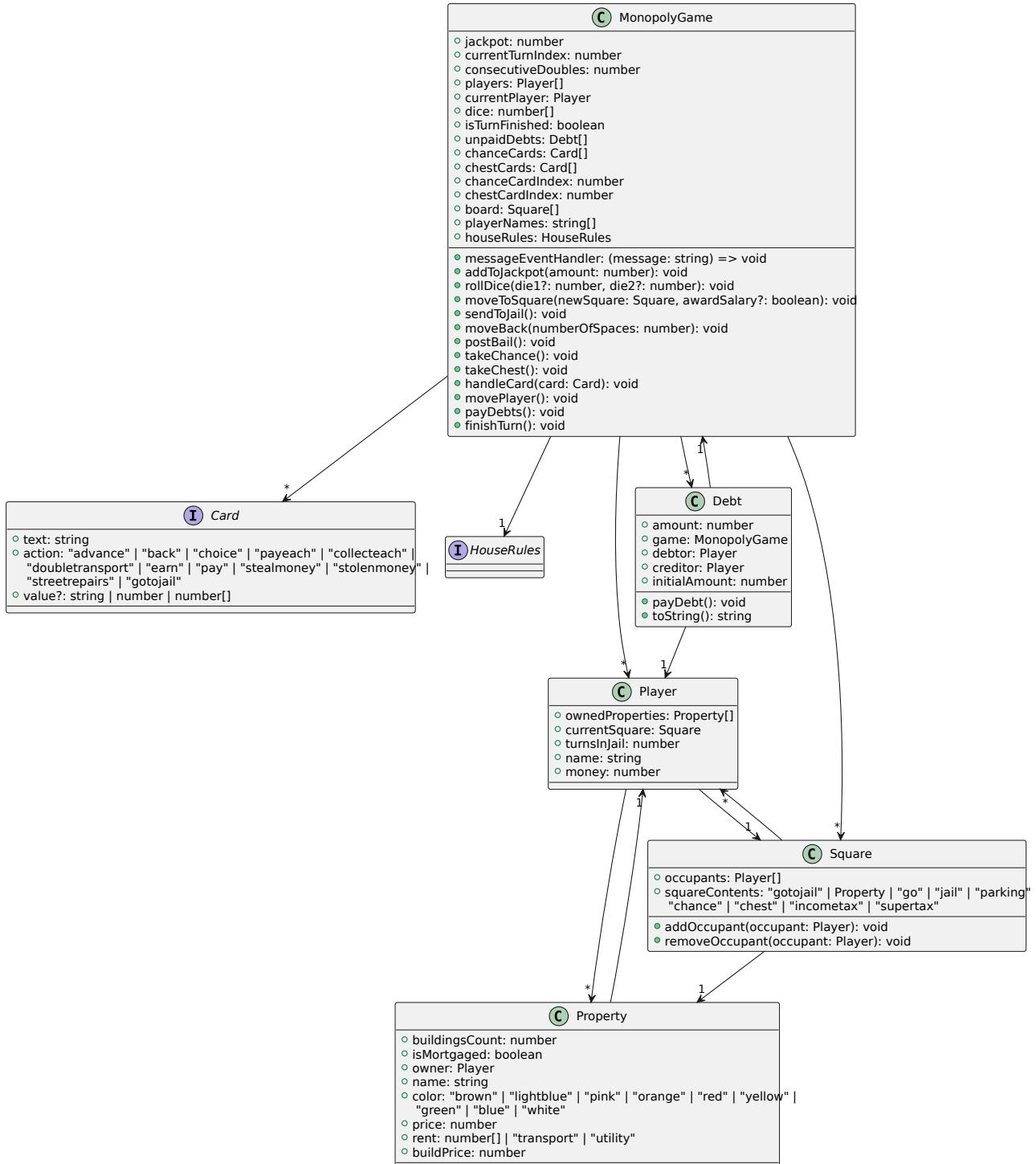


FIGURE 21.1. – Diagramme de classes logicielles (TypeScript) pour le projet [Emojopoly].(<https://github.com/Chuzzy/Emojopoly>)

21. Critique d'une conception

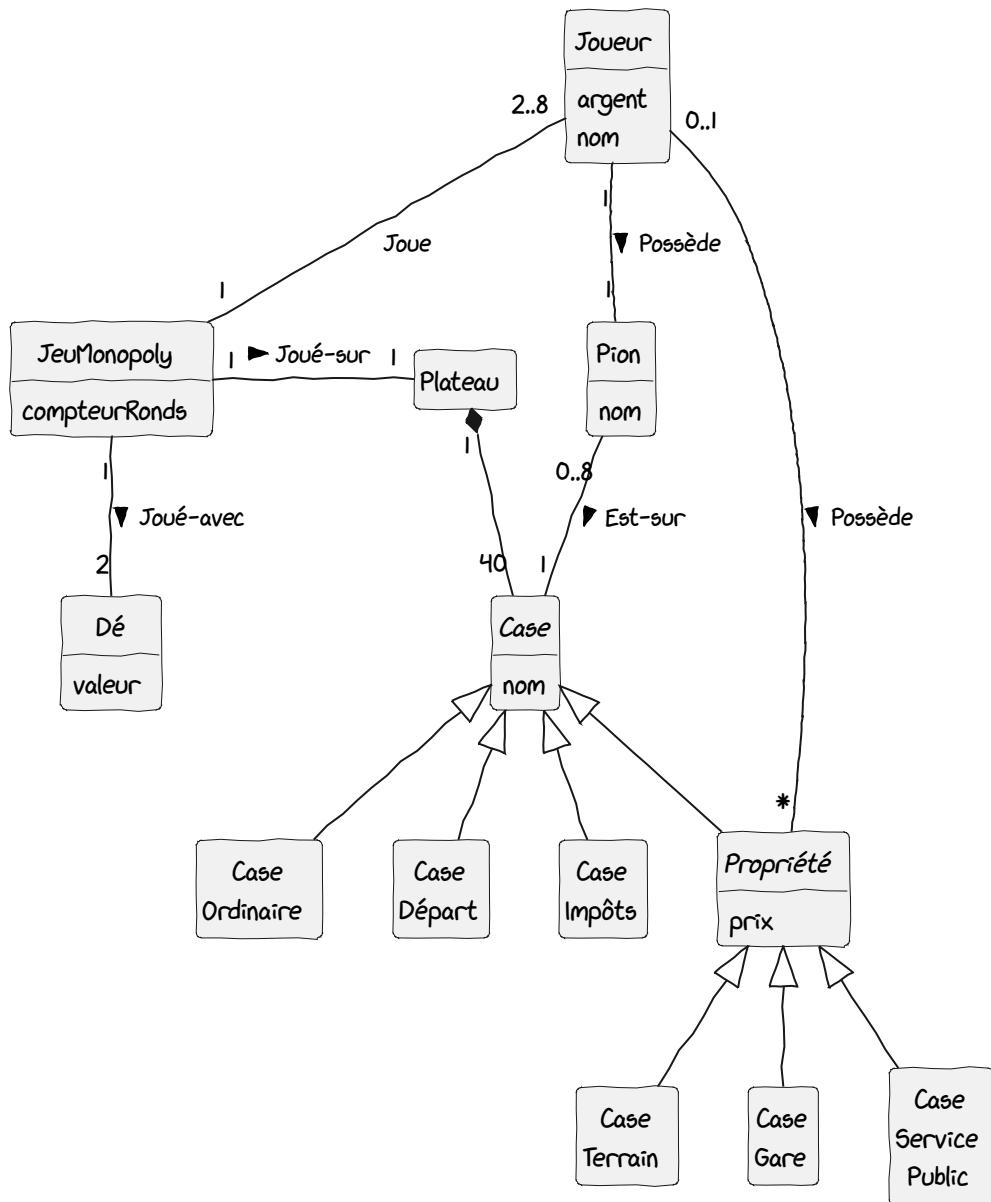


FIGURE 21.2. – MDD (version française) de Monopoly proposé par Larman (2005)

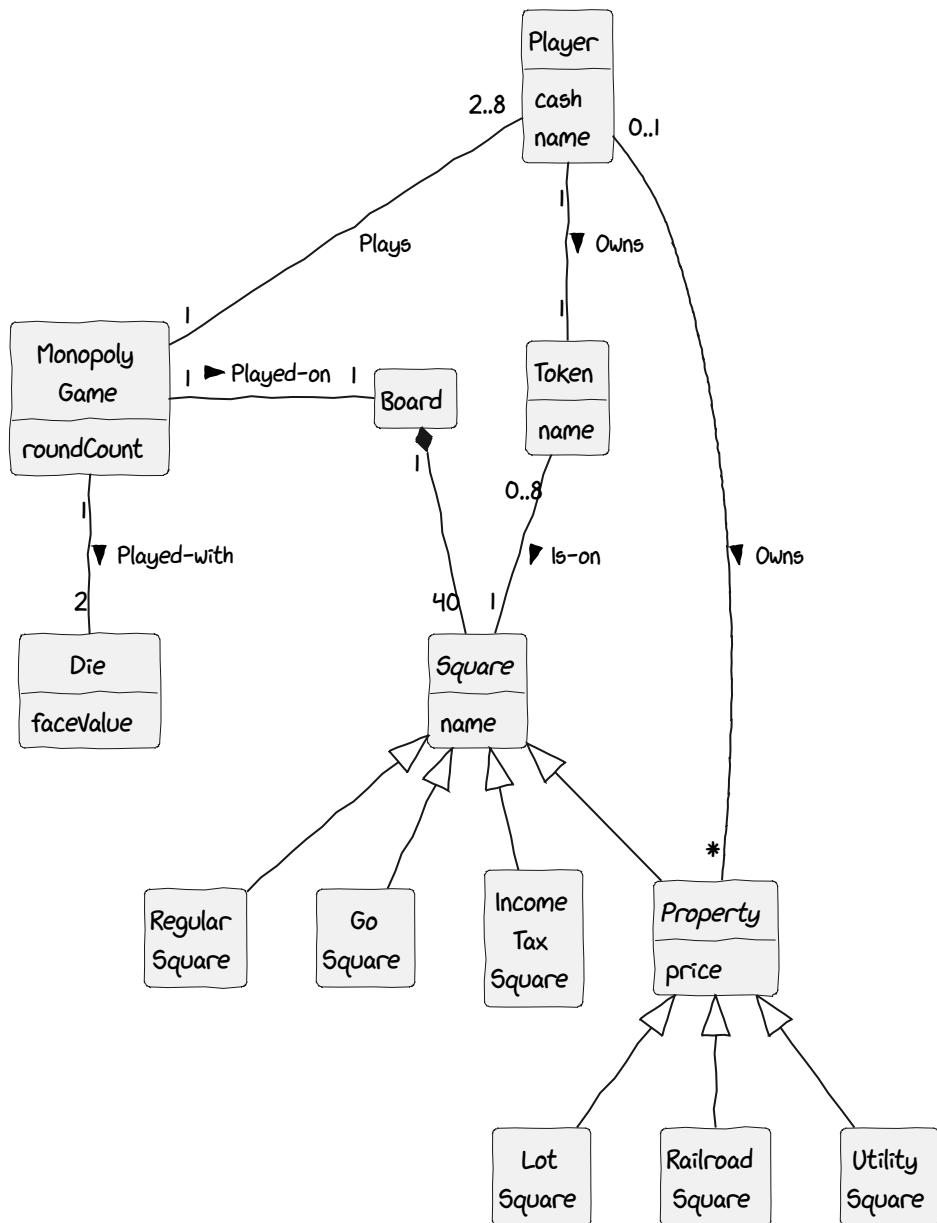


FIGURE 21.3. – MDD (version anglaise) de Monopoly proposé par Larman (2005)

22. Coder des méthodes à partir des diagrammes de séquence

Pour chacun des diagrammes suivants, écrivez les classes TypeScript avec les méthodes indiquées dans le diagramme.

(Cet exercice complémente le livre de [Larman, 2005](#) à la section F18.6/A20.4.).

Astuce

Vous pouvez utiliser VS Code pour vous aider avec le TypeScript, mais cet outil ne sera pas forcément permis lors d'un examen.

Voici un modèle à suivre. Pour le diagramme suivant :

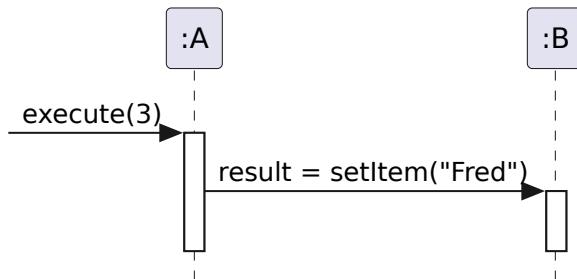


FIGURE 22.1. – Exemple de diagramme de séquence.

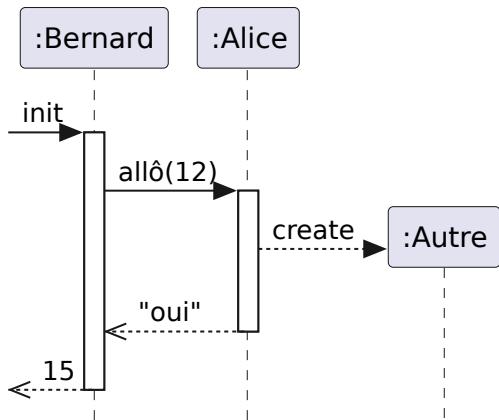
On code les classes suivantes en TypeScript :

```
class A {
    b: B; // A envoie un message à B, visibilité d'attribut
    execute(arg0:number):any {
        const result = this.b.setItem("Fred");
    }
}

class B {
    setItem(arg0:string):any {
```

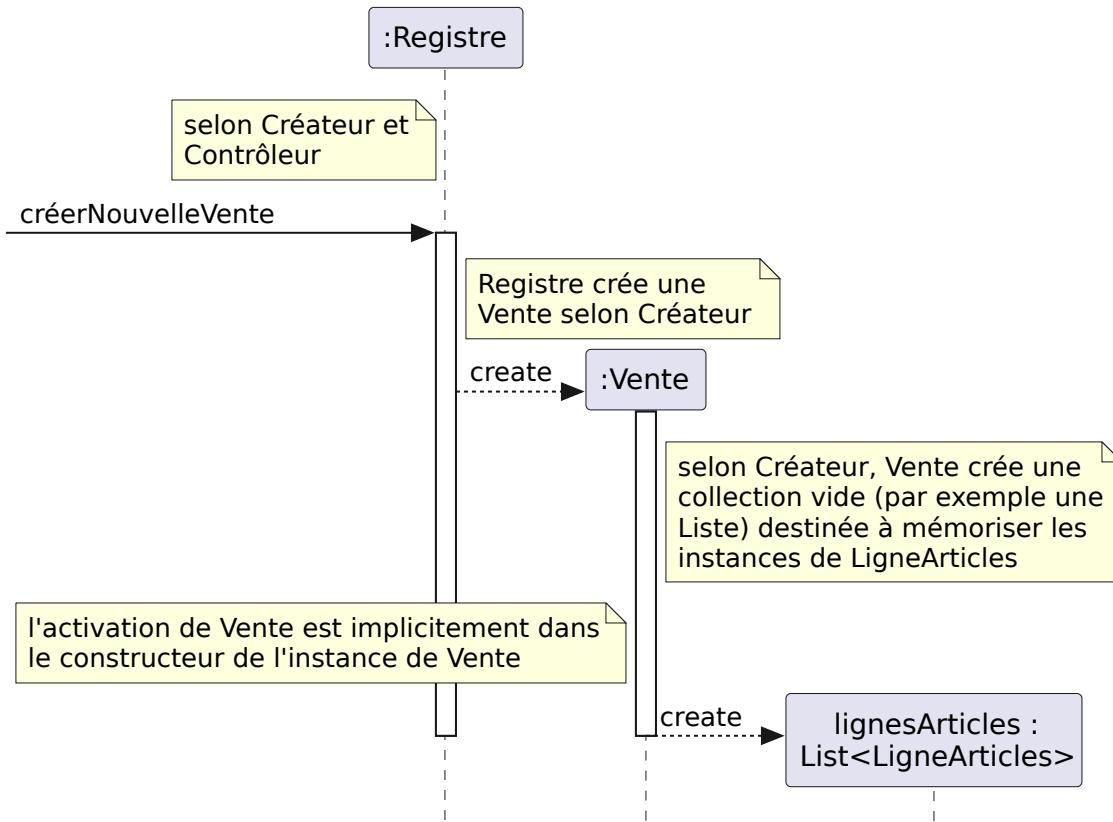
```
//...
}
}
```

1. Écrivez le code pour la figure suivante.

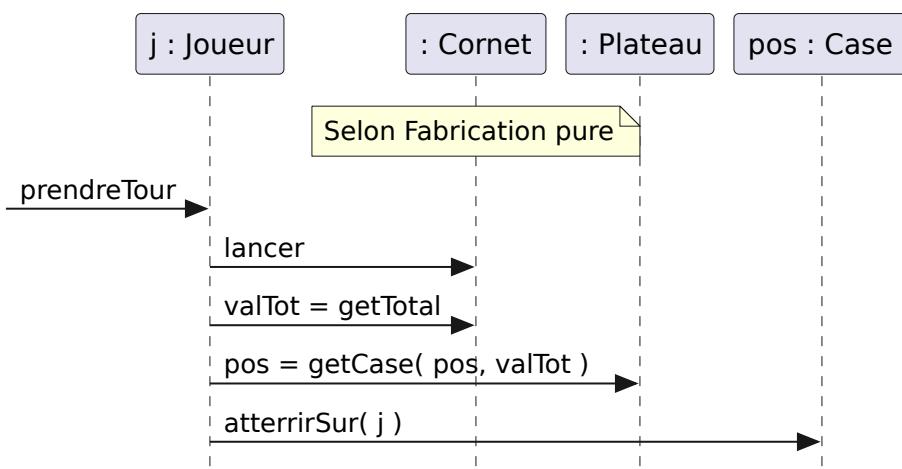


22. Coder des méthodes à partir des diagrammes de séquence

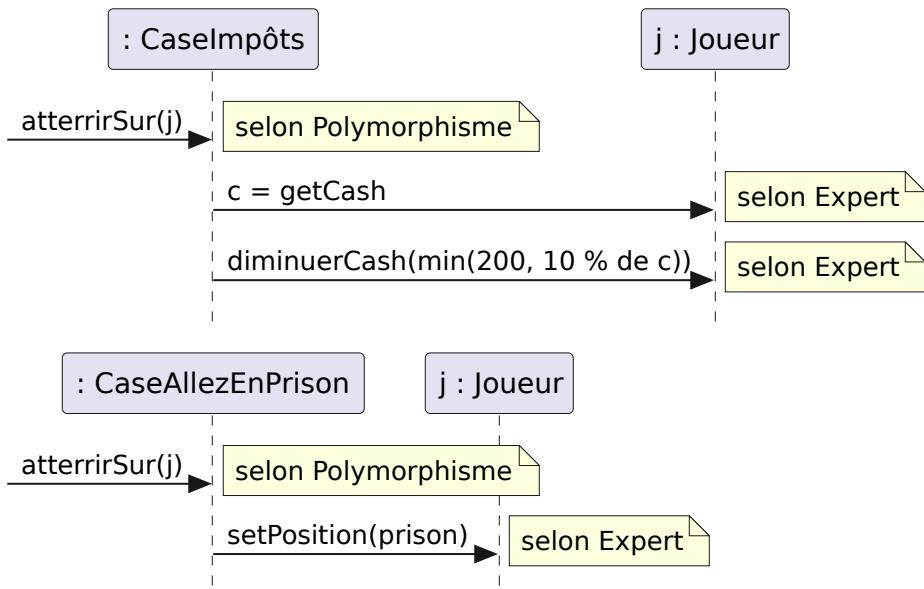
2. Écrivez le code pour la figure suivante décrivant la création de la collection de Vente (tirée de Larman, 2005, figure. 17.6).



3. Écrivez le code pour la figure suivante décrivant l'utilisation d'un *Cornet* dans le jeu de Monopoly (tirée de Larman, 2005, fig. F22.9)



4. Écrivez le code pour les figures suivantes décrivant les appels polymorphes de la méthode `atterrirSur` dans le jeu Monopoly (tirées de Larman, 2005, fig. F22.6 et F22.7)



A. Cas d'utilisation - Réserver un livre de la bibliothèque

Parties prenantes et intérêts :

- **La personne membre.** Elle veut un moyen de recherche exact et rapide et ne veut pas que la Bibliothèque mémorise des informations sur ses recherches (confidentialité). Elle veut pouvoir réaliser des réservations aisément, obtenir un service rapide en fournissant un minimum d'efforts. Elle veut également une preuve de réservation.
- **La Bibliothèque.** Elle veut enregistrer correctement les réservations et satisfaire les souhaits des membres.

Préconditions : La personne membre est identifiée et authentifiée

Acteur principal : Personne membre

1. La personne membre choisit la fonction « recherche » et saisit du texte décrivant le livre (par exemple, une partie du titre, « UML »).
2. Le système affiche une liste de livres (le titre, l'auteur(e) et l'année) correspondant à la recherche, par exemple « UML 2 et les design patterns, Craig Larman, 2005 » et « UML par la pratique, Pascal Rocques, 2009 ».
3. La personne membre choisit un livre parmi les résultats, par exemple « UML 2 et les designs patterns, Craig Larman, 2005 ».
4. Le système affiche les informations détaillées du livre (le titre, l'auteur(e), le numéro ISBN, la maison d'édition, le numéro de l'édition et l'année) ainsi que la liste de tous les exemplaires du livre indiquant s'ils sont disponibles ou pas, par exemple deux exemplaires du livre « UML 2 et les design patterns », un avec l'identificateur d'exemplaire « 1 » qui est disponible et un avec l'identificateur d'exemplaire « 2 » qui n'est pas disponible.
5. La personne membre réserve un exemplaire du livre qui est disponible.
6. Le système confirme la réservation en affichant un numéro de réservation avec le nom du membre et le code de l'exemplaire du livre.

Extensions (scénarios alternatifs) :

2. Aucun livre ne correspond au texte de la recherche.
 1. Le système affiche un message indiquant qu'aucun livre n'a été trouvé.
 2. La personne membre lance une nouvelle recherche.
4. Tous les exemplaires sont indisponibles.

A. Cas d'utilisation - Réserver un livre de la bibliothèque

1. Le système affiche toutes les informations du livre et des exemplaires, mais un message indique qu'il n'est pas possible de réserver, faute d'exemplaires disponibles.
2. La personne membre lance une nouvelle recherche.

B. Cas d'utilisation - Traiter une vente

Voici des artefacts en PlantUML proposés pour le cas d'utilisation *Traiter une vente* documenté par Larman (2005).

B.1. DSS

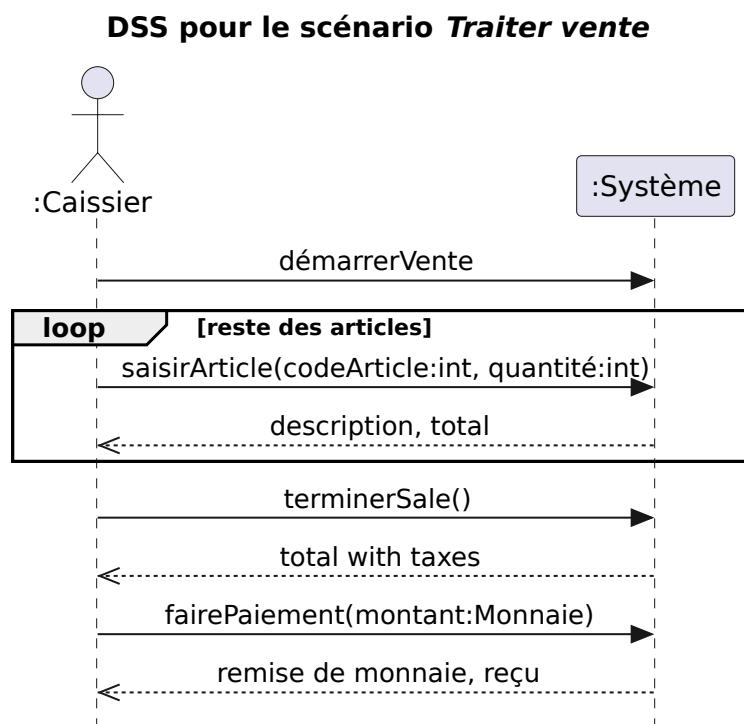
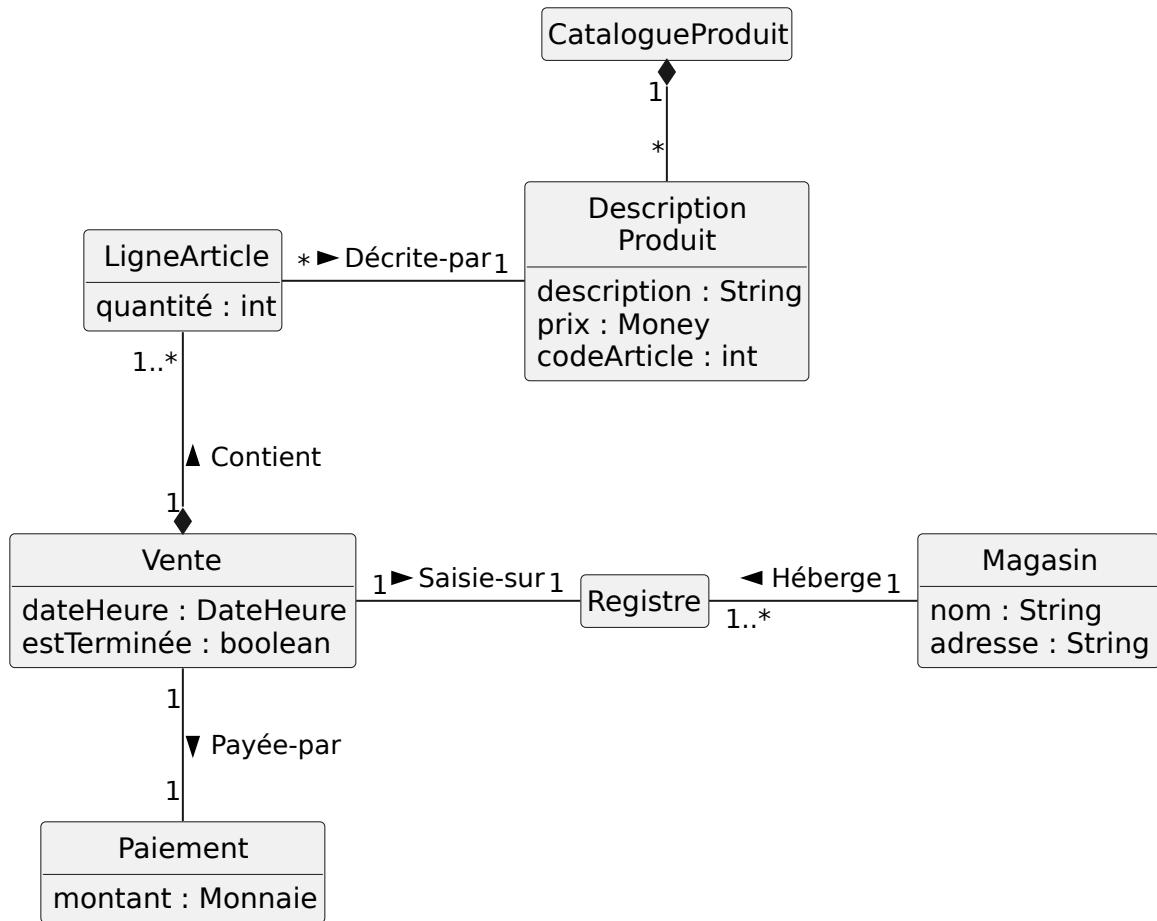


FIGURE B.1. – DSS pour le scénario Traiter une vente de Larman (2005).

B. Cas d'utilisation - Traiter une vente

B.2. MDD partiel



C. Cas d'utilisation - Ouvrir la caisse

Le contexte est l'étude de cas du système POS NextGen, notamment le scénario principal du cas d'utilisation *Ouvrir la caisse* (en anglais *Cash In*). C'est l'acte d'un caissier ou d'une caissière qui arrive avec son tiroir-caisse contenant déjà de l'argent, qui s'authentifie sur la caisse (le registre) et qui saisit le montant en espèces dans le tiroir-caisse.

i Note

En effet, ce cas d'utilisation n'est pas présenté en détail par Larman (2005), bien qu'il soit mentionné dans son livre dans un diagramme de cas d'utilisation. Alors, nous proposons pour cet exercice des modifications au modèle du domaine présenté par Larman afin de modéliser cette fonctionnalité additionnelle du logiciel.

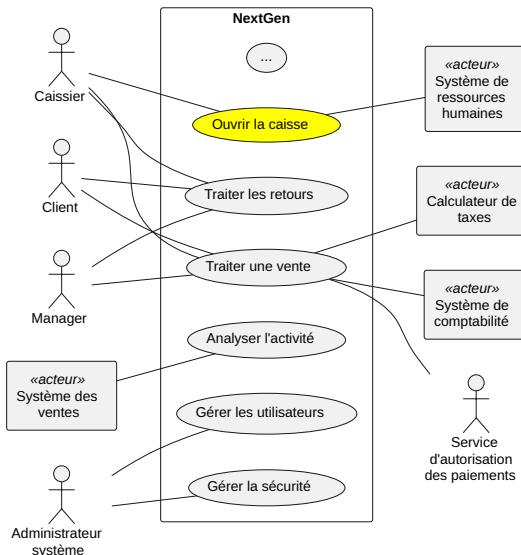


FIGURE C.1. – Diagramme de cas d'utilisation pour le système NextGen.

C.1. Terminologie

Quelques termes du domaine d'affaires doivent être compris avant de procéder :

C. Cas d'utilisation - Ouvrir la caisse

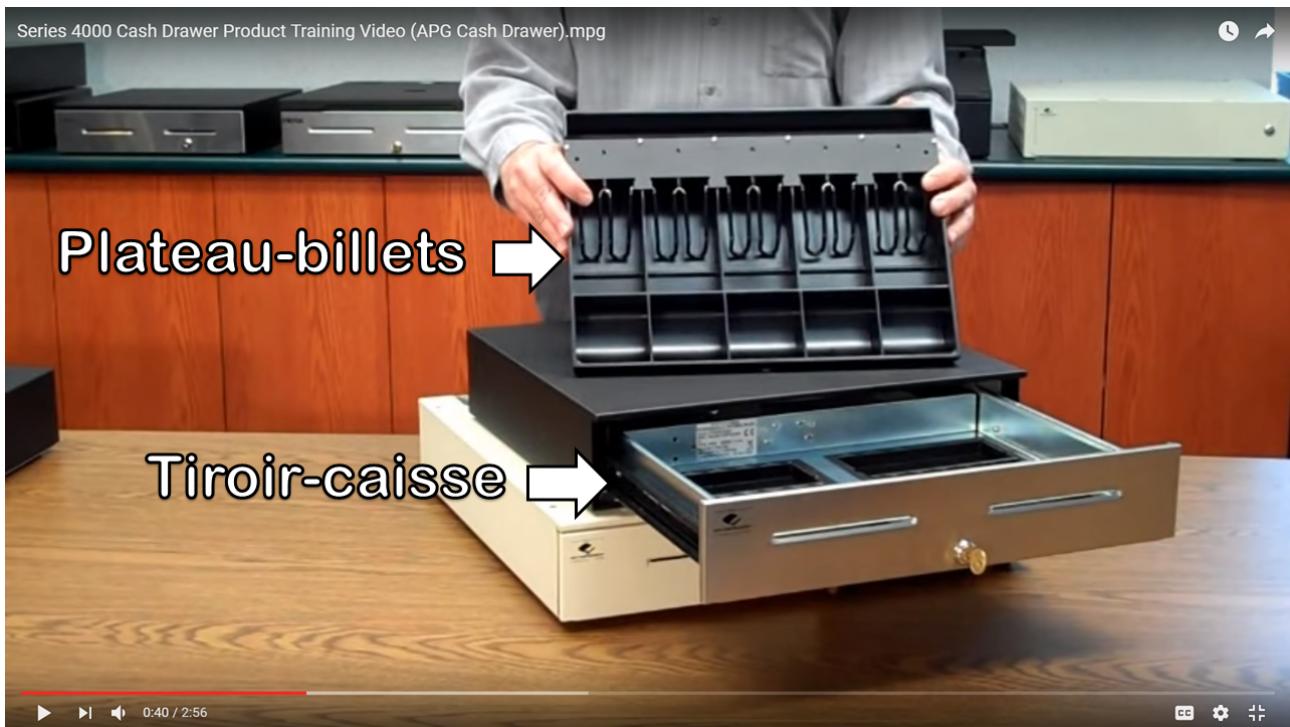


FIGURE C.2. – Plateau-billets et tiroir-caisse. Capture d'écran de la vidéo sur YouTube.

Tiroir-caisse C'est la partie de la caisse qui s'ouvre, dans laquelle on peut placer un plateau-billets.

Plateau-billets C'est un contenant pour les billets de banque et les pièces de monnaie qui facilite le changement de personnel à une caisse. Chaque caissier ou caissière possède un plateau-billets et l'apporte lorsqu'il est au début ou à la fin de son quart de travail.

C.2. Cas d'utilisation : Ouvrir la caisse

Acteur principal : Caissier ou Caissière

Préconditions : La caisse est libre et son tiroir-caisse est vide (il n'y a pas de plateau dedans).

Garanties de succès (postconditions) : La caissière ou le caissier est authentifié. Son plateau-billets est placé dans le tiroir-caisse, et son identificateur est enregistré. Le montant d'argent du plateau est enregistré. L'heure de l'arrivée du caissier ou de la caissière est enregistrée.

Scénario principal (succès)

1. Le Caissier ou la caissière arrive à la caisse avec son plateau-billets.
2. Le Caissier ou la caissière saisit son identifiant et son mot de passe dans la boîte de dialogue d'authentification.

3. Le Système authentifie le Caissier ou la caissière.
4. Le Système ouvre le tiroir-caisse et demande au Caissier ou à la caissière de poser son plateau dans le tiroir-caisse.
5. Le Caissier ou la caissière pose son plateau dans le tiroir-caisse.
6. Le Système reconnaît l'identificateur du plateau.
7. Le Système demande au Caissier ou à la caissière de saisir le montant d'argent du plateau.
8. Le Caissier ou la caissière saisit le montant d'argent du plateau.
9. Le Système demande au Caissier ou à la caissière de fermer le tiroir-caisse.
10. Le Caissier ou la caissière ferme le tiroir-caisse.

Spécifications particulières :

Les caisses sont configurées avec un modèle de plateau-billets comme celui-ci : ([voir exemple sur YouTube](#).)

Fréquence d'occurrence : Normalement, au début du quart de travail de chaque caissier ou caissière.

C.3. Modèle du domaine partiel

Voici un exemple pour le système POS NextGen. Notez les nouvelles classes conceptuelles **Plateau-Billets** (un objet physique) et **MisePlateau** (une transaction) faisant partie du scénario d'*Ouvrir la caisse*.

On remarque que lorsqu'un Registre (Caisse) n'a pas de Caissier (l'état du système au début du cas d'utilisation), l'objet Registre n'est associé à aucun objet Caissier. Nous avons donc modifié les cardinalités de l'association en conséquence. C'est une différence par rapport au modèle du domaine de base présenté par Larman, qui n'avait pas considéré ce cas d'utilisation, bien qu'il soit dans l'ensemble des spécifications du système.

Les associations en rouge sont celles qui sont affectées par la dynamique de ce cas d'utilisation.

C. Cas d'utilisation - Ouvrir la caisse

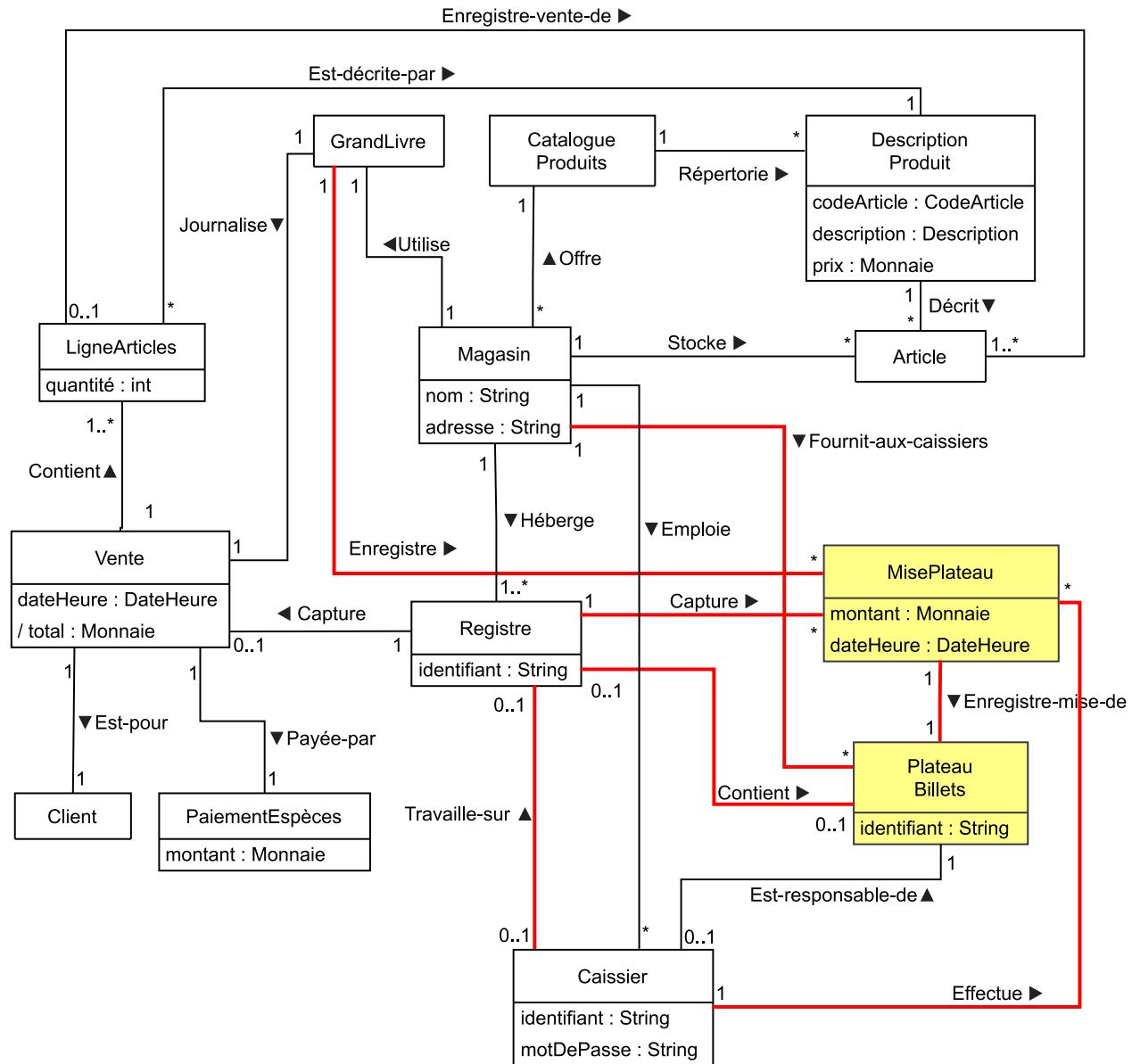


FIGURE C.3. – Modèle du domaine partiel du système POS NextGen avec deux nouvelles classes conceptuelles MisePlateau et PlateauBillets, qui modélisent des éléments du cas d'utilisation « Ouvrir la caisse ».

C.4. Diagramme de séquence système (DSS)

La figure C.4 est le diagramme de séquence système (DSS) pour le scénario « Ouvrir la caisse ».

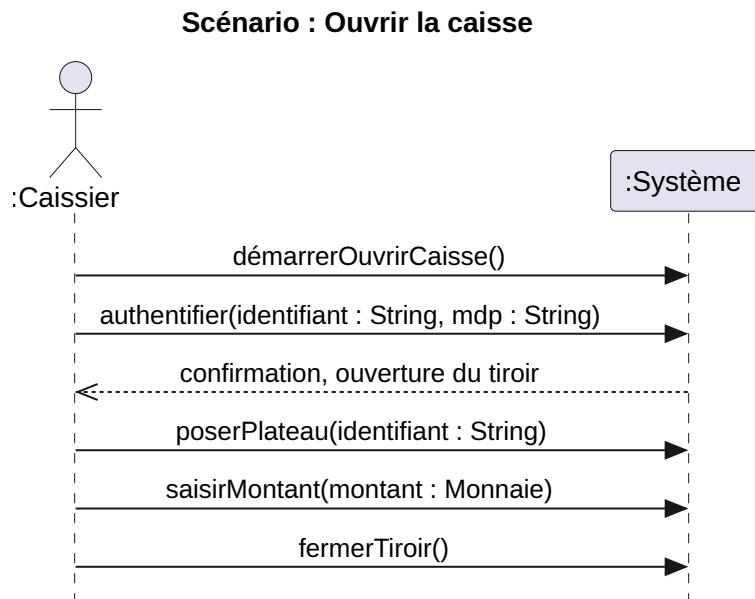


FIGURE C.4. – Diagramme de séquence système (DSS) pour le scénario « Ouvrir la caisse ».

C. Cas d'utilisation - Ouvrir la caisse

C.5. Contrats d'opération

Voici les contrats pour chaque opération système.

Opération : démarrerOuvrirCaisse()

Postconditions :

- Une instance *mp* de MisePlateau a été créée.

Opération : authentifier(identifiant : String, mdp : String)

Postconditions :

- *mp* a été associée à un Caissier, sur la base de correspondance avec **identifiant**.
- Le Registre en cours a été associé à un Caissier, sur la base de correspondance avec **identifiant**.

Opération : poserPlateau(identifiant : String)

Postconditions :

- *mp* a été associée à un PlateauBillets, sur la base de correspondance avec **identifiant**.
- Le Registre en cours a été associé à un PlateauBillets, sur la base de correspondance avec **identifiant**.

Opération : saisirMontant(montant : Monnaie)

Postcondition :

- *mp.montant* est devenu **montant**.

Opération : fermerTiroir()

Postconditions :

- *mp.dateHeure* est devenue la date et l'heure actuelles.
- *mp* a été associée à GrandLivre.
- *mp* a été associée à Registre.