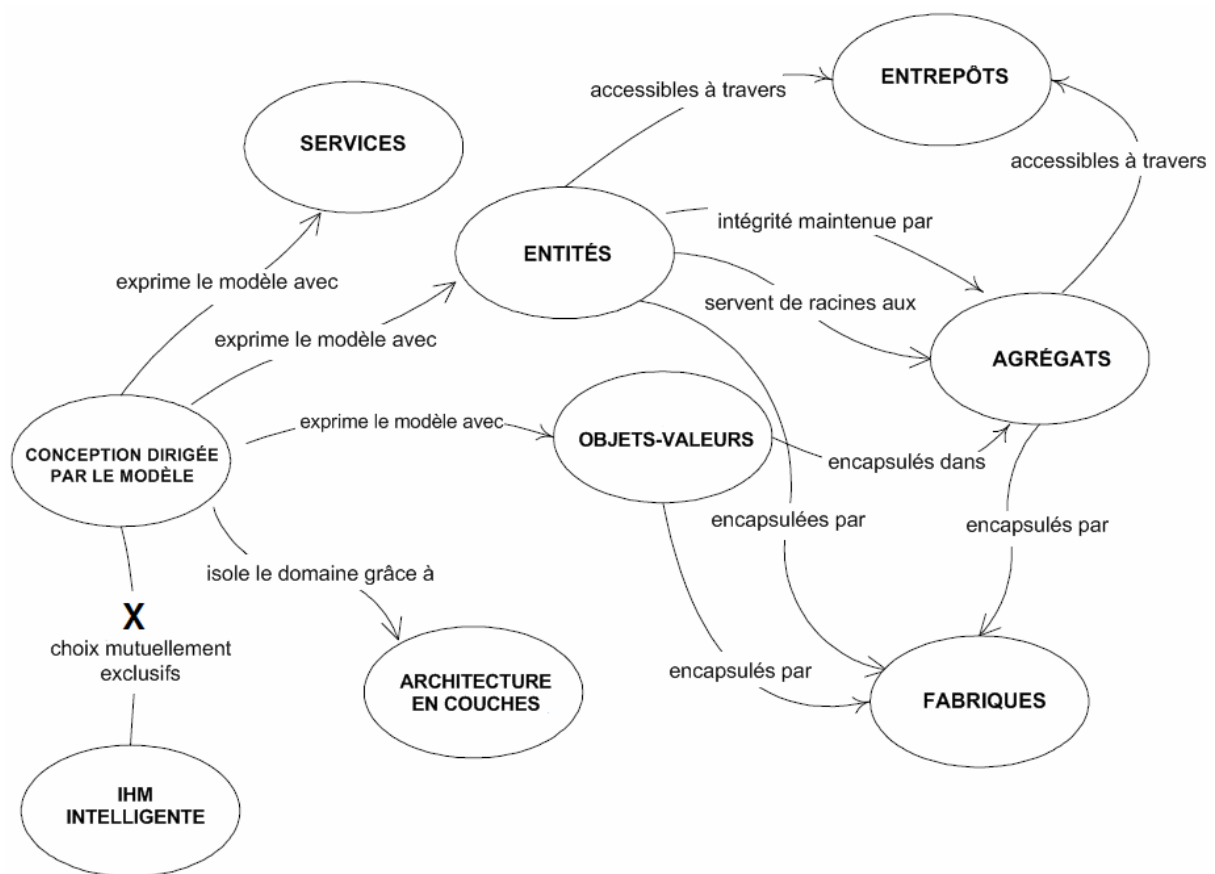


Domain-Driven Design *Vite fait*



par Abel Avram & Floyd Marinescu

édité par : Dan Bergh Johnson, Vladimir Gitlevich

traduction : Guillaume Lebur

Domain-Driven Design

Vite fait

Tous droits réservés.

C4Media, Editeur de InfoQ.com.

Ce livre fait partie de la collection de livres InfoQ "Enterprise Software Development".

Pour plus d'informations ou pour commander ce livre ou d'autres livres InfoQ, prière de contacter books@c4media.com.

Aucune partie de cette publication ne peut être reproduite, stockée dans un système de recherche, ou transmise sous aucune forme ni aucun moyen, électronique, mécanique, photocopie, recodage, scanner ou autre sans que cela ne soit autorisé par les Sections 107 ou 108 du Copyright Act 1976 des Etats-Unis, ni sans l'autorisation écrite préalable de l'éditeur.

Les termes utilisés par les entreprises pour distinguer leurs produits sont souvent déclarés comme des marques commerciales. Dans tous les cas où C4Media Inc. est informée d'une telle déclaration, les noms de produits apparaissent avec une Majuscule initiale ou EN TOUTES LETTRES MAJUSCULES.

Toutefois, les lecteurs devraient contacter les entreprises appropriées pour des informations plus complètes au sujet des marques commerciales et de leur enregistrement.

Certains diagrammes utilisés dans ce livre ont été reproduits, avec autorisation, sous la licence Creative Commons, courtesy of : Eric Evans, DOMAIN-DRIVEN DESIGN, Addison-Wesley, © Eric Evans, 2004.

Crédits de production :
Résumé DDD par : Abel Avram
Responsable éditorial : Floyd Marinescu
Couverture : Gene Steffanson
Composition: Laura Brown et Melissa Tessier
Traduction : [Guillaume Lebur](#)
Remerciements spéciaux à Eric Evans.

Library of Congress Cataloging-in-Publication Data:

ISBN: 978-1-4116-0925-9

Imprimé aux Etats-Unis d'Amérique

10 9 8 7 6 5 3 2 1

Table des matières

Préface : Pourquoi DDD Vite fait ?	6
Introduction	8
1 Qu'est-ce que Domain Driven Design ?	10
Bâtir la connaissance du domaine	14
2 Le Langage omniprésent	18
Le besoin d'un langage commun	18
Créer le Langage omniprésent.....	20
3 Conception dirigée par le Modèle	26
Les blocs de construction d'une conception orientée Modèle.....	30
L'architecture en couches	30
Les Entités.....	33
Les Objets-Valeurs	35
Les Services	37
Les Modules	40
Les Agrégats.....	41
Les Fabriques	45
Les Entrepôts.....	49
4 Refactorer pour une vision plus profonde	55
Refactorer en continu.....	55
Mettre au jour les concepts clés.....	57
5 Préserver l'intégrité du modèle	62
Contexte borné	63
Intégration continue	66
Carte de Contexte.....	67
Noyau partagé	68
Client-Fournisseur	69

Conformiste	71
Couche anticorruption.....	73
Chemins séparés.....	75
Service Hôte ouvert	76
Distillation	77
6 DDD compte aujourd’hui : une interview d’Eric Evans	82
Lexique français-anglais des termes DDD.....	88

Préface : Pourquoi DDD Vite fait ?

J'ai entendu parler de Domain Driven Design et j'ai rencontré Eric Evans pour la première fois dans une petite assemblée d'architectes lors d'une conférence organisée à la montagne par Bruce Eckel à l'été 2005. Un certain nombre de gens que je respecte participaient à la conférence, dont Martin Fowler, Rod Johnson, Cameron Purdy, Randy Stafford et Gregor Hohpe.

Le groupe semblait assez impressionné par la vision Domain Driven Design, et impatient d'en apprendre plus. J'ai aussi senti que tout le monde aurait aimé que ces concepts rentrent davantage dans les moeurs. Quand j'ai constaté qu'Eric utilisait le modèle du domaine pour proposer des solutions à divers défis techniques dont le groupe discutait, et mesuré l'importance qu'il accordait au domaine métier plutôt qu'à la hype autour d'une technologie en particulier, j'ai tout de suite su que cette vision était de celles qui manquaient cruellement à la communauté.

Nous, les membres de la communauté des développeurs d'applications d'entreprise, et particulièrement la communauté des développeurs web, avons été pollués par des années d'une hype qui nous a éloignés du développement orienté objet en lui-même. Dans la communauté Java, la notion de bon modèle de domaine s'est perdue, noyée dans la frénésie autour des EJB et des modèles container/component des années 1999-2004. Par chance, des changements technologiques et les expériences collectives de la communauté du développement logiciel sont en train de nous ramener vers les paradigmes orienté objet traditionnels. Cependant, il manque à la communauté une vision claire de la manière d'appliquer l'orienté objet à l'échelle de l'Entreprise, raison pour laquelle je pense que DDD est important.

Malheureusement, hormis un petit groupe d'architectes parmi les plus vétérans, j'avais la perception que très peu de gens étaient au courant de DDD, et c'est pour cela qu'InfoQ a commandé l'écriture de ce livre.

J'ai bon espoir qu'en publiant un résumé, une introduction aux fondamentaux de DDD courte et rapide à lire et en la rendant librement téléchargeable sur InfoQ avec à côté une version papier de poche et bon marché, cette vision puisse devenir parfaitement répandue.

Ce livre n'introduit **aucun concept nouveau** ; il tente de résumer de façon concise l'essence de DDD, puisant principalement dans le livre original d'Eric Evans sur le sujet, ainsi que dans d'autres sources publiées depuis comme l' *Applying DDD* de Jimmy Nilsson et divers forums de discussion sur DDD. Ce livre vous donnera un cours accéléré sur les fondamentaux de DDD, mais il ne se substitue pas aux

nombreux exemples et études de cas fournies dans le livre d'Eric ou aux exemples de terrain proposés dans le livre de Jimmy. Je vous encourage vivement à lire l'un et l'autre de ces deux excellents ouvrages. En attendant, si vous pensez que la communauté a besoin que DDD fasse partie de notre conscience collective, n'hésitez pas à parler autour de vous de ce livre et du travail d'Eric.

Floyd Marinescu

Co-fondateur et Chief Editor d'InfoQ.com

Introduction

Les logiciels sont des instruments créés pour nous aider à traiter la complexité de la vie moderne. Les logiciels sont juste un moyen pour nous d'atteindre un but, et généralement ce but est quelque chose de tout à fait pratique et réel. Par exemple nous utilisons des logiciels pour le contrôle du trafic aérien, ce qui est directement relié au monde qui nous entoure. Nous voulons aller d'un endroit à un autre par la voie des airs, et pour ça nous utilisons une machinerie sophistiquée, alors nous créons des logiciels pour coordonner les vols des milliers d'avions qui se trouvent dans le ciel à toute heure.

Les logiciels doivent être pratiques et utiles ; autrement nous n'investirions pas autant de temps et de ressources dans leur création. Cela les rend éminemment reliés à un certain aspect de nos vies. Un packaging logiciel utile ne peut pas être décorrélié de cette sphère de réalité : le domaine qu'il est censé nous aider à gérer. Au contraire, le logiciel y est profondément entremêlé.

La conception de logiciels est un art, et comme tout art elle ne peut pas être enseignée et apprise comme une science précise, au moyen de théorèmes et de formules. Nous pouvons découvrir des principes et des techniques utiles à appliquer tout au long du processus de création de logiciel, mais nous ne serons probablement jamais capables de fournir un chemin exact à suivre en partant du besoin du monde réel pour arriver jusqu'au module de code destiné à répondre à ce besoin. Comme une photo ou un bâtiment, un produit logiciel contiendra toujours la touche personnelle de ceux qui l'ont conçu et développé, un petit quelque chose du charisme et du flair (ou du manque de flair) de ceux qui ont contribué à sa naissance et à sa croissance.

Il y a différentes manières d'aborder la conception logicielle. Ces 20 dernières années, l'industrie du logiciel a connu et utilisé plusieurs méthodes pour créer ses produits, chacune avec ses avantages et ses inconvénients. L'objectif de ce livre est de se focaliser sur une méthode de conception qui a émergé et évolué au cours des deux dernières décennies, mais s'est plus distinctement cristallisée depuis quelques années : la conception dirigée par le domaine. Eric Evans a grandement contribué à la question en couchant sur le papier dans un unique livre une grande part de la connaissance accumulée sur la conception dirigée par le domaine. Pour une présentation plus détaillée du sujet, nous vous recommandons la lecture de son livre « Domain-Driven Design: Tackling Complexity in the Heart of Software » publié par Addison-Wesley, ISBN: 0-321-12521-5.

Vous pourrez aussi découvrir de nombreux et précieux points de vue en suivant le groupe de discussion Domain Driven Design à l'adresse :

<http://groups.yahoo.com/group/domaindrivendesign>

Ce livre est seulement une introduction au sujet, destinée à vous donner rapidement une compréhension fondamentale, mais non détaillée, de Domain Driven Design.

Nous voulons simplement aiguïser votre appétit pour les bonnes pratiques de conception logicielle grâce aux principes et lignes directrices utilisées dans le monde de la conception dirigée par le domaine.

1

Qu'est-ce que Domain Driven Design ?

Le développement logiciel est le plus souvent employé pour automatiser des processus qui existent déjà dans le monde réel, ou pour fournir des solutions à des problèmes métier réels. Ces processus métier à automatiser ou ces problèmes du monde réel que le logiciel traite constituent le domaine du logiciel. Nous devons comprendre dès le départ que le logiciel trouve son origine dans ce domaine et lui est profondément lié.

Les logiciels sont faits de code. Nous pourrions être tentés de passer trop de temps sur le code, et voir les logiciels simplement comme des objets et des méthodes.

Pensez à la métaphore de la fabrication de voitures. Les travailleurs impliqués dans la construction automobile peuvent se spécialiser dans la production de pièces de voiture, mais ce faisant ils ont souvent une vision limitée du processus de fabrication automobile dans son ensemble. Ils commencent à voir la voiture comme un énorme assemblage de pièces qui doivent aller les unes avec les autres, mais une voiture est bien plus que cela. Une bonne voiture, ça commence par une vision. Cela commence par des spécifications écrites avec soin. Et ça se poursuit avec de la conception. Beaucoup, beaucoup de conception. Des mois, peut-être des années passées sur la conception, à la modifier et à la raffiner jusqu'à ce qu'elle atteigne la perfection, jusqu'à ce qu'elle reflète la vision originale. Le processus de conception ne se fait pas entièrement sur papier. Une grande partie consiste à fabriquer des modèles de la voiture, et à les tester sous certaines conditions pour vérifier qu'ils marchent. On modifie la conception sur la base des résultats de ces tests. Finalement, la voiture est envoyée en production, et les pièces sont créées et assemblées.

Le développement logiciel est similaire. On ne peut pas se contenter de rester assis et taper du code. On peut le faire, et ça marche bien pour des cas triviaux. Mais on ne peut pas créer des logiciels complexes de cette manière.

Pour créer un bon logiciel, vous devez savoir de quoi il y est question. Vous ne pouvez pas créer un système logiciel bancaire à moins d'avoir une bonne compréhension de ce qu'est la banque, vous devez comprendre le *domaine* de la banque.

Est-il possible de créer un logiciel bancaire complexe sans une bonne connaissance du domaine ? Pas une seconde. Jamais. Qui s'y connaît en banque ? L'architecte logiciel ? Non. Il utilise juste une banque pour s'assurer que son argent est en sécurité et disponible quand il en a besoin. L'analyste logiciel ? Pas vraiment. Il sait analyser un sujet précis, quand on lui donne tous les éléments nécessaires. Le développeur ? Vous pouvez tout de suite oublier. Qui, alors ? Les banquiers, bien sûr. Le système bancaire est très bien compris par les gens de l'intérieur, par ses spécialistes. Ils en connaissent tous les détails, tous les pièges, tous les problèmes possibles, toutes les règles. Voilà par où l'on devrait toujours commencer : le domaine.

Lorsque nous débutons un projet informatique, nous devrions nous concentrer sur le domaine dans lequel il opère. Le seul but du logiciel est d'améliorer un domaine spécifique. Pour être capable de faire cela, le logiciel doit se mettre au diapason du domaine pour lequel il a été créé. Sans cela, il introduira des tensions dans le domaine, provoquant des dysfonctionnements, des dégâts, voire même semant le chaos.

Comment peut-on mettre le logiciel au diapason du domaine ? La meilleure façon d'y arriver est de faire du logiciel un reflet du domaine. Le logiciel doit incorporer les concepts et éléments qui sont au cœur du domaine, et saisir avec précision les relations entre eux. Le logiciel doit modéliser le domaine.

Quelqu'un qui n'a pas de connaissances bancaires doit pouvoir en apprendre beaucoup en lisant simplement le code du modèle du domaine. C'est essentiel. Un logiciel dont les racines ne sont pas enfouies profondément dans le domaine ne réagira pas bien au changement au fil du temps.

On commence donc par le domaine. Quoi d'autre ensuite ? Un domaine est une chose qui appartient à ce bas-monde. On ne peut pas juste le prendre et le verser sur le clavier afin qu'il rentre dans l'ordinateur et devienne du code. Il nous faut créer une abstraction du domaine. On en apprend beaucoup sur un domaine quand on parle avec les experts du domaine. Mais cette connaissance brute ne sera pas facilement transformable en des constructions logicielles, à moins que nous en fabriquions une abstraction, un schéma dans notre esprit. Au début, le schéma est toujours incomplet. Mais avec le temps, en travaillant dessus, nous l'améliorons et il devient de plus en plus clair pour nous. Qu'est-ce que cette abstraction ? C'est un modèle, un modèle du domaine. Selon Eric Evans, un modèle du domaine n'est pas un diagramme particulier ; c'est l'idée qu'on cherche à véhiculer à travers le diagramme. Ce n'est pas simplement la connaissance contenue dans le cerveau d'un expert du domaine ; c'est une abstraction rigoureusement organisée et sélective de cette connaissance. Un diagramme peut représenter et communiquer un modèle, comme peut le faire un code soigneusement écrit, comme peut le faire aussi une phrase en français.

Le modèle est notre représentation interne du domaine ciblé, et elle est absolument nécessaire tout au long de la conception comme du processus de développement. Pendant la conception, nous nous remémorons le modèle et nous nous y référons de nombreuses fois. Le monde qui nous entoure représente bien plus que ce que nos cerveaux peuvent traiter. Même un domaine spécifique pourrait bien être plus que ce qu'un esprit humain peut manier à la fois. Nous devons organiser l'information, la systématiser, la diviser en plus petits morceaux, regrouper ces morceaux dans des modules logiques, en prendre un à la fois et le traiter. Il nous faut même laisser de côté certaines parties du domaine. Un domaine contient simplement une masse trop grande d'information pour l'inclure toute entière dans le modèle. Et pour une grande partie, il n'est même pas nécessaire de la prendre en compte. C'est un défi en soi. Que garder et quoi jeter ? Ca fait partie de la conception, du processus de création logicielle. Le logiciel bancaire gardera sûrement une trace de l'adresse du client, mais il ne devrait pas s'occuper de la couleur de ses yeux. C'est un exemple évident, mais il peut y en avoir d'autres moins flagrants.

Un modèle est une partie essentielle d'une conception logicielle. Nous en avons besoin pour pouvoir gérer la complexité. Toute notre mécanique de raisonnement sur le domaine est synthétisée dans ce modèle. Tout ça est très bien, mais cela doit sortir de notre esprit. Ce n'est pas très utile si ça y reste enfermé, vous ne croyez pas ? Nous devons communiquer ce modèle aux experts du domaine, à nos collègues concepteurs, et aux développeurs. Le modèle est l'essence du logiciel, mais nous devons inventer des façons de l'exprimer, de le transmettre aux autres. Nous ne sommes pas tout seuls dans ce processus, nous devons donc partager la connaissance et l'information, et nous devons le faire bien, précisément, complètement et sans ambiguïté. Il y a différentes manières de le faire. L'une est graphique : diagrammes, cas d'utilisation, dessins, photos, etc. L'autre est de l'écrire. De coucher sur le papier notre vision du domaine. Une autre encore est le langage. Nous pouvons et nous devrions créer un langage pour communiquer des problèmes spécifiques du domaine. Nous détaillerons tous ces moyens plus tard, mais l'idée principale est que *nous devons communiquer le modèle*.

Lorsque notre modèle est exprimé, on peut commencer à concevoir le code. Ce qui est différent de concevoir le logiciel. Concevoir le logiciel, c'est comme imaginer l'architecture d'une maison, c'est porter un œil global sur l'édifice. A l'inverse, concevoir le code c'est travailler les détails, comme l'emplacement d'un tableau sur tel ou tel mur. Le design du code est tout aussi important, mais pas aussi fondamental que la conception du logiciel. Une erreur dans le design du code se corrige d'habitude facilement, tandis que les erreurs de conception du logiciel sont beaucoup plus coûteuses à réparer. C'est une chose de déplacer un tableau un peu plus à gauche, mais c'en est une toute autre de démolir un côté de la maison pour le refaire différemment. Néanmoins, le produit final ne sera pas bon sans un bon design du code. C'est là que les design patterns de code s'avèrent pratiques, et on devrait toujours les appliquer

quand c'est nécessaire. De bonnes techniques de programmation aident à créer du code propre et maintenable.

Il y a différentes approches de la conception logicielle. Une d'entre elles est la méthode de conception en cascade. Cette méthode implique un certain nombre d'étapes. Les experts métiers mettent en place un ensemble de spécifications qui sont communiquées aux analystes métier. Les analystes créent un modèle basé sur ces spécifications, et font passer le résultat aux développeurs, qui se mettent à coder en se basant sur ce qu'ils ont reçu. C'est un flux de connaissance à sens unique. Même si cette approche a été une approche traditionnelle en conception de logiciels et qu'on l'a utilisée avec un certain degré de succès au fil des années, elle a ses failles et ses limites. Le principal problème est qu'il n'y a pas de feedback des analystes aux experts métier ni des développeurs aux analystes.

Une autre approche est celle des Méthodes Agiles, comme eXtreme Programming (XP). Ces méthodologies sont un mouvement collectif contre l'approche en cascade, résultant de la difficulté d'essayer d'obtenir toutes les spécifications dès le début, particulièrement à la lumière du fait que les exigences peuvent changer. C'est vraiment difficile de créer un modèle complet qui couvre d'avance tous les aspects d'un domaine. Ça demande beaucoup de réflexion, souvent on ne voit pas tous les enjeux dès le début, et on ne peut pas prévoir non plus certains effets de bord négatifs ou erreurs dans notre conception. Un autre problème que l'agilité essaie de résoudre est ce qu'on appelle l'« *analysis paralysis* », situation où les membres de l'équipe ont tellement peur de prendre des décisions de conception qu'il ne progressent pas du tout. Même si les apôtres de l'agilité reconnaissent l'importance des décisions de conception, ils s'opposent à l'idée de tout concevoir dès le début. Au lieu de ça, ils font un grand usage de la flexibilité dans l'implémentation, et à travers un développement itératif, une participation continue du décideur métier et beaucoup de refactoring, l'équipe de développement parvient à en apprendre plus sur le domaine du client et peut mieux produire du logiciel qui satisfait les besoins du client.

Les méthodes Agiles ont leurs propres problèmes et leurs limites ; elles prônent la simplicité, mais chacun a sa propre vision de ce que cela veut dire. De plus, un refactoring fait en continu par des développeurs sans de solides principes de design produira du code difficile à comprendre ou à modifier. Et s'il est vrai que l'approche en cascade peut mener à une surabondance d'ingénierie, la crainte de l'excès d'ingénierie peut quant à elle conduire à une autre peur : celle de mener une conception en profondeur et mûrement réfléchie.

Ce livre présente les principes de Domain-Driven Design qui, quand ils sont appliqués, peuvent grandement augmenter la capacité de n'importe quel processus de développement à modéliser et implémenter les problématiques complexes d'un

domaine de façon maintenable. Domain-Driven Design combine pratiques de conception et de développement, et montre comment la conception et le développement peuvent travailler de concert pour créer une meilleure solution. Une bonne conception va améliorer le développement, de même que le feedback provenant du processus de développement va améliorer le design.

Bâtir la connaissance du domaine

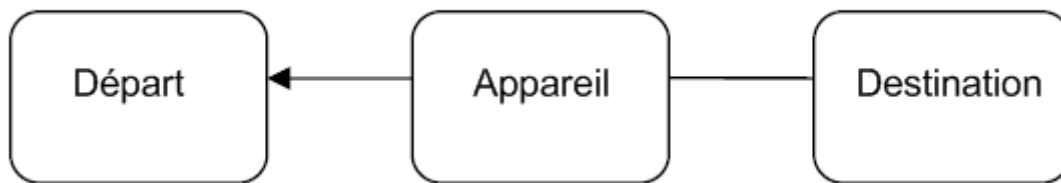
Considérons l'exemple d'un projet de système de surveillance de vols aéronautiques, et voyons comment la connaissance du domaine peut être bâtie.

A un instant donné, des milliers d'avions sont en l'air dans tous les cieux de la planète. Ils cheminent chacun vers leur destination, et il est assez important de s'assurer qu'ils ne se heurtent pas en l'air. Nous n'essaierons pas de travailler sur le système de contrôle du trafic tout entier, mais sur un plus petit sous-ensemble : un système de suivi de vol. Le projet proposé est un système de surveillance qui repère chaque vol dans une zone donnée, détermine si le vol suit sa route prévue ou non, et s'il y a possibilité de collision.

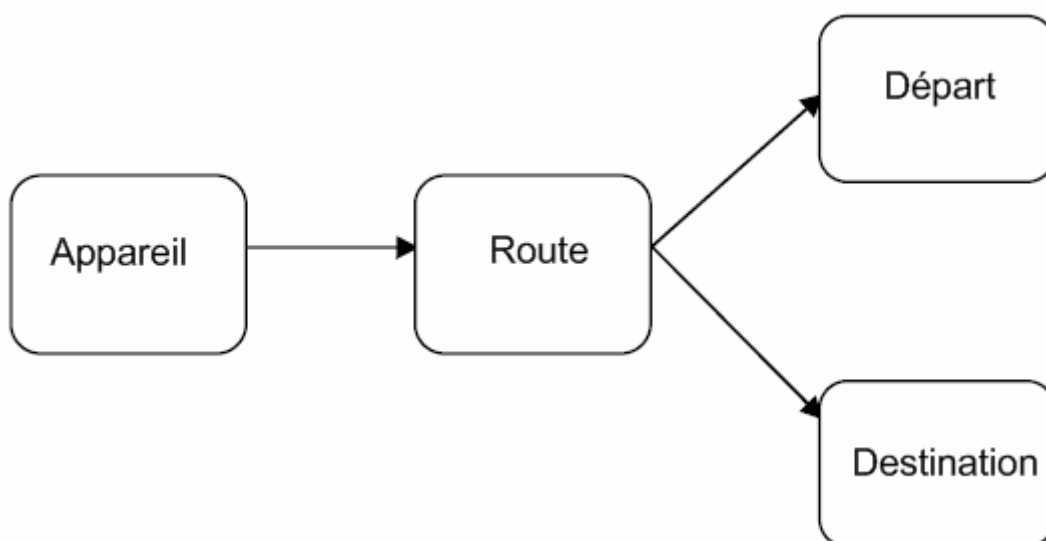
Par où allons-nous commencer, d'un point de vue développement ? Dans la section précédente, nous avons dit que nous devrions commencer par comprendre le domaine, qui dans ce cas est le suivi du trafic aérien. Les aiguilleurs du ciel sont les spécialistes de ce domaine. Mais les aiguilleurs ne sont pas des concepteurs système ni des spécialistes en informatique. Vous ne pouvez pas attendre d'eux qu'ils vous tendent une description complète du domaine qui pose problème.

Les aiguilleurs du ciel ont une vaste connaissance de leur domaine, mais pour pouvoir bâtir un domaine, vous devez extraire l'information essentielle et la généraliser. Quand vous allez commencer à leur parler, vous en entendrez beaucoup sur les décollages d'avions, les atterrissages, les appareils en vol et le risque de collision, les avions qui attendent la permission d'atterrir, etc. Pour trouver un ordre dans cette quantité d'information apparemment chaotique, nous devons bien commencer quelque part.

L'aiguilleur et vous convenez que chaque appareil a un terrain d'aviation de départ et d'arrivée. Nous avons donc un appareil, un départ et une destination, comme indiqué dans la figure ci-dessous.

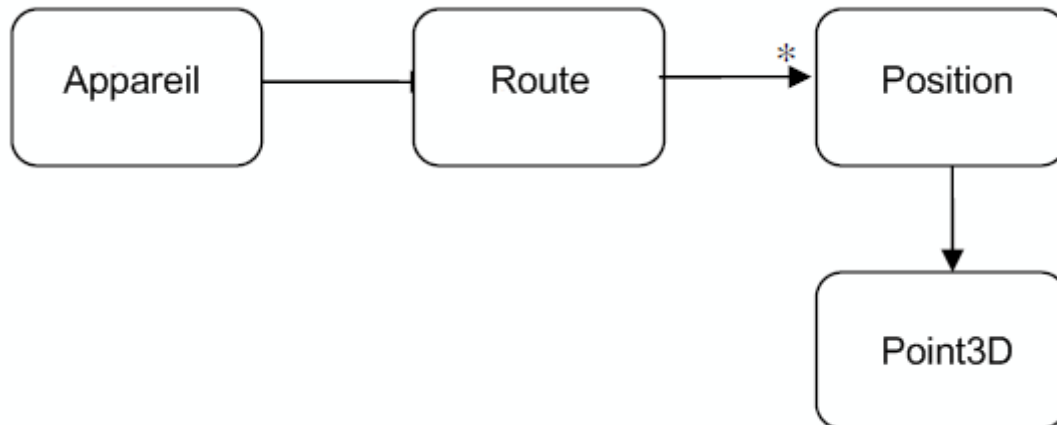


OK, l'avion décolle d'un endroit et touche le sol dans un autre. Mais que se passe-t-il en l'air ? Quel chemin de vol suit-il ? En vérité, nous nous intéressons davantage à ce qui se passe une fois qu'il est en l'air. L'aiguilleur dit qu'on assigne à chaque avion un plan de vol qui est censé décrire le voyage aérien en entier. Lorsque vous entendez parler de plan de vol, vous pouvez penser dans votre for intérieur qu'il s'agit du chemin suivi par l'avion dans le ciel. En discutant plus avant, vous notez un mot intéressant : route. Ce mot capte votre attention immédiatement, et ce pour une bonne raison. La route représente un concept important en matière de transport aérien. C'est ce que les avions font en volant, ils suivent une route. Il est évident que le départ et la destination de l'aéronef sont aussi les points de début et de fin de la route. Donc au lieu d'associer l'appareil aux points de départ et de destination, il semble plus naturel de le relier à une route, qui à son tour est associée au départ et à la destination correspondants.

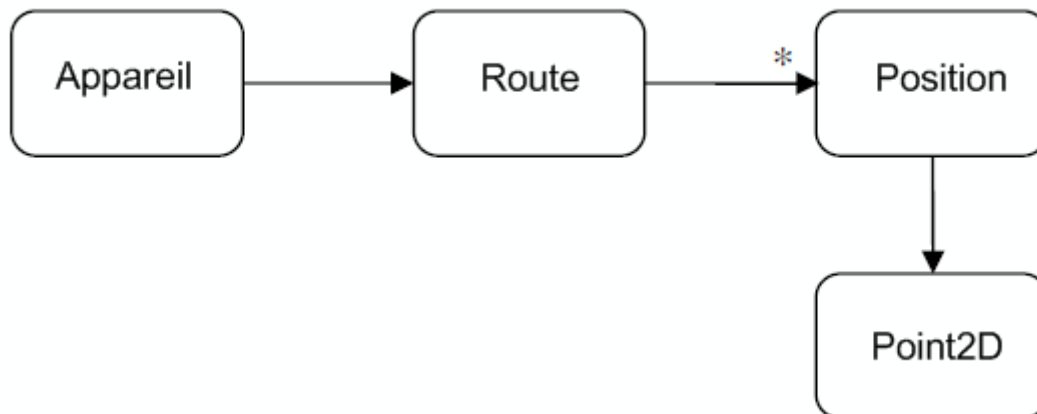


En parlant avec l'aiguilleur des routes suivies par les avions, vous découvrez que la route est en fait formée de petits segments qui, mis bout à bout, constituent une sorte de ligne tortueuse allant du départ à la destination. La ligne est censée passer par des points fixes déterminés à l'avance. Donc, une route peut être considérée comme une série de positions consécutives. A ce niveau, vous ne voyez plus le départ et la destination comme les points terminaux de la route, mais juste comme deux positions parmi d'autres. C'est probablement assez différent de la façon dont les voit

l'aiguilleur, mais il s'agit d'une abstraction nécessaire qui sera utile plus tard. Les changements résultant de ces découvertes sont les suivants :



Le diagramme montre un autre élément : le fait que chaque position est un point de l'espace suivi par la route, et elle est exprimée sous forme d'un point en 3 dimensions. Mais en parlant à l'aiguilleur, vous allez découvrir qu'il ne le voit pas de la même manière. En fait, il voit la route comme une projection sur Terre du vol de l'avion. Les positions sont juste des points à la surface de la Terre déterminés de manière unique par leur latitude et leur longitude. Le diagramme correct est donc :



Que se passe-t-il ici en réalité ? Vous et les experts du domaine, vous êtes en train de parler, d'échanger de la connaissance. Vous commencez à poser des questions, et ils vous répondent. Ce faisant, ils font ressortir des concepts essentiels du domaine du trafic aérien. Il est possible que ces concepts sortent mal dégrossis et désorganisés, mais ils sont néanmoins essentiels pour comprendre le domaine. Il vous faut en apprendre le plus possible sur le domaine de la part des experts. Et en glissant les bonnes questions, en traitant l'information de la bonne manière, vous et les experts allez commencer à esquisser une vue du domaine, un modèle du domaine. Cette vue

n'est ni complète ni juste, mais c'est le point de départ dont vous avez besoin. Essayez de comprendre les concepts essentiels du domaine.

C'est une partie importante du design. D'habitude, de longues discussions ont lieu entre les architectes logiciels ou les développeurs et les experts du domaine. Les spécialistes logiciels veulent extraire la connaissance des experts du domaine, et ils doivent aussi la transformer en une forme utile. Arrivés à un certain point, ils pourraient avoir envie de créer un premier prototype pour voir comment ça marche à ce moment-là. Ce faisant, il est possible qu'ils rencontrent certains problèmes dans leur modèle ou leur approche, et qu'ils veuillent changer le modèle. La communication ne se fait pas seulement dans un sens, des experts du domaine vers l'architecte logiciel puis vers les développeurs. Il y a aussi des retours, qui aident à créer un meilleur modèle ainsi qu'une compréhension plus claire et plus juste du domaine. Les experts du domaine connaissent bien leur terrain d'expertise, mais ils organisent et utilisent leur connaissance d'une manière spécifique, qui n'est pas toujours la bonne pour une implémentation dans un système logiciel. L'esprit d'analyse du concepteur du logiciel aide à dénicher certains concepts-clés du domaine lors des discussions avec les experts, et aide aussi à construire une structure pour de futures discussions comme nous le verrons au chapitre suivant. Nous, les spécialistes en logiciel (architectes logiciels et développeurs), et les experts du domaine, créons le modèle du domaine ensemble, et le modèle est l'endroit où ces deux champs d'expertise se rencontrent. Ce processus peut sembler très gourmand en temps, et c'est le cas, mais c'est comme ça qu'il doit être, parce qu'au final le but du logiciel est de résoudre des problèmes métier dans un domaine de la vie réelle, et il doit donc se fondre parfaitement dans le domaine.

2

Le Langage omniprésent

Le besoin d'un langage commun

Le chapitre précédent a fait la démonstration qu'il était absolument nécessaire de développer un modèle du domaine en faisant travailler les spécialistes informatique avec les experts du domaine ; toutefois cette approche se heurte couramment à quelques difficultés dues à une barrière de communication initiale. Les développeurs ont la tête pleine de classes, de méthodes, d'algorithmes, de patterns, et ont tendance à toujours vouloir faire correspondre un concept de la vie réelle à un artefact de programmation. Ils veulent voir les classes d'objets qu'ils devront créer et les relations entre elles qu'il faudra modéliser. Ils pensent en termes d'héritage, de polymorphisme, d'orienté objet, etc. Et ils parlent comme ça tout le temps. Et c'est normal pour eux d'agir ainsi. Les développeurs restent des développeurs. Mais les experts du domaine ne connaissent en général rien de tout cela. Ils n'ont aucune idée de ce que peuvent être des bibliothèques logicielles, des frameworks, la persistance, et dans bien des cas même une base de données. Ils connaissent leur champ d'expertise particulier.

Dans l'exemple de surveillance du trafic aérien, les experts du domaine s'y connaissent en avions, en routes, en altitudes, longitudes et latitudes, ils maîtrisent les déviations par rapport à l'itinéraire normal, les trajectoires des avions. Et ils parlent de ces choses dans leur propre jargon, ce qui parfois n'est pas très facile à suivre pour quelqu'un de l'extérieur.

Pour surmonter la différence de style de communication, quand nous construisons le modèle, nous devons nous parler pour échanger des idées sur le modèle, sur les éléments impliqués dans le modèle, la manière dont nous les relierons, sur ce qui est pertinent ou non. La communication à ce niveau est primordiale pour la réussite du projet. Si quelqu'un dit quelque chose et que l'autre ne comprend pas, ou pire encore, comprend autre chose, quelles chances y-a-t-il pour que le projet aboutisse ?

Un projet se trouve face à de sérieux problèmes lorsque les membres de l'équipe ne partagent pas un langage commun pour discuter du domaine. Les experts du domaine utilisent leur jargon tandis que les membres de l'équipe technique ont leur propre langage prévu pour discuter du domaine en termes de conception.

La terminologie des discussions de tous les jours est déconnectée de la terminologie embarquée dans le code (qui est finalement le produit le plus important d'un projet informatique). Et même un individu identique utilise un langage différent à l'oral et à l'écrit, si bien que les expressions du domaine les plus perspicaces émergent souvent dans une forme transitoire qui n'est jamais capturée dans le code ni même dans les documents écrits.

Lors de ces séances de discussion, on utilise souvent la traduction pour faire comprendre aux autres le sens de certains concepts. Les développeurs peuvent essayer d'expliquer quelques design patterns en utilisant un langage de non-spécialiste, parfois sans succès. Les experts du domaine vont s'efforcer de faire passer certaines de leurs idées en créant probablement un nouveau jargon. Au cours de ce processus, la communication souffre, et ce genre de traduction ne favorise pas le mécanisme de construction de la connaissance.

Nous avons tendance à utiliser nos propres dialectes pendant ces séances de conception, mais aucun de ces dialectes ne peut être un langage commun parce qu'aucun ne sert les besoins de tout le monde.

Nous avons résolument besoin de parler le même langage quand nous nous réunissons pour parler du modèle et le définir. Quel langage cela va-t-il être ? Celui des développeurs ? Celui des experts du domaine ? Quelque chose entre les deux ?

Un des principes de base de Domain-Driven Design est d'utiliser un langage basé sur le modèle. Puisque le modèle est le terrain d'entente, l'endroit où le logiciel rencontre le domaine, il est justifié de l'utiliser comme terrain de construction de ce langage.

Utilisez le modèle comme colonne vertébrale d'un langage. Demandez à ce que l'équipe utilise le langage assidûment dans tous les échanges, et aussi dans le code. Quand elle partage la connaissance et dessine les contours du modèle, l'équipe utilise l'expression orale, écrite et des diagrammes. Assurez-vous que le langage apparaisse en continu dans toutes les formes de communication utilisées par l'équipe ; c'est en cela que le langage est appelé *Langage omniprésent*.

Le Langage omniprésent fait le lien entre toutes les parties du design, et jette les fondements d'un bon fonctionnement de l'équipe de conception. Cela prend des mois et même des années pour que puisse prendre forme le design de projets de grande envergure. Les membres de l'équipe découvrent que certains des concepts initiaux étaient inexacts ou utilisés mal à propos, ou ils découvrent de nouveaux éléments du

design qui nécessitent d'être examinés et insérés dans la conception globale. Tout cela n'est pas possible sans un langage commun.

Les langages n'apparaissent pas du jour au lendemain. Ca demande un sérieux travail et beaucoup de concentration pour s'assurer que les éléments clés du langage voient le jour. Il nous faut trouver ces concepts clés qui définissent le domaine et la conception, trouver les mots qui leur correspondent, et commencer à les utiliser. Certains d'entre eux sont faciles à repérer, mais il y en a des plus ardues.

Éliminez les difficultés en testant des expressions différentes, qui reflètent des modèles alternatifs. Puis refactorisez le code, en renommant les classes, les méthodes et les modules pour vous conformer au nouveau modèle. Trouvez une solution au flou entourant certains termes grâce à la conversation, exactement de la même manière qu'on s'accorde sur la signification de mots ordinaires.

Construire un langage comme cela conduit à un résultat manifeste : le modèle et le langage sont fortement interconnectés l'un à l'autre. Un changement dans le langage doit devenir un changement dans le modèle.

Les experts du domaine devraient s'opposer à des termes et des structures maladroites ou inaptes à véhiculer la compréhension du domaine. S'il y a des choses que les experts du domaine ne comprennent pas dans le modèle ou le langage, alors il est plus probable que quelque chose cloche dans ce dernier. D'un autre côté, les développeurs devraient surveiller les ambiguïtés et les incohérences qui auront tendance à apparaître dans la conception.

Créer le Langage omniprésent

Comment pouvons-nous commencer à créer un langage ? Voici un dialogue hypothétique entre un développeur logiciel et un expert du domaine dans le projet de surveillance du trafic aérien. Faites bien attention aux mots qui apparaissent en gras.

Développeur : Nous voulons surveiller le trafic aérien. Par où est-ce qu'on commence ?

Expert : Commençons par la base. Tout ce trafic est composé d'avions. Chaque avion décolle d'un lieu de **départ**, et atterrit sur un lieu de **destination**.

Développeur : Ca, c'est simple. Une fois en vol, l'avion peut opter pour n'importe quel chemin aérien choisi par les pilotes ? Est-ce que c'est de leur ressort de décider la voie à prendre, du moment qu'ils atteignent la destination ?

Expert : Oh, non. Les pilotes reçoivent une **route** à suivre. Et ils doivent rester aussi proches que possible de cette route.

Développeur : Je vois cette **route** comme un chemin en 3D dans le ciel. Si nous utilisons un système de coordonnées cartésiennes, alors la route est simplement une série de points 3D.

Expert : Je ne pense pas. Nous ne voyons pas la **route** comme ça. La route est en fait la projection sur le sol du chemin aérien attendu de l'appareil. La **route** passe par une série de points au sol déterminés par leur **latitude** et **longitude**.

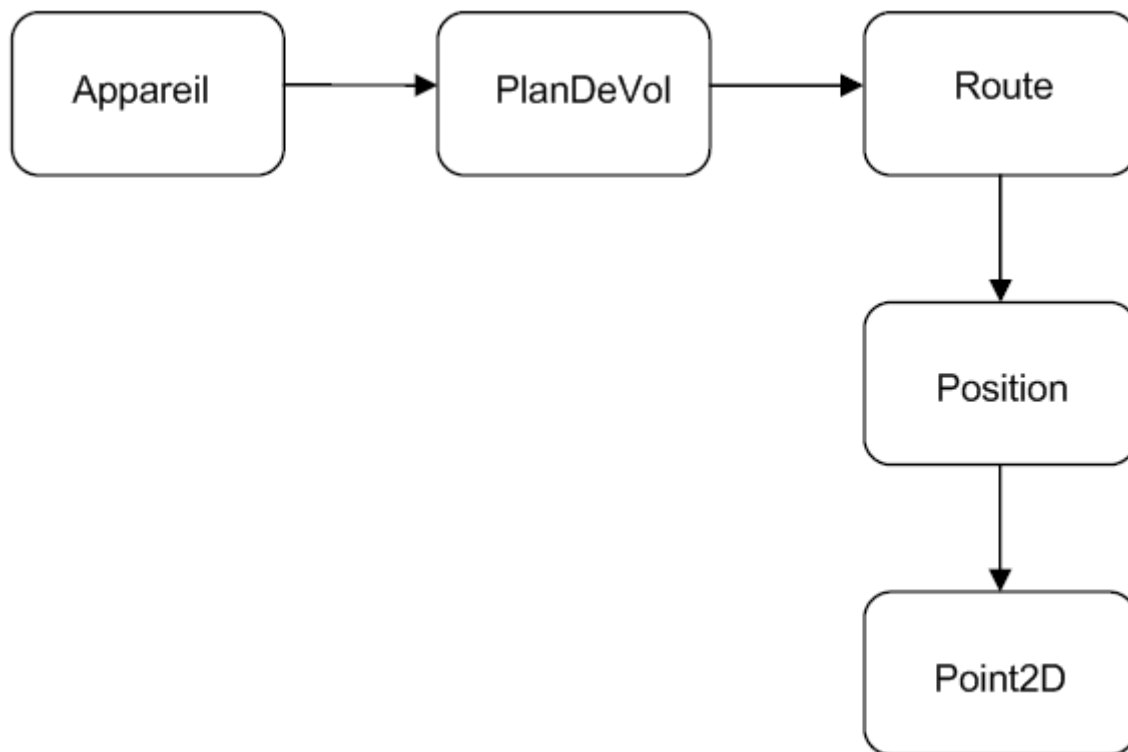
Développeur : OK, appelons alors chacun de ces points une **position**, puisque c'est un point fixe à la surface de la Terre. Ensuite on utilisera une série de points 2D pour décrire le chemin. Et au passage, le **départ** et la **destination** sont simplement des **positions**. On ne devrait pas les considérer comme des concepts à part. La route atteint sa **destination** comme elle atteint n'importe quelle autre **position**. L'avion doit suivre la route, mais est-ce que ça veut dire qu'il peut voler aussi haut ou aussi bas qu'il veut ?

Expert : Non. L'altitude qu'un appareil doit avoir à un moment donné est aussi établie dans le **plan de vol**.

Développeur : **Plan de vol** ? Qu'est-ce que c'est ?

Expert : Avant de quitter l'aéroport, les pilotes reçoivent un **plan de vol** détaillé qui comprend toutes sortes d'information sur le **vol** : la route, l'**altitude** de croisière, la **vitesse** de croisière, le type d'**appareil**, et même des informations sur les membres de l'équipage.

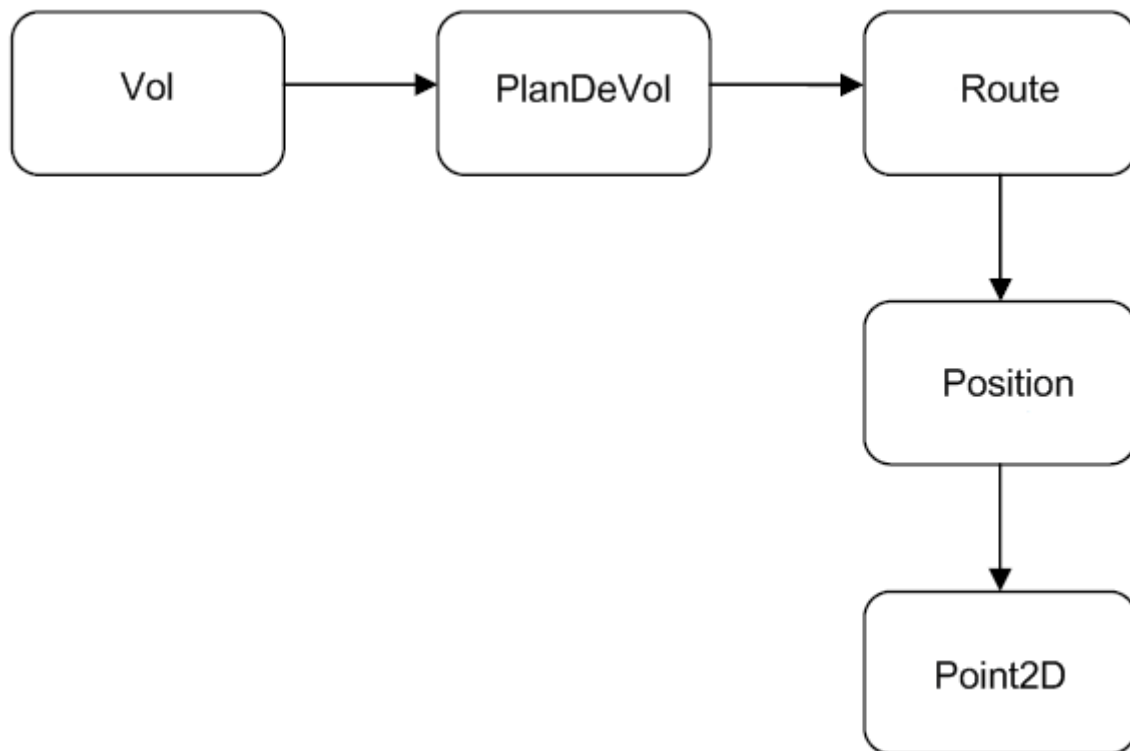
Développeur : Hmm, le **plan de vol** m'a l'air assez important. On va l'inclure dans le modèle.



Développeur : Voilà qui est mieux. Maintenant que je le vois, je réalise quelque chose. Quand on surveille le trafic aérien, en fait on ne s'intéresse pas aux avions eux-mêmes, s'ils sont blancs ou bleus, ou si c'est des Boeing ou des Airbus. On s'intéresse à leur **vol**. C'est ça qu'on trace et qu'on mesure, en réalité. Je pense qu'on devrait changer le modèle un petit peu pour le rendre plus précis.

Remarquez comment cette équipe, en parlant du domaine de la surveillance du trafic aérien autour de son modèle naissant, est lentement en train de créer un langage constitué des mots en gras. Notez aussi la manière dont ce langage change le modèle !

Cependant, dans la vie réelle ce dialogue est beaucoup plus bavard, et les gens parlent très souvent des choses indirectement, entrent trop dans les détails, ou choisissent de mauvais concepts ; ceci peut rendre la formation du langage très difficile. Pour commencer à attaquer ce problème, tous les membres de l'équipe devraient être conscients du besoin de créer un langage commun et on devrait leur rappeler de rester concentrés sur l'essentiel, et d'utiliser le langage à chaque fois que c'est nécessaire. Nous devrions utiliser notre jargon aussi peu que possible pendant ce genre de séance, et on devrait se servir du Langage omniprésent car il nous aide à communiquer clairement et précisément.



Il est aussi fortement recommandé aux développeurs d'implémenter les concepts principaux du modèle dans le code. On pourrait écrire une classe pour Route et une autre pour Position. La classe Position pourrait hériter d'une classe Point2D, ou pourrait contenir un Point2D comme attribut principal. Cela dépend d'autres facteurs dont nous discuterons plus tard. En créant des classes pour les concepts du modèle correspondants, nous faisons du mapping entre le modèle et le code, et entre le langage et le code. C'est très utile car cela rend le code plus lisible, et en fait une reproduction du modèle. Le fait d'avoir un code qui exprime le modèle porte ses fruits plus tard dans le projet, lorsque le modèle devient gros, et lorsque des changements dans le code peuvent avoir des conséquences indésirables si le code n'a pas été conçu proprement.

Nous avons vu comment le langage est partagé par l'équipe entière, et aussi comment il aide à bâtir la connaissance et à créer le modèle. Que devrions-nous employer pour communiquer le langage ? Simplement l'expression orale ? Nous avons utilisé des diagrammes. Quoi d'autre ? De l'écrit ?

Certains diront qu'UML est suffisamment bon pour construire un modèle entièrement dessus. Et en effet, c'est un très bon outil pour coucher sur le papier des concepts clés comme les classes, et pour exprimer les relations qui existent entre elles. Vous pouvez dessiner trois ou quatre classes sur un calepin, écrire leurs noms, et montrer leurs liaisons. C'est très simple pour tout le monde de suivre le cheminement de votre pensée, et l'expression graphique d'une idée est facile à comprendre. Tout le monde partage instantanément la même vision d'un sujet donné, et il devient plus simple de

communiquer en se basant là-dessus. Quand de nouvelles idées apparaissent, le diagramme est modifié pour refléter le changement conceptuel.

Les diagrammes UML sont très utiles lorsque le nombre d'éléments en jeu est petit. Mais de l'UML, ça peut pousser comme des champignons après une pluie d'été. Que faire quand on a des centaines de classes qui occupent une feuille de papier aussi longue que le Mississippi ? C'est difficilement lisible même par des spécialistes en logiciel, sans parler des experts du domaine. Ils n'y comprendront pas grand-chose lorsque tout ça enflera, et ça enfle même dans des projets de taille moyenne.

De plus, UML sait bien exprimer les classes, leurs attributs et leurs relations. Mais les comportements des classes et les contraintes ne se formulent pas aussi facilement. Pour cela, UML a recours à du texte placé sous forme de notes dans le diagramme. Donc UML ne sait pas véhiculer deux aspects importants d'un modèle : la signification des concepts qu'il représente et ce que l'objet est censé faire. Mais ça ne fait rien, puisque nous pouvons y ajouter d'autres outils de communication.

On peut utiliser des documents. Une manière conseillée de communiquer le modèle est de faire de petits diagrammes, chacun contenant un sous-ensemble du modèle. Ces diagrammes contiendront plusieurs classes et leurs relations. Cela regroupe déjà une bonne partie des concepts en jeu. Puis nous pouvons ajouter du texte au diagramme. Le texte va expliquer le comportement et les contraintes que le diagramme ne peut pas représenter. Chaque sous-section comme celle-là tente d'expliquer un aspect important du domaine, elle braque un « projecteur » pour éclairer une partie du domaine.

Ces documents peuvent même être dessinés à main levée, car ça transmet l'impression qu'ils sont temporaires, et pourraient être changés dans un futur proche, ce qui est vrai parce que le modèle est modifié de nombreuses fois au début avant d'atteindre un état plus stable.

Il pourrait être tentant d'essayer de créer un grand diagramme couvrant le modèle tout entier. Toutefois, la plupart du temps de tels diagrammes sont presque impossibles à constituer. Et qui plus est, même si vous parvenez à fabriquer ce diagramme unique, il sera si encombré qu'il ne véhiculera pas la compréhension mieux que le ferait l'assortiment de petits diagrammes.

Faites attention aux longs documents. Ils prennent beaucoup de temps à écrire, et peuvent devenir obsolètes avant d'être terminés. Les documents doivent être synchronisés avec le modèle. De vieux documents qui utilisent le mauvais langage et ne reflètent pas le modèle ne sont pas très utiles. Essayez de les éviter quand c'est possible.

Il est aussi possible de communiquer en utilisant du code. La communauté XP plaide abondamment pour cette approche. Du code bien écrit peut être très communicatif.

Bien que le comportement exprimé par une méthode soit clair, le nom de la méthode est-il aussi clair que son corps ? Les assertions d'un test parlent d'elles-mêmes, mais qu'en est-il du nom des variables et de la structure générale du code ? Donnent-ils haut et fort une version intégrale des faits ? Du code qui fonctionnellement fait ce qu'il y a à faire, n'exprime pas forcément ce qu'il y a à exprimer. Ecrire un modèle en code est très difficile.

Il y a d'autres façons de communiquer lors de la conception. Ce n'est pas l'objectif de ce livre de les présenter toutes. Une chose est néanmoins claire : l'équipe de conception, composée d'architectes logiciels, de développeurs et d'experts du domaine, a besoin d'un langage qui unifie leurs actions, et les aide à créer un modèle et à l'exprimer dans du code.

3

Conception dirigée par le Modèle

Les chapitres précédents soulignaient l'importance d'une approche du développement logiciel centrée sur le domaine métier. Nous avons dit qu'il était fondamentalement important de créer un modèle profondément enraciné dans le domaine, et qui reflète les concepts essentiels du domaine avec une grande précision. Le Langage omniprésent devrait être pratiqué pleinement tout au long du processus de modélisation de manière à faciliter la communication entre les spécialistes logiciels et les experts du domaine, et à découvrir les concepts clés du domaine qui devront être utilisés dans le modèle. L'objectif de ce processus de modélisation est de créer un bon modèle. L'étape suivante est l'implémentation du modèle en code. C'est une phase tout aussi importante du processus de développement logiciel. Si vous avez créé un excellent modèle mais que vous échouez à le transférer correctement dans le code, il en résultera en un logiciel de qualité discutable.

Il arrive que les analystes logiciels travaillent avec les experts du domaine métier pendant des mois, découvrent les éléments fondamentaux du domaine, mettent en lumière leurs relations, et créent un modèle correct, qui capture le domaine de façon exacte. Puis le modèle est passé aux développeurs. Ce qui peut se produire, c'est que les développeurs regardent le modèle et découvrent que certains des concepts ou relations qui s'y trouvent ne peuvent pas être correctement exprimés par du code. Ils utilisent donc le modèle original comme source d'inspiration, mais ils créent leur propre conception qui emprunte des idées du modèle, et y ajoutent certaines de leurs propres idées. Le processus de développement se poursuit, et davantage de classes sont ajoutées au code, creusant l'écart entre le modèle d'origine et l'implémentation finale. On n'est plus assuré d'avoir le bon résultat à la fin. Il se peut que de bons développeurs mettent sur pied un produit qui marche, mais résistera-t-il à l'épreuve du temps ? Sera-t-il facilement extensible ? Sera-t-il aisément maintenable ?

Tout domaine peut être exprimé à travers de nombreux modèles, et tout modèle peut se traduire de diverses manières dans le code. A chaque problème particulier il peut y

avoir plus d'une solution. Laquelle choisir ? Avoir un modèle analytiquement correct ne signifie pas qu'il puisse être directement exprimé en code. A ce stade, la question principale est la suivante : comment allons-nous aborder la transition du modèle vers le code ?

Parmi les techniques de conception parfois préconisées, il y a ce qu'on appelle le *modèle d'analyse*, qui est vu comme étant séparé du design du code et généralement fait par des personnes différentes. Le modèle d'analyse est le produit de l'analyse du domaine métier, il se traduit par un modèle qui ne prend pas en compte le logiciel utilisé dans l'implémentation. On se sert de ce genre de modèle pour comprendre le domaine. Un certain niveau de connaissance est bâti, et le modèle résultant peut être analytiquement correct. Le logiciel n'est pas pris en compte à cette étape car on le considère comme un facteur de confusion. Ce modèle parvient ensuite aux développeurs qui sont censés faire la conception. Comme le modèle n'a pas été construit avec des principes de conception en tête, il ne servira probablement pas bien cet objectif. Les développeurs devront l'adapter, ou créer une conception séparée. Et il cesse d'y avoir un mappage entre le modèle et le code. Résultat, les modèles d'analyse sont rapidement abandonnés après qu'on ait commencé à coder.

Un des principaux problèmes de cette approche est que les analystes ne peuvent pas prévoir certains défauts de leur modèle, ni toutes les subtilités du domaine. Il arrive que les analystes soient allés trop dans les détails de certains composants du modèle, et n'en aient pas assez précisé d'autres. On découvre des détails très importants pendant le processus de conception et d'implémentation. Il peut s'avérer qu'un modèle fidèle au domaine ait de sérieux problèmes avec la persistance des objets, ou un comportement inacceptable en termes de performances. Les développeurs seront obligés de prendre des décisions par eux-mêmes, et apporteront des modifications à la conception pour résoudre un problème réel qui n'avait pas été pris en compte lorsque le modèle avait été créé. Ils produisent une conception qui s'éloigne du modèle, le rendant moins pertinent.

Si les analystes travaillent indépendamment, ils vont finalement créer un modèle. Quand ce modèle est transmis aux concepteurs, une partie de la connaissance du domaine et du modèle détenue par les analystes est perdue. Même si on peut exprimer le modèle à travers des diagrammes et des documents écrits, il y a des chances pour que les concepteurs ne saisissent pas tout le sens du modèle, les relations entre certains objets, ou leur comportement. Il y a des détails d'un modèle qui ne s'expriment pas facilement sous forme de diagramme, et ne peuvent pas être présentés dans leur intégralité même dans des documents écrits. Il ne sera pas facile pour les développeurs de les saisir. Dans certains cas ils feront des suppositions sur le comportement voulu, et il est possible qu'ils fassent les mauvaises, débouchant sur un fonctionnement incorrect du programme.

Les analystes tiennent leurs propres réunions fermées où l'on débat de beaucoup de choses autour du domaine, et où il y a beaucoup de partage de connaissance. Ils créent un modèle censé contenir toute l'information sous forme condensée, et les développeurs doivent en assimiler l'intégralité en lisant les documents qu'on leur donne. Ce serait bien plus productif si les développeurs pouvaient se joindre aux réunions des analystes et ainsi accéder à une vue claire, complète du domaine et du modèle avant de commencer à concevoir le code.

Une meilleure approche est de lier étroitement modélisation du domaine et conception. Le modèle devrait être construit en gardant un œil sur les considérations logicielles et conceptuelles. On devrait intégrer les développeurs dans le processus de modélisation. L'idée importante est de choisir un modèle qui puisse être exprimé convenablement dans du logiciel, pour que le processus de conception soit direct et basé sur le modèle. Le fait de coupler étroitement le code à un modèle sous-jacent donne du sens au code et rend le modèle pertinent.

Impliquer les développeurs permet d'avoir du feedback. Cela permet de s'assurer que le modèle peut être implémenté en logiciel. Si quelque chose ne va pas, le problème est identifié à un stade précoce, et on peut facilement le corriger.

Ceux qui écrivent le code devraient très bien connaître le modèle, et se sentir responsables de son intégrité. Ils devraient réaliser qu'un changement dans le code implique un changement dans le modèle ; sans quoi ils refactoriseront le code jusqu'au point où il aura cessé d'exprimer le modèle original. Si l'analyste est dissocié du processus d'implémentation, il perdra rapidement toute préoccupation concernant les limites introduites par le développement. Le résultat de cela sera un modèle qui n'est pas pragmatique.

Toute personne technique qui contribue à l'élaboration du modèle doit passer du temps à manipuler le code, quel que soit le rôle premier qu'il ou elle joue dans le projet. Toute personne ayant la responsabilité de modifier du code doit apprendre à exprimer un modèle à travers le code. Chaque développeur doit être impliqué, à un niveau ou un autre, dans la discussion autour du modèle et être en contact avec les experts du domaine. Ceux qui contribuent de manière non technique doivent engager consciemment, avec ceux qui manipulent le code, un échange dynamique d'idées sur le modèle à travers le Langage omniprésent.

Si on ne peut pas faire la correspondance entre la conception, ou une partie centrale de celle-ci, et le modèle du domaine, ce modèle n'aura que peu de valeur, et la justesse du logiciel sera suspecte. Qui plus est, les mappages complexes entre modèles et fonctions conceptuelles seront difficiles à comprendre, et, en pratique, impossibles à maintenir lorsque la conception changera. Un fossé mortel s'ouvrira entre l'analyse et

la conception si bien que le discernement qu'on gagnera à pratiquer l'une des deux activités ne nourrira pas l'autre.

Concevez toute portion du système logiciel de telle sorte qu'elle reflète le modèle du domaine de manière très littérale, de cette manière le mappage sera évident. Revisitez ensuite le modèle et modifiez-le pour qu'il puisse être implémentable plus naturellement dans le logiciel, même si votre but reste une représentation fine du domaine. Exigez un modèle unique qui serve bien ces deux objectifs, en plus de supporter un Langage omniprésent fluide.

Puisez dans le modèle la terminologie à utiliser pour la conception et l'affectation des responsabilités de base. Le code devient une expression du modèle, donc il se peut qu'un changement apporté au code se traduise en un changement du modèle. Son effet doit se propager en conséquence à travers le reste des activités du projet.

Lier étroitement l'implémentation à un modèle nécessite généralement des langages et outils de développement qui supportent un paradigme de modélisation, comme par exemple la programmation orientée objet.

La programmation orientée objet convient bien à l'implémentation de modèle car ils sont tous deux basés sur le même paradigme. La programmation orientée objet prévoit des classes d'objets et leurs associations, des instances d'objets, et l'envoi de messages entre eux. Les langages de POO rendent possible la création de mappages directs entre les objets du modèle avec leurs relations, et leurs équivalents en programmation.

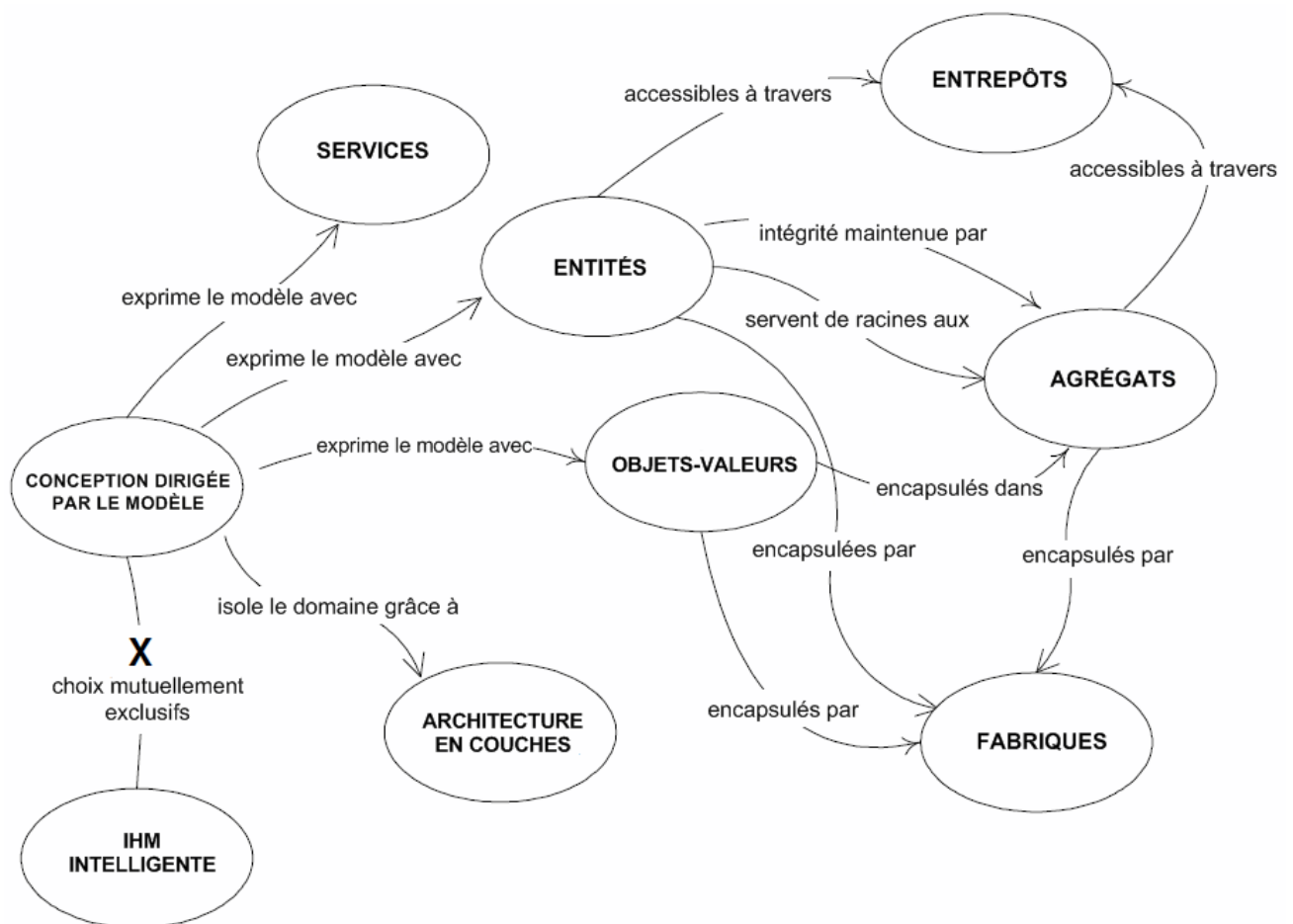
Les langages procéduraux offrent un support limité de la conception dirigée par le Modèle. Ces langages ne proposent pas les briques nécessaires pour implémenter les composants-clés d'un modèle. Certains disent qu'on peut faire de la POO avec un langage procédural comme le C, et en effet, une partie des fonctionnalités peuvent être reproduites de cette façon. Les objets peuvent être simulés par des structures de données. Ce genre de structure ne contient pas le comportement de l'objet, il faut le rajouter séparément dans des fonctions. La signification de telles données existe seulement dans l'esprit du développeur, car le code lui-même n'est pas explicite. Un programme écrit dans un langage procédural est généralement perçu comme un ensemble de fonctions, l'une appelant l'autre, qui travaillent ensemble pour atteindre un résultat donné. Un programme comme cela est incapable d'encapsuler facilement des connexions conceptuelles, ce qui rend le mappage entre le domaine et le code difficile à réaliser.

Certains domaines spécifiques, comme les mathématiques, sont facilement modélisables et implémentables en programmation procédurale, car bien des théories mathématiques se traitent facilement en utilisant des appels de fonctions et des structures de données, en effet il s'agit avant tout de calculs. Il y a des domaines plus

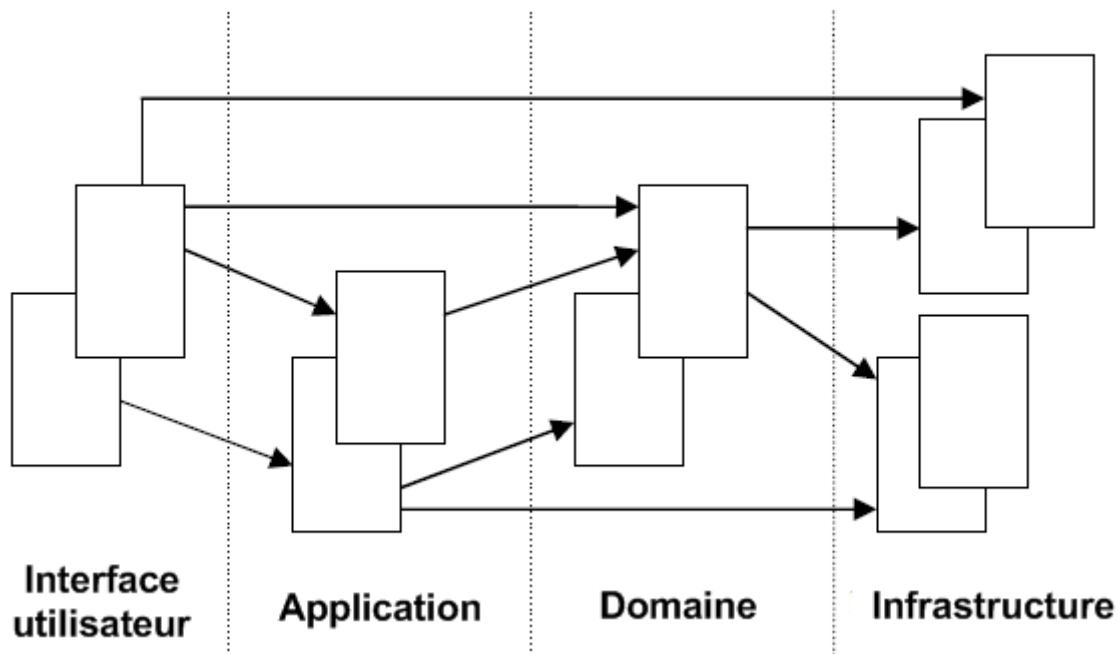
complexes qui ne sont pas juste une suite de concepts abstraits impliquant des calculs, et ne peuvent pas être réduits à un jeu d'algorithmes, c'est pourquoi les langages procéduraux sont bien en peine d'exprimer ces modèles. Pour cette raison, la programmation procédurale n'est pas recommandée en conception dirigée par le modèle.

Les blocs de construction d'une conception orientée Modèle

Les sections suivantes de ce chapitre exposent les principaux patterns utilisés en conception dirigée par le modèle. Le diagramme qui suit est une carte des patterns présentés et des relations qui existent entre eux.



L'architecture en couches



Lorsqu'on crée une application logicielle, une grande part de l'application n'est pas reliée directement au domaine, mais fait partie de l'infrastructure ou sert le logiciel lui-même. Il peut arriver, et c'est justifié, que la partie domaine d'une application soit assez réduite comparée au reste, puisqu'une application typique contient beaucoup de code lié à l'accès à la base de données, à l'accès aux fichiers ou au réseau, aux interfaces utilisateur, etc.

Dans un programme orienté objet, le code de l'IHM, de la base de données, et d'autres modules de support se retrouve souvent directement écrit dans les objets métier. Une partie de la logique métier est quant à elle embarquée dans le comportement des composants d'IHM et des scripts de base de données. Cela arrive parfois car c'est la façon la plus simple de faire marcher les choses rapidement.

Cependant, quand du code relatif au domaine est mélangé aux autres couches, il devient très difficile de garder un œil dessus et de s'en souvenir. Des changements superficiels dans l'IHM peuvent en réalité modifier la logique métier. Il se peut que changer une règle métier exige une fouille méticuleuse dans le code de l'IHM, dans le code de la base de données, ou d'autres éléments du programme. Implémenter des objets orientés modèle cohérents devient difficilement praticable. Les tests automatisés deviennent peu commodes. Avec toute la technologie et toute la logique incluse dans chaque activité, on doit garder un programme très simple ou il devient impossible à comprendre.

Pour cette raison, vous devez partitionner un programme complexe en COUCHES. Imaginez une conception au sein de chaque COUCHE qui soit cohérente et ne dépende que des couches en-dessous. Suivez les modèles d'architecture standard pour fournir

un couplage faible aux couches du dessus. Concentrez tout le code lié au modèle du domaine dans une couche et isolez-le du code d'interface utilisateur, applicatif, et d'infrastructure. Les objets du domaine, libérés de la responsabilité de s'afficher eux-mêmes, de se stocker eux-mêmes, de gérer les tâches applicatives, et ainsi de suite, pourront se concentrer sur l'expression du modèle du domaine. Cela permet à un modèle d'évoluer jusqu'à être assez riche et assez clair pour capturer la connaissance essentielle du métier et la mettre en œuvre.

Une solution architecturale courante pour les conceptions dirigées par le domaine contient quatre couches conceptuelles :

Interface utilisateur (couche de présentation)	Responsable de la présentation de l'information à l'utilisateur et de l'interprétation des commandes de l'utilisateur.
Couche application	C'est une couche fine qui coordonne l'activité de l'application. Elle ne contient pas de logique métier. Elle ne recèle pas l'état des objets métier, mais elle peut détenir l'état de la progression d'une tâche applicative.
Couche domaine	Cette couche contient les informations sur le domaine. C'est le cœur du logiciel métier. L'état des objets métier est renfermé ici. La persistance des objets métier et peut-être aussi leur état est délégué à la couche infrastructure.
Couche infrastructure	Cette couche agit comme une bibliothèque de soutien pour toutes les autres couches. Elle fournit la communication entre les couches, implémente la persistance des objets métier, contient les bibliothèques auxiliaires de la couche d'interface utilisateur, etc.

Il est important de scinder une application en couches séparées, et d'établir des règles d'interaction entre elles. Si le code n'est pas clairement séparé en couches, il deviendra vite si enchevêtré qu'il sera très difficile de gérer les changements. Une simple modification dans une section du code pourra avoir des résultats inattendus et indésirables dans d'autres sections. La couche domaine devrait se concentrer sur les enjeux du cœur du domaine. Elle ne devrait pas être impliquée dans des activités d'infrastructure. L'IHM ne devrait être étroitement liée ni à la logique métier, ni aux tâches qui incombent normalement à la couche infrastructure. Une couche application est nécessaire dans beaucoup de cas : il faut qu'il y ait un gestionnaire au-dessus de la logique métier qui supervise et coordonne l'activité d'ensemble de l'application.

Par exemple, une interaction typique entre application, domaine et infrastructure pourrait ressembler à ceci : l'utilisateur veut réserver une route pour un vol, et

demande au service application dans la couche applicative de le faire. Le tiers application récupère les objets métier appropriés auprès de l'infrastructure et invoque les méthodes adéquates dessus, par exemple pour vérifier les marges de sécurité par rapport aux autres vols déjà réservés. Une fois que les objets du domaine ont fait toutes les vérifications et mis à jour leur statut à « décidé », le service application persiste les objets en passant par l'infrastructure.

Les Entités

Il y a une catégorie d'objets dont l'identité semble rester la même au fil des états du logiciel. Chez ces objets, ce ne sont pas les attributs qui comptent, mais une ligne de continuité et d'identité qui s'étend sur la durée de vie d'un système et peut s'allonger au-delà. Ces objets sont appelés Entités.

Les langages de POO conservent des instances d'objets en mémoire, et ils associent une référence ou une adresse mémoire à chaque objet. Cette référence est unique à un objet à un instant donné, mais il n'y a aucune garantie qu'elle le reste indéfiniment. En fait, c'est plutôt le contraire. Les objets sont constamment enlevés et remis dans la mémoire, ils sont sérialisés et envoyés à travers le réseau pour être recréés à l'autre bout, ou ils sont détruits. Cette référence, qui fait office d'identité pour l'environnement d'exécution du programme, n'est pas l'identité dont nous parlons. Si une classe porte une information sur la météo, comme la température, il est tout à fait possible d'avoir deux instances distinctes de cette classe qui contiennent toutes deux la même valeur. Les objets sont parfaitement égaux et interchangeables, mais ils ont des références différentes. Ce ne sont pas des entités.

Si nous devons implémenter le concept d'une personne utilisant un programme logiciel, nous créerions probablement une classe *Personne* avec une série d'attributs : nom, date de naissance, lieu de naissance, etc. Est-ce qu'un de ces attributs constitue l'identité de la personne ? Le nom ne peut pas être l'identité parce qu'il peut y avoir d'autres gens qui s'appellent pareil. On ne pourrait pas faire la distinction entre deux personnes homonymes, si on devait seulement prendre en compte leur nom. Nous ne pouvons pas non plus utiliser la date de naissance, parce qu'il y a beaucoup de gens nés le même jour. Il en va de même pour lieu de naissance. Un objet doit se distinguer des autres même s'ils peuvent avoir les mêmes attributs. Une confusion d'identité peut conduire à une corruption des données.

Considérons l'exemple d'un système de comptes bancaires. Chaque compte a son propre numéro. Un compte peut être précisément identifié par son numéro. Ce numéro reste inchangé tout au long de la vie du système, et assure la continuité. Le numéro de compte peut exister en tant qu'objet en mémoire, ou il peut être détruit de la mémoire

et envoyé à la base de données. Il peut aussi être archivé quand le compte est fermé, mais il existe toujours quelque part du moment qu'on a un intérêt à le garder sous le coude. Peu importe la représentation qui en est faite, le numéro reste le même.

Donc, implémenter des entités dans le logiciel revient à créer de l'identité. Pour une personne ça peut être une combinaison d'attributs : nom, date de naissance, lieu de naissance, nom des parents et adresse actuelle. Le numéro de sécurité sociale est aussi utilisé aux Etats-Unis pour créer l'identité. Pour un compte bancaire, le numéro de compte semble suffire. Généralement, l'identité est un attribut de l'objet, une combinaison d'attributs, un attribut spécialement créé pour préserver et exprimer l'identité, ou même un comportement. Il est important que deux objets avec des identités différentes soient facilement distingués par le système, et que deux objets de même identité soient considérés comme les mêmes. Si cette condition n'est pas satisfaite, alors le système entier peut devenir corrompu.

Il y a différentes manières de créer une identité unique pour chaque objet. L'ID peut être généré automatiquement par un module, et utilisé en interne dans le logiciel sans être rendu visible pour l'utilisateur. Ca peut être une clé primaire dans une table de la base de données, qui est certifiée unique dans la base. Lorsque l'objet est pris dans la base, son ID est récupéré et recréé en mémoire. L'ID peut aussi être créé par l'utilisateur comme c'est le cas avec les codes associés aux aéroports. Chaque aéroport a un identifiant chaîne de caractères unique qui est reconnu internationalement et utilisé par les agences de voyages du monde entier pour identifier les aéroports dans la programmation de leurs voyages. Une autre solution est d'utiliser les attributs de l'objet pour créer l'ID, et quand ça ne suffit pas, un autre attribut peut être ajouté pour aider à identifier l'objet en question.

Lorsqu'un objet se distingue par son identité plutôt que par ses attributs, faites de cela un élément primordial de sa définition dans le modèle. Gardez une définition de la classe simple et focalisée sur la continuité du cycle de vie et l'identité. Définissez une façon de distinguer chaque objet quelle que soit sa forme ou son historique. Restez vigilant par rapport aux spécifications qui demandent à retrouver des objets par leurs attributs. Définissez une opération qui garantit la production d'un résultat unique pour chaque objet, si possible en y rattachant un symbole qui est garanti unique. Ce moyen d'identification peut venir de l'extérieur, ou ça peut être un identifiant arbitraire créé par et pour le système, mais il doit correspondre aux distinctions d'identité dans le modèle. Le modèle doit définir ce que signifie « être la même chose ».

Les entités sont des objets importants d'un modèle du domaine, et elles devraient être examinées dès le début du processus de modélisation. Il est aussi important de déterminer si un objet doit être une entité ou non, c'est ce dont il est question dans le pattern suivant.

Les Objets-Valeurs

Nous avons parlé des entités et de l'importance de les reconnaître tôt dans la phase de modélisation. Les entités sont des objets nécessaires dans un modèle du domaine. Devons-nous faire de tous les objets des entités ? Chaque objet doit-il avoir une identité ?

Il pourrait être tentant de faire de tous les objets des entités. Les entités peuvent être suivies à la trace. Mais créer et tracer l'identité a un coût. On doit s'assurer que chaque instance a son identité unique, et étiqueter les identités n'est pas simple. Décider ce qui constitue une identité demande une réflexion longue et minutieuse, car une mauvaise décision mènerait à des objets de même identité, ce qu'on ne souhaite pas. Il y a aussi des répercussions sur les performances dans le fait de faire de tous les objets des entités. Il faut qu'il y ait une instance pour chaque objet. Si Client est un objet entité, alors une instance de cet objet, qui représente un client particulier d'une banque, ne peut pas être réutilisée pour des opérations sur des comptes correspondant à d'autres clients. La conséquence est qu'une telle instance doit être créée pour chaque client. Cela peut avoir comme résultat une dégradation des performances du système lorsqu'on a affaire à des milliers d'instances.

Considérons l'exemple d'une application de dessin. On présente à l'utilisateur un canevas, et il peut y dessiner tous les points et lignes de l'épaisseur, du style et de la couleur qu'il souhaite. Il est utile de créer une classe d'objets nommée Point, et le programme pourrait créer une instance de cette classe pour chaque point sur le canevas. Un point comme celui-là contiendrait deux attributs associés à ses coordonnées sur l'écran ou dans le canevas. Est-il nécessaire de considérer chaque point comme ayant une identité ? Est-ce qu'il possède une continuité ? Il semble que la seule chose qui compte chez un tel objet, ce sont ses coordonnées.

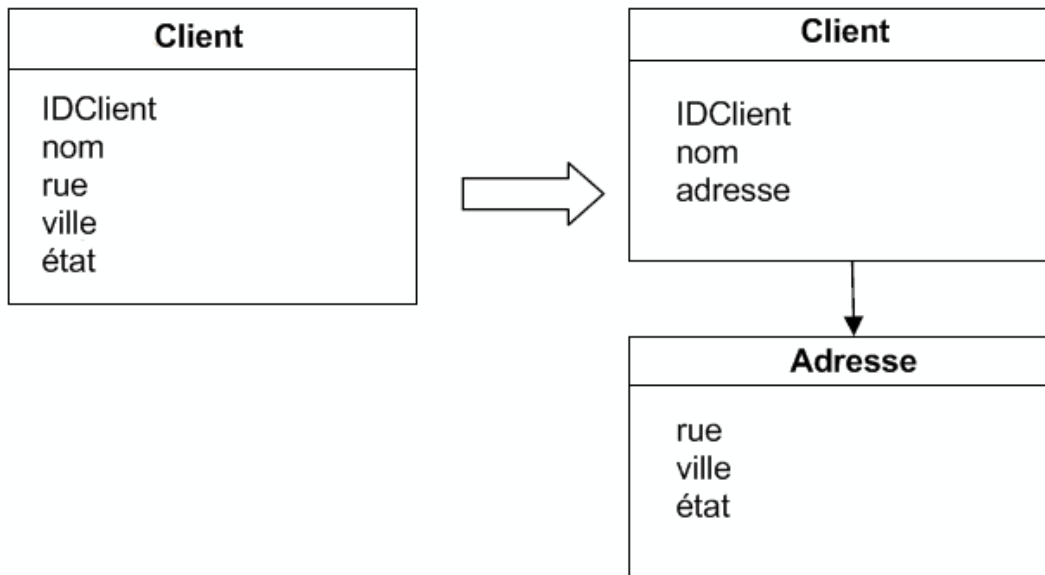
Il y a des cas où on a purement besoin de stocker des attributs d'un élément du domaine. Ce qui nous intéresse n'est pas de savoir de quel objet il s'agit, mais quels attributs il a. Un objet qui est utilisé pour décrire certains aspects d'un domaine et qui n'a pas d'identité, est appelé Objet-Valeur.

Il est nécessaire de distinguer les Objets Entités des Objets-Valeurs. Ça ne sert à rien de faire de tous les objets des entités juste pour des raisons d'uniformité. En fait, il est plutôt recommandé de choisir comme entités seulement les objets qui sont conformes à la définition d'une entité. Et de faire du reste des objets des Objets-Valeurs. (Nous présenterons un autre type d'objet dans la section suivante, mais supposons pour l'instant que nous avons seulement des objets entités et des objets-valeurs.) Ça va simplifier la conception, et il y aura d'autres conséquences positives.

N'ayant pas d'identité, les Objets-Valeurs peuvent facilement être créés et jetés. Personne n'a à se soucier de leur créer une identité, et le garbage collector règle son compte à l'objet quand il n'est plus référencé par aucun autre objet. Ca simplifie beaucoup le design.

Il est fortement recommandé que les objets-valeurs soient immuables. Ils sont créés à l'aide d'un constructeur, et jamais modifiés au cours de leur vie. Lorsque vous voulez une valeur différente pour l'objet, vous en créez tout simplement un autre. Ceci a des conséquences importantes sur la conception. Etant immuables, et n'ayant pas d'identité, les Objets-Valeurs peuvent être partagés. Ca peut s'avérer obligatoire dans certains designs. Les objets immuables sont partageables, avec d'importantes retombées en termes de performances. Ils ont aussi un caractère d'intégrité, au sens intégrité des données. Imaginez ce que cela voudrait dire de partager un objet qui n'est pas immuable. Un système de réservation de voyages en avion pourrait créer des objets pour chaque vol. Un des attributs pourrait être le code du vol. Un client réserve un vol pour une destination donnée. Un autre client veut réserver sur le même vol. Le système choisit de réutiliser l'objet qui comporte le code du vol, parce qu'il s'agit du même vol. Entretemps, le client change d'avis, et choisit de prendre un vol différent. Le système change le code du vol car il n'est pas immuable. Résultat, le code du vol du premier client change aussi.

Il y a une règle d'or : si les Objets-Valeurs sont partageables, ils doivent être immuables. Les Objets-Valeurs devraient garder une certaine minceur et une certaine simplicité. Lorsqu'un Objet-Valeur est demandé par un tiers, il peut simplement être passé par valeur, ou on peut en créer une copie et lui donner. Faire une copie d'un Objet-Valeur est simple, et généralement sans aucune conséquence. S'il n'y a pas d'identité, vous pouvez faire autant de copies que vous voulez, et toutes les détruire si nécessaire.



Les Objets-Valeurs peuvent contenir d'autres Objets-Valeurs, ils peuvent même contenir des références à des Entités. Bien que les Objets-Valeurs soient simplement utilisés pour contenir les attributs d'un objet du domaine, ça ne signifie pas qu'ils devraient comporter une longue liste de tous les attributs. Les attributs peuvent être regroupés dans différents objets. Les attributs qu'on choisit pour constituer un Objet-Valeur devraient former un tout conceptuel. A un client, on associe un nom, une rue, une ville et un Etat. C'est mieux si les informations sur l'adresse sont contenues dans un objet séparé, et l'objet client aura une référence à cet objet. La rue, la ville et l'Etat devraient avoir leur propre objet, l'Adresse, car ils ont conceptuellement leur place ensemble, plutôt qu'être des attributs séparés du client, comme indiqué dans le diagramme ci-dessus.

Les Services

Quand on analyse le domaine et qu'on essaye de déterminer les principaux objets qui le composent, on découvre que certains aspects du domaine ne se transposent pas facilement en objets. On considère de manière générale que les objets ont des attributs, un état interne qu'ils gèrent, et qu'ils exposent un comportement. Lorsqu'on élabore le Langage omniprésent, les concepts clés du domaine sont introduits dans le langage, et les noms de ce dernier sont facilement convertibles en objets. Les verbes du langage, associés aux noms correspondants, viennent former une partie du comportement de ces objets. Mais il y a certaines actions dans le domaine, certains verbes, qui ne semblent appartenir à aucun objet. Ils représentent un comportement important du domaine, ils ne peuvent donc être négligés ou simplement incorporés dans des Entités ou des Objets-Valeurs. Ajouter un tel comportement à un objet le corromprait, le ferait

incarner une fonctionnalité qui ne lui appartient pas. Néanmoins, utilisant un langage orienté objet, nous sommes obligés d'employer un objet à cet effet. Il ne peut pas simplement y avoir une fonction séparée, toute seule. Elle doit être rattachée à un objet. Souvent, ce genre de comportement fonctionne à travers plusieurs objets, potentiellement de classes différentes. Par exemple, transférer de l'argent d'un compte à un autre : cette fonction devrait-elle se trouver dans le compte qui envoie ou dans le compte qui reçoit ? L'un et l'autre semblent tout autant mal placés.

Lorsque ce genre de comportement est identifié dans le domaine, la meilleure méthode est de le déclarer en tant que Service. Un tel objet n'a pas d'état interne, et son objectif est simplement de fournir de la fonctionnalité au domaine. L'assistance fournie par un Service peut être non négligeable, et un service peut regrouper des fonctionnalités connexes qui servent aux Entités et aux Objets-Valeurs. C'est beaucoup mieux de déclarer le Service explicitement, parce que cela crée une distinction claire dans le domaine, ça encapsule un concept. Mettre ce genre de fonctionnalité dans une Entité ou un Objet-Valeur engendre de la confusion, car ce que ces objets représentent n'apparaîtra pas clairement.

Les services agissent comme des interfaces qui fournissent des opérations. Les services sont courants dans les frameworks techniques, mais ils peuvent aussi être utilisés dans la couche domaine. L'intérêt d'un Service ne réside pas dans l'objet qui rend le service, il est plutôt lié aux objets sur ou pour le compte desquels les opérations sont effectuées. Par conséquent, il n'est pas inhabituel qu'un Service devienne le point de connexion de beaucoup d'objets. C'est une des raisons pour lesquelles un comportement qui appartient naturellement à un Service ne devrait pas être inclus dans les objets du domaine. Si on intègre une telle fonctionnalité dans les objets du domaine, un réseau d'associations dense se crée entre eux et les objets bénéficiaires des opérations. Un degré élevé de couplage entre de nombreux objets est le signe d'un piètre design parce que ça rend le code difficile à lire et à comprendre, et, plus important encore, ça le rend difficile à changer.

Un Service ne doit pas remplacer une opération qui appartient normalement aux objets du domaine. On ne devrait pas créer un Service pour chaque opération dont on a besoin. Mais lorsqu'il ressort qu'une telle opération est un concept important du domaine, un Service doit être créé pour ça. Un service compte 3 caractéristiques :

1. L'opération exécutée par le Service fait référence à un concept du domaine qui n'appartient pas naturellement à une Entité ou un Objet-Valeur.
2. L'opération effectuée fait référence à d'autres objets du domaine.
3. L'opération n'a pas d'état.

Quand un processus ou une transformation significative du domaine n'est pas la responsabilité naturelle d'une Entité ou d'un Objet-Valeur, vous pouvez ajouter une opération au modèle sous la forme d'une interface autonome déclarée en tant que Service. Définissez l'interface dans des termes du langage du modèle et assurez-vous que le nom de l'opération fait partie du Langage omniprésent. Faites en sorte que le Service n'ait pas d'état.

Quand on utilise des Services, il est important de préserver l'isolation de la couche domaine. On peut facilement confondre les services qui appartiennent à la couche du domaine, et ceux qui appartiennent à l'infrastructure. Il peut aussi y avoir des services dans la couche application, ce qui ajoute encore un niveau de complexité. Ces services sont encore plus difficiles à séparer de leurs homologues résidant dans la couche domaine. Au cours du travail sur le modèle et pendant la phase de conception, on doit s'assurer que le niveau domaine reste isolé des autres niveaux.

Les Services de l'application et les Services du domaine sont généralement tous deux construits autour des Entités et des Valeurs du domaine, ils fournissent des fonctionnalités qui sont directement reliées à ces objets. Décider à quelle couche un Service appartient n'est pas chose aisée. Si l'opération effectuée appartient conceptuellement à la couche application, alors c'est là qu'on devrait placer le Service. Si l'opération concerne des objets du domaine, si elle est strictement liée au domaine, et répond à un besoin du domaine, alors elle devrait appartenir à la couche domaine.

Considérons un exemple pratique, une application web de reporting. Les rapports utilisent des données stockées dans une base, et ils sont générés à partir de gabarits. Le résultat final est une page HTML présentée à l'utilisateur dans un navigateur web.

La couche d'IHM est contenue dans des pages web et permet à l'utilisateur de se connecter, de choisir le rapport souhaité et cliquer sur un bouton pour le lancer. La couche application est une couche fine qui se trouve entre l'interface utilisateur, le domaine et l'infrastructure. Elle interagit avec l'infrastructure de la base de données pendant les opérations de connexion, et avec la couche domaine lorsqu'elle a besoin de créer des rapports. La couche domaine va contenir le cœur du domaine, les objets directement reliés aux rapports. Rapport et Gabarit sont deux de ces objets, sur lesquels sont basés les rapports. La couche infrastructure assurera l'accès à la base de données et aux fichiers.

Quand un utilisateur sélectionne un rapport à créer, il sélectionne en réalité le nom d'un rapport dans une liste de noms. C'est le reportID, une chaîne de caractères. D'autres paramètres sont passés, comme les éléments à afficher dans le rapport et l'intervalle de temps des données qu'il contient. Mais nous parlerons seulement du reportID pour plus de simplicité. Ce nom est passé par le biais de la couche application à la couche domaine. Etant donné le nom, la couche domaine est responsable de la

création et du retour du rapport. Puisque les rapports sont basés sur des gabarits, on pourrait créer un Service, dont le but serait d'obtenir le gabarit qui correspond à un reportID. Ce gabarit est stocké dans un fichier ou en base de données. L'objet Rapport lui-même n'est pas l'endroit approprié pour mettre une telle opération. Elle n'appartient pas à l'objet Gabarit non plus. Donc on crée un Service à part dont l'objectif est de récupérer le gabarit d'un rapport en se basant sur l'ID du rapport. Ce serait un service situé dans la couche du domaine. Il se servirait de l'infrastructure fichier pour aller chercher le gabarit sur le disque.

Les Modules

Dans une application vaste et complexe, le modèle a tendance à devenir de plus en plus gros. Le modèle atteint un point où il est difficile de parler de lui dans sa globalité, et où il devient compliqué de comprendre les relations et interactions entre ses différentes parties. C'est la raison pour laquelle il est nécessaire d'organiser le modèle en modules. Les modules sont utilisés comme méthode d'organisation des concepts et tâches connexes en vue de réduire la complexité.

Les modules sont largement utilisés dans la plupart des projets. Il est plus facile de saisir le tableau d'ensemble d'un gros modèle si vous regardez les modules qu'il contient, puis les relations entre ces modules. Une fois qu'on a compris l'interaction entre les modules, on peut commencer à s'intéresser aux détails à l'intérieur d'un module. C'est une manière simple et efficace de gérer la complexité.

Une autre raison d'utiliser des modules est liée à la qualité du code. Il est largement reconnu que le code logiciel doit avoir un fort niveau de cohésion et un faible niveau de couplage. Même si la cohésion commence à l'échelle de la classe et de la méthode, elle peut être appliquée à l'échelle du module. Il est recommandé de regrouper les classes fortement liées dans des modules pour fournir le maximum de cohésion possible. Il y a plusieurs types de cohésion. Deux des plus répandues sont la *cohésion communicationnelle* et la *cohésion fonctionnelle*. La cohésion communicationnelle est effective lorsque des parties du module opèrent sur les mêmes données. Ça a du sens de les regrouper, car il existe une forte relation entre elles. La cohésion fonctionnelle est atteinte quand toutes les parties du module travaillent de concert pour réaliser une tâche bien définie. Elle est considérée comme la meilleure forme de cohésion.

Utiliser des modules dans la conception est une façon de renforcer la cohésion et de diminuer le couplage. Les modules devraient être constitués d'éléments qui vont de pair fonctionnellement ou logiquement, ce qui assure la cohésion. Les modules devraient avoir des interfaces bien définies auxquelles accèdent d'autres modules. Au lieu d'appeler trois objets d'un module, il est mieux d'accéder à une interface, car cela

réduit le couplage. Un couplage faible diminue la complexité, et augmente la maintenabilité. Il est plus facile de comprendre comment un système fonctionne quand il y a peu de connexions entre des modules qui assurent des tâches bien définies, que quand chaque module a des tas de connexions avec tous les autres.

Choisissez des Modules qui incarnent la nature du système et renferment un ensemble de concepts cohérents. Cela mène souvent à un couplage faible entre modules, mais si ce n'est pas le cas, trouvez une manière de modifier le modèle pour débroussailler les concepts, ou cherchez un concept que vous auriez ignoré et qui pourrait être la base d'un Module rassemblant les éléments de manière sensée. Visez un couplage faible, dans le sens de concepts qu'on peut comprendre et réfléchir indépendamment les uns des autres. Raffinez le modèle jusqu'à ce qu'il soit partitionné selon des concepts de haut niveau du domaine et que le code correspondant soit également découplé.

Donnez aux Modules des noms qui deviendront partie intégrante du Langage omniprésent. Les modules et leurs noms doivent donner un aperçu du domaine.

Les concepteurs ont l'habitude de créer des modules dès le début. Ce sont des éléments ordinaires de nos conceptions. Après que le rôle du module ait été décidé, il reste généralement inchangé, tandis que l'intérieur du module peut varier beaucoup. Il est recommandé de s'aménager de la flexibilité, de permettre aux modules d'évoluer avec le projet et de ne pas les figer. C'est vrai qu'un refactoring de module peut être plus coûteux qu'un refactoring de classe, mais lorsqu'une erreur de design d'un module est constatée, il vaut mieux s'y attaquer en changeant le module qu'en trouvant des solutions palliatives.

Les Agrégats

Les trois derniers patterns de ce chapitre vont traiter d'un autre défi de modélisation, un défi lié au cycle de vie des objets du domaine. Les objets du domaine passent par une série d'états au cours de leur vie. Ils sont créés, placés en mémoire et utilisés dans des traitements, puis ils sont détruits. Dans certains cas ils sont sauvegardés dans des emplacements permanents, comme une base de données, où on peut les récupérer un peu plus tard, ou bien ils sont archivés. A un moment donné ils peuvent être complètement effacés du système, y compris de la base et de l'archive.

Gérer le cycle de vie d'un objet du domaine constitue un défi en soi, et si ce n'est pas fait correctement, cela peut avoir un impact négatif sur le modèle du domaine. Nous allons présenter trois patterns qui nous aident à régler cela. Agrégat est un pattern de domaine utilisé pour définir l'appartenance et les frontières des objets. Les Fabriques

et les Entrepôts sont deux design patterns qui nous aident à traiter la création des objets et leur stockage. Nous allons commencer par parler des Agrégats.

Un modèle peut contenir un grand nombre d'objets du domaine. Aussi grande soit l'attention portée à la conception, il arrive que de nombreux objets soient associés entre eux, créant un réseau de relations complexe. Il y a plusieurs types d'associations. Pour chaque association traversable du modèle, il doit y avoir un mécanisme logiciel correspondant qui la met en application. Les véritables associations entre objets du domaine se retrouvent dans le code, et souvent même dans la base de données. Une relation un-à-un entre un client et le compte bancaire ouvert à son nom est exprimée sous la forme d'une référence entre deux objets, et engendre une relation entre deux tables en base, celle qui contient les clients et celle qui contient les comptes.

Bien souvent, le défi avec les modèles n'est pas de faire en sorte qu'ils soient suffisamment complets, mais de les rendre les plus simples et les plus compréhensibles possible. La plupart du temps, il s'avère payant d'éliminer ou de simplifier des relations du modèle. A moins bien sûr qu'elles ne recèlent une compréhension profonde du domaine.

Une association un-à-plusieurs est plus complexe car elle met en jeu de nombreux objets qui deviennent liés. Cette relation peut être simplifiée en la transformant en une association entre un objet et une collection d'autres objets, bien que ça ne soit pas toujours possible.

Il existe des associations plusieurs-à-plusieurs et un grand nombre d'entre elles sont bidirectionnelles. Cela augmente beaucoup la complexité, rendant la gestion du cycle de vie de ce genre d'objets assez difficile. Le nombre d'associations devrait être réduit autant que possible. En premier lieu, les associations qui ne sont pas essentielles pour le modèle devraient être retirées. Il se peut qu'elles existent dans le domaine, mais qu'elles ne soient pas nécessaires dans notre modèle, donc on peut les enlever. Deuxièmement, la multiplicité peut être réduite en ajoutant une contrainte. Si beaucoup d'objets satisfont à une relation, c'est possible qu'un seul puisse le faire si on impose la bonne contrainte sur la relation. Troisièmement, dans bien des cas les associations bidirectionnelles peuvent être transformées en associations unidirectionnelles. Chaque voiture a un moteur, et chaque moteur a une voiture dans laquelle il tourne. La relation est bidirectionnelle, mais elle peut facilement être simplifiée en considérant que la voiture a un moteur, et pas l'inverse.

Après avoir réduit et simplifié les associations entre objets, il se peut qu'on ait encore beaucoup de relations sur les bras. Un système bancaire détient et traite des données client. Ces données comprennent les informations personnelles du client, comme le nom, l'adresse, les numéros de téléphone, la description de l'emploi, et des informations sur le compte : numéro de compte, solde, opérations effectuées, etc.

Quand le système archive ou supprime complètement les informations sur un client, il doit s'assurer que toutes les références sont enlevées. Si beaucoup d'objets possèdent ces références, il est difficile d'être sûr qu'elles ont toutes été retirées. De plus, lorsque les données d'un client changent, le système doit vérifier qu'elles sont correctement mises à jour à travers tout le système, et que l'intégrité des données est garantie. On laisse généralement à la base de données le soin de s'en occuper. Les transactions sont utilisées pour assurer l'intégrité des données. Mais si le modèle n'a pas été soigneusement conçu, il y aura un fort degré de sollicitation de la base, conduisant à des performances médiocres. Même si les transactions de base de données jouent un rôle vital dans de telles opérations, il est souhaitable de résoudre certains problèmes liés à l'intégrité des données directement dans le modèle.

Il est aussi nécessaire de pouvoir faire respecter les invariants. Les invariants sont ces règles qui doivent être maintenues à chaque fois que les données changent. C'est dur à accomplir quand beaucoup d'objets ont des références vers les objets dont les données changent.

Il est difficile de garantir la cohérence des modifications apportées aux objets dans un modèle avec des associations complexes. Dans bien des cas, les invariants s'appliquent à des objets étroitement liés, pas juste à des objets discrets. Même des politiques de verrouillage prudentes peuvent amener plusieurs utilisateurs à interférer entre eux sans raison et rendre un système inutilisable.

Pour cette raison, utilisez des Agrégats. Un Agrégat est un groupe d'objets associés qui sont considérés comme un tout unique vis-à-vis des modifications de données. L'Agrégat est démarqué par une frontière qui sépare les objets situés à l'intérieur de ceux situés à l'extérieur. Chaque Agrégat a une racine. La racine est une Entité, et c'est le seul objet accessible de l'extérieur. La racine peut posséder des références vers n'importe quel objet de l'agrégat, et les autres objets peuvent se référencer entre eux, mais un objet externe peut seulement avoir une référence vers l'objet racine. S'il y a d'autres Entités à l'intérieur de la frontière, l'identité de ces entités est locale, elle fait sens seulement à l'intérieur de l'agrégat.

Comment l'Agrégat assure-t-il l'intégrité des données et fait-il respecter les invariants ? Puisque les objets externes peuvent seulement avoir des références vers la racine, ça veut dire qu'ils ne peuvent pas directement modifier les autres objets de l'agrégat. Tout ce qu'ils peuvent faire, c'est changer la racine, ou demander à la racine d'effectuer des actions. La racine sera capable de modifier les autres objets, mais c'est une opération contenue dans l'agrégat, et contrôlable. Si la racine est supprimée et retirée de la mémoire, tous les autres objets de l'agrégat seront supprimés aussi, car il n'y aura plus d'autre objet possédant une référence vers l'un d'entre eux. Lorsqu'un quelconque changement est apporté à la racine qui affecte indirectement les autres

objets de l'agrégat, il est facile d'assurer les invariants, car c'est la racine qui le fait. Ca serait beaucoup plus dur à faire si des objets externes avaient un accès direct aux objets internes et les modifiaient. Garantir les invariants dans cette circonstance impliquerait de mettre la logique adéquate dans des objets externes, ce qui n'est pas souhaitable.

La racine a la possibilité de passer des références éphémères d'objets internes à des objets externes, à la condition que les objets externes ne conservent pas la référence après que l'opération soit finie. Une illustration simple de cela consiste à passer des copies d'Objets Valeurs aux objets externes. Ce qui arrive à ces objets n'est pas vraiment important, parce que ça n'affectera pas l'intégrité de l'agrégat de toute façon.

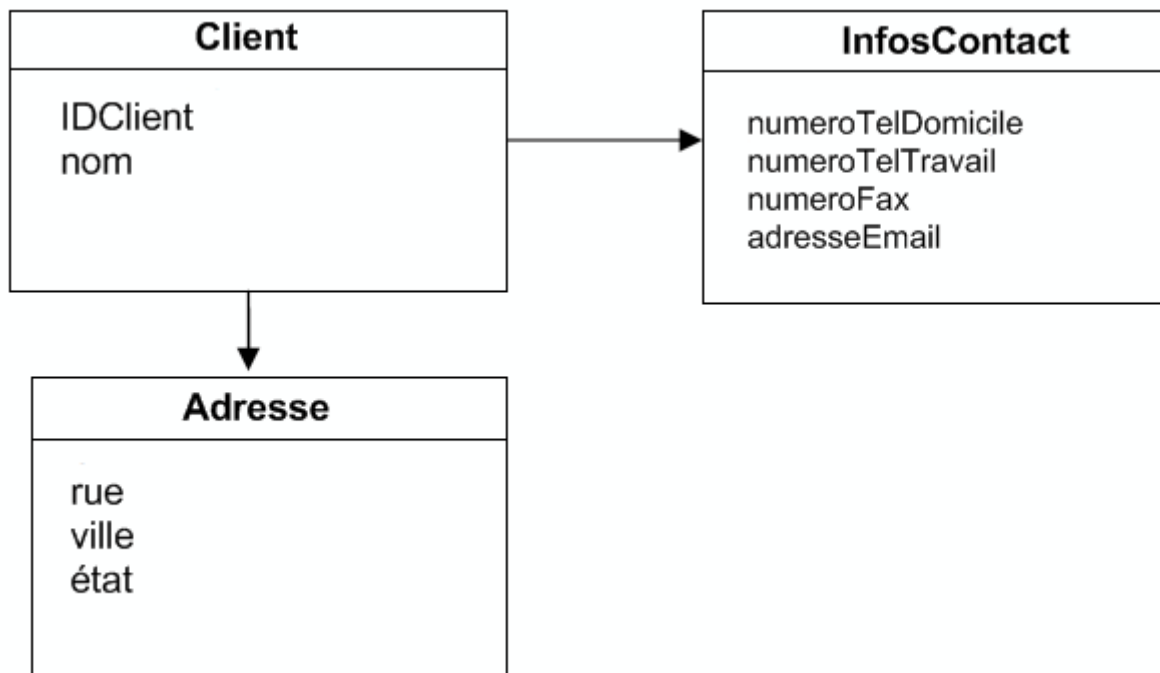
Si les objets d'un Agrégat sont stockés en base de données, seule la racine devrait pouvoir être obtenue par des requêtes. On devrait accéder aux autres objets à travers des associations traversables.

Les objets à l'intérieur d'un Agrégat doivent être autorisés à détenir des références vers les racines d'autres Agrégats.

L'Entité racine a une identité globale, et elle est responsable du maintien des invariants. Les Entités internes ont une identité locale.

Rassemblez les Entités et Objets Valeurs dans des Agrégats et définissez des frontières autour de chacun. Dans chaque Agrégat, choisissez une Entité qui sera la racine, et faites-lui contrôler l'accès aux objets situés à l'intérieur de la frontière. Vous ne devez autoriser les objets externes à détenir des références que vers la racine. Des références éphémères vers des membres internes peuvent être passées au-dehors uniquement dans le cadre d'une utilisation pour une opération ponctuelle. Puisque la racine contrôle l'accès, elle ne peut pas être prise de court par une modification inattendue des éléments internes. Grâce à cet arrangement, il devient pratique d'assurer les invariants des objets de l'Agrégat et de l'Agrégat dans sa globalité lors de tout changement d'état.

Un exemple simple d'Agrégation est présenté dans le diagramme qui suit. Le client est la racine de l'Agrégat, et tous les autres objets sont internes. Si on a besoin de l'Adresse, une copie de celle-ci peut être passée aux objets externes.



Les Fabriques

Les Entités et les Agrégats peuvent parfois être vastes et complexes – trop complexes pour être créés dans le constructeur de l’entité racine. En fait, essayer de construire un agrégat complexe dans son constructeur serait entrer en contradiction avec la manière dont ça se passe souvent dans le domaine lui-même, où des choses sont créées par d’autres choses (comme les composants électroniques sont créés par des lignes d’assemblage). Ce serait comme demander à une imprimante de se construire elle-même.

Lorsqu’un objet client veut créer un autre objet, il appelle son constructeur et lui passe éventuellement des paramètres. Mais quand la construction de l’objet est un processus laborieux, créer l’objet demande une grande connaissance de sa structure interne, des relations entre les autres objets qu’il contient, et des règles qui s’y appliquent. En d’autres termes, chaque client de l’objet va devoir détenir une connaissance spécifique de l’objet à construire. Ça rompt l’encapsulation des objets du domaine et des Agrégats. Si le client appartient à la couche application, une partie de la couche domaine a été déplacée ailleurs, ce qui détraque toute la conception. Dans la vie réelle, c’est comme si on nous donnait du plastique, du caoutchouc, du métal, du silicium et qu’on construisait notre propre imprimante. Ce n’est pas impossible à faire, mais cela en vaut-il réellement la peine ?

La création d’un objet a beau être une opération importante en soi, les opérations d’assemblage complexes ne sont pas de la responsabilité des objets créateurs.

Combiner de telles responsabilités peut produire des designs disgracieux qui sont difficiles à comprendre.

C'est pourquoi il est nécessaire d'introduire un nouveau concept qui aide à encapsuler le processus de création d'un objet complexe. C'est ce qu'on appelle une **Fabrique**. Les Fabriques sont utilisées pour encapsuler la connaissance nécessaire à la création des objets, et elles sont particulièrement utiles pour créer des Agrégats. Quand la racine d'un Agrégat est créée, tous les objets contenus dans l'Agrégat sont créés en même temps qu'elle, et tous les invariants sont appliqués.

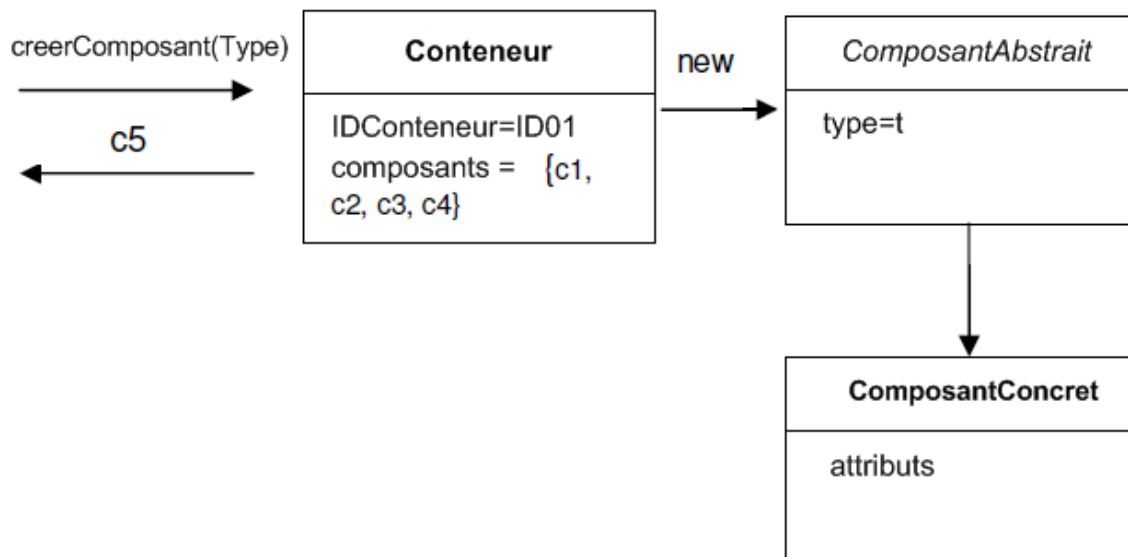
Il est important que le processus de création soit atomique. Si ce n'est pas le cas, il y a des chances pour que le processus soit à moitié terminé chez certains objets, les laissant dans un état indéterminé. C'est encore plus vrai pour les Agrégats. Lorsqu'on crée la racine, il est nécessaire de créer aussi tous les objets sujets à des invariants. Sinon, les invariants ne peuvent pas être appliqués. Pour des Objets Valeurs immuables, cela veut dire que tous les attributs sont initialisés dans un état valide. Si un objet ne peut pas être créé correctement, une exception devrait être levée, assurant qu'on ne retourne pas une valeur invalide.

Pour cette raison, faites basculer la responsabilité de créer des instances d'objets complexes et d'Agrégats dans un objet séparé, qui potentiellement n'a lui-même aucune responsabilité dans le modèle du domaine mais fait tout de même partie du design du domaine. Fournissez une interface qui encapsule tout l'assemblage complexe et ne nécessite pas que le client référence les classes concrètes des objets à instancier. Créez les Agrégats entiers de façon unitaire, en appliquant leurs invariants.

On utilise plusieurs design patterns pour implémenter les Fabriques. Le livre « Design Patterns » (Gamma et al.) les décrit en détail, et présente entre autres ces deux patterns : Méthode de fabrication, et Fabrique abstraite¹. Nous n'allons pas tenter de présenter ces patterns d'un point de vue conception, mais d'un point de vue modélisation du domaine.

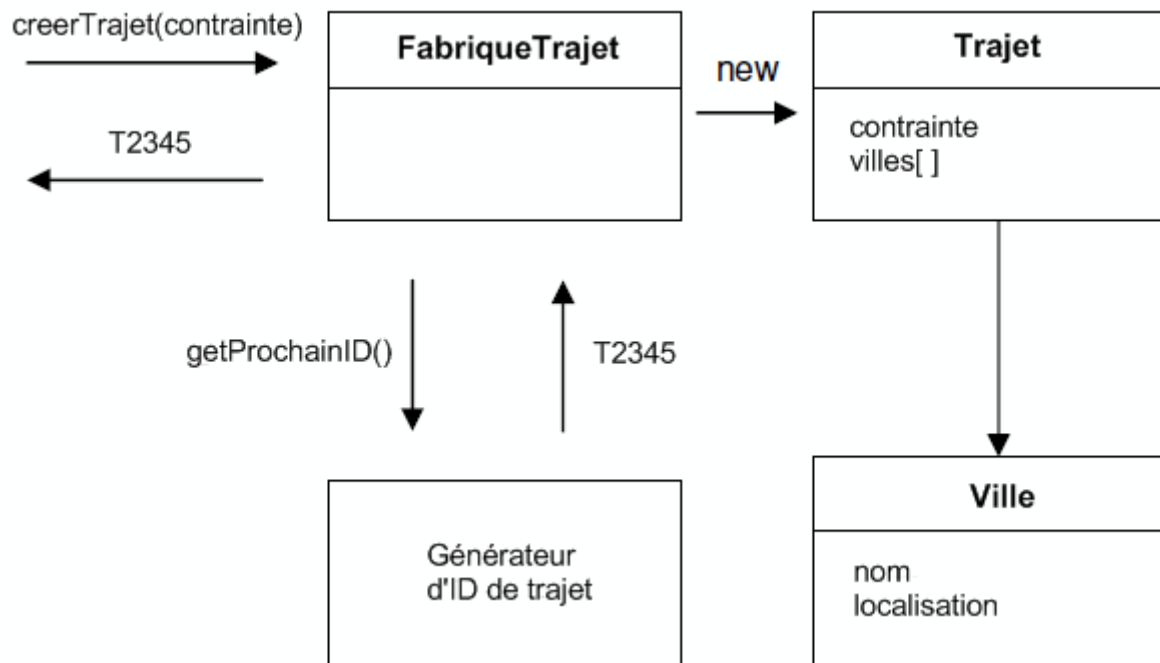
Une Méthode de fabrication est une méthode d'un objet qui contient et masque la connaissance nécessaire pour créer un autre objet. C'est très utile lorsqu'un client veut créer un objet qui appartient à un Agrégat. La solution est d'ajouter une méthode à la racine d'agrégat qui s'occupe de créer l'objet, de mettre en œuvre tous les invariants, et de retourner une référence vers cet objet ou vers une copie de cet objet.

¹ *Factory Method et Abstract Factory*



Le conteneur contient des composants qui sont d'un certain type. Quand un tel composant est créé, il est nécessaire qu'il appartienne automatiquement à un conteneur. Le client appelle la méthode `creerComposant(Type t)` du conteneur. Le conteneur instancie un nouveau composant. La classe concrète du composant est déterminée en se basant sur son type. Après sa création, le composant est ajouté à la collection de composants du conteneur, et une copie en est retournée au client.

Il y a des fois où la construction d'un objet est soit plus complexe, soit elle implique la création d'une série d'objets. Par exemple : la création d'un Agrégat. Cacher les besoins internes de création d'un Agrégat peut se faire dans un objet Fabrique séparé qui est dédié à cette tâche. Prenons l'exemple d'un module d'un programme qui calcule la route pouvant être suivie par une voiture d'un point de départ à une destination, étant donné une série de contraintes. L'utilisateur se connecte au site web qui exécute l'application et spécifie une des contraintes suivantes : trajet le plus court, trajet le plus rapide, trajet le moins cher. Les trajets créés peuvent être annotés avec des informations utilisateur qu'il faut sauvegarder, pour qu'on puisse les retrouver plus tard lorsque l'utilisateur se connectera à nouveau.



Le générateur d'ID de route est utilisé pour créer une identité unique pour chaque route, chose nécessaire pour une Entité.

Lorsqu'on crée une Fabrique, on est forcé de violer l'encapsulation d'un objet, ce qui doit être fait avec prudence. A chaque fois que quelque chose change dans l'objet qui a un impact sur les règles de construction ou sur certains des invariants, nous devons nous assurer que le code de la Fabrique est mis à jour pour supporter les nouvelles conditions. Les Fabriques sont profondément liées aux objets qu'elles créent. Ca peut être une faiblesse, mais ça peut aussi être une force. Un Agrégat contient une série d'objets étroitement reliés entre eux. La construction de la racine est associée à la création des autres objets de l'Agrégat, et il doit y avoir une logique pour constituer un Agrégat. Cette logique n'appartient pas naturellement à un des objets, parce qu'elle concerne la construction d'autres objets. Il semble adapté d'utiliser une classe Fabrique spéciale à qui on confie la tâche de créer l'Agrégat tout entier, et qui contiendra les règles, les contraintes et les invariants devant être appliqués pour que l'Agrégat soit valide. Les objets resteront simples et serviront leur propre but sans être gênés par le fouillis d'une logique de construction complexe.

Les Fabriques d'Entités sont différentes des Fabriques d'Objets Valeurs. Les Objets Valeurs sont généralement immuables, et tous les attributs nécessaires doivent être produits au moment de leur création. Lorsque l'objet est créé, il doit être valide et finalisé. Il ne changera pas. Les Entités ne sont pas immuables. Elles peuvent être changées plus tard en modifiant des attributs, en ajoutant que tous les invariants doivent être respectés. Une autre différence vient du fait que les Entités ont besoin d'une identité alors que les Objets Valeurs, non.

Certaines fois, on n'a pas besoin d'une Fabrique, un simple constructeur suffit. Utilisez un constructeur lorsque :

- La construction n'est pas compliquée.
- La création d'un objet n'implique pas la création d'autres, et tous les attributs requis sont passés via le constructeur.
- Le client s'intéresse à l'implémentation, et potentiellement il veut choisir la Stratégie à utiliser.
- La classe est le type. Il n'y a pas de hiérarchie en jeu, donc pas besoin de choisir parmi une liste d'implémentations concrètes.

Une autre observation est que les Fabriques doivent soit créer de nouveaux objets à partir de zéro, soit on leur demande de reconstituer des objets qui existaient précédemment, mais qui ont probablement été persistés dans une base de données. Ramener des Entités en mémoire depuis leur lieu stockage en base met en jeu un processus complètement différent de la création de nouvelles Entités. Une différence évidente est que l'objet existant n'a pas besoin d'une nouvelle identité. L'objet en a déjà une. Les violations des invariants sont traitées différemment. Quand un nouvel objet est créé à partir de rien, toute violation d'invariant finit en exception. On ne peut pas faire ça avec des objets recréés depuis une base. Les objets doivent être réparés d'une manière ou d'une autre pour qu'ils puissent être fonctionnels, sinon il y a perte de données.

Les Entrepôts

Dans une conception dirigée par le modèle, les objets ont un cycle de vie qui commence par leur création et se termine avec leur suppression ou leur archivage. Un constructeur ou une Fabrique se charge de la création de l'objet. Tout l'objectif de créer des objets est de pouvoir les utiliser. Dans un langage orienté objet, on doit posséder une référence vers un objet pour être capable de s'en servir. Pour avoir cette référence, le client doit soit créer l'objet, soit l'obtenir d'un autre, en traversant une association existante. Par exemple, pour obtenir un Objet-Valeur d'un Agrégat, le client doit le demander à la racine de l'Agrégat. L'ennui, c'est que maintenant le client a une référence vers la racine. Dans de grosses applications, ça peut être un problème parce qu'on doit s'assurer que le client a toujours une référence vers l'objet dont il a besoin, ou vers un autre objet qui lui, a cette référence. Utiliser ce genre de règle dans la conception va forcer les objets à conserver une série de références qu'ils ne garderaient probablement pas autrement. Cela augmente le couplage, créant une série d'associations dont on n'a pas vraiment besoin.

Utiliser un objet implique que l'objet ait déjà été créé. Si l'objet est la racine d'un Agrégat, alors c'est une Entité, et il y a des chances pour qu'elle soit stockée dans un état persistant dans une base de données ou une autre forme de stockage. Si c'est un Objet-Valeur, il peut être obtenu auprès d'une Entité en traversant une association, mais il s'avère que bon nombre d'objets peuvent être directement récupérés dans une base de données. Cela résout le problème de l'obtention de références d'objets. Quand un client veut utiliser un objet, il accède à la base, y retrouve l'objet et s'en sert. Cela semble une solution simple et rapide, mais ça a des impacts négatifs sur le design.

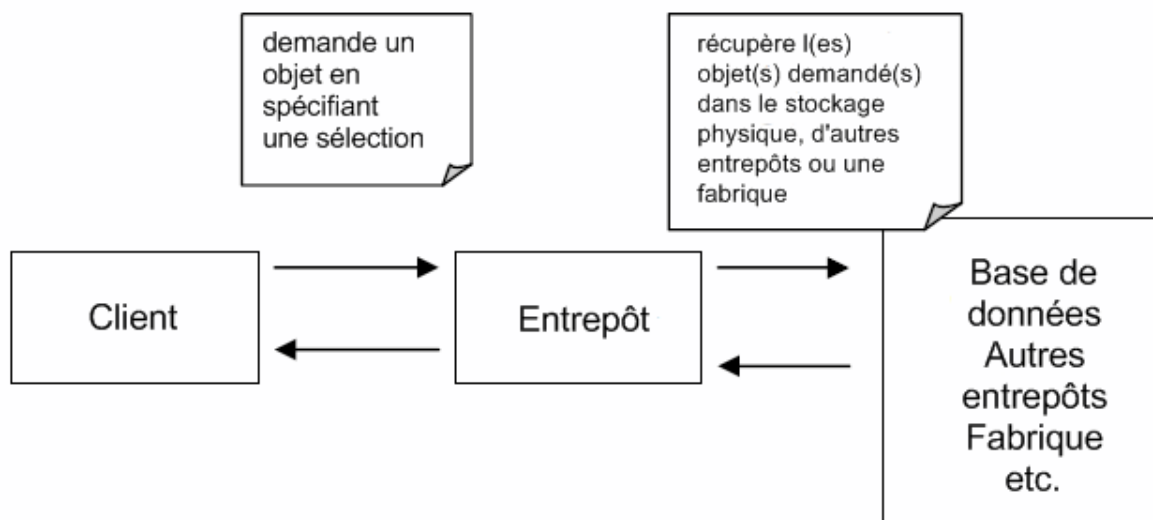
Les bases de données font partie de l'infrastructure. Une solution médiocre serait que le client connaisse les détails requis pour accéder à une base. Par exemple, le client devrait créer des requêtes SQL pour récupérer les données souhaitées. La requête de base de données retournerait un jeu d'enregistrements, exposant encore plus de ses détails internes. Quand beaucoup de clients doivent créer des objets directement depuis la base de données, ce genre de code se retrouve éparpillé un peu partout dans le domaine. A cet instant le modèle du domaine est compromis. Il doit traiter de nombreux détails d'infrastructure au lieu de s'occuper des concepts du domaine. Que se passe-t-il si la décision est prise de changer la base de données sous-jacente ? Tout ce code dispersé doit être modifié pour qu'on puisse accéder au nouvel emplacement de stockage. Lorsque du code client accède à une base directement, il est possible qu'il restaure un objet interne à un Agrégat. Cela rompt l'encapsulation de l'Agrégat, avec des conséquences inconnues.

Un client a besoin d'un moyen concret d'acquérir des références vers des objets du domaine préexistants. Si l'infrastructure rend cela facile, il est probable que les développeurs du code client ajouteront plus d'associations traversables et embrouillent le modèle. Ou alors, il se peut qu'ils utilisent des requêtes pour puiser directement dans la base les données exactes dont ils ont besoin, ou pour piocher quelques objets spécifiques plutôt que de naviguer à partir des racines d'agrégats. La logique du domaine est déplacée dans les requêtes et le code client, et les Entités et Objets Valeurs deviennent de simples conteneurs de données. La complexité technique brute de l'implémentation de l'infrastructure d'accès à la base submerge rapidement le code client, ce qui amène les développeurs à abêtir la couche domaine, et rend le modèle obsolète. L'effet général est qu'on perd la focalisation sur le domaine et qu'on compromet le design.

Pour cette raison, utilisez un Entrepôt, dont le but est d'encapsuler toute la logique nécessaire à l'obtention de références d'objets. Les objets du domaine n'auront pas à s'occuper de l'infrastructure de récupération des références aux autres objets du domaine dont ils ont besoin. Ils iront simplement les chercher dans l'Entrepôt et le modèle retrouvera sa clarté et sa focalisation.

L'Entrepôt est capable de conserver des références vers des objets. Quand un objet est créé, il peut être sauvegardé dans l'Entrepôt, et y être récupéré plus tard quand on voudra l'utiliser. Si le client demande un objet à l'Entrepôt, et que l'Entrepôt ne l'a pas, ce dernier peut aller le chercher dans l'emplacement de stockage physique. Dans tous les cas, l'Entrepôt agit comme un endroit où sont emmagasinés des objets globalement accessibles.

L'Entrepôt peut aussi comporter une Stratégie. Il peut accéder à un stockage persistant ou à un autre en fonction de la Stratégie précisée. Il se peut qu'il utilise différents emplacements de stockage pour différents types d'objets. Cela a pour effet global de découpler le modèle du domaine du besoin de stocker des objets ou leurs références et de celui d'accéder à l'infrastructure de persistance sous-jacente.

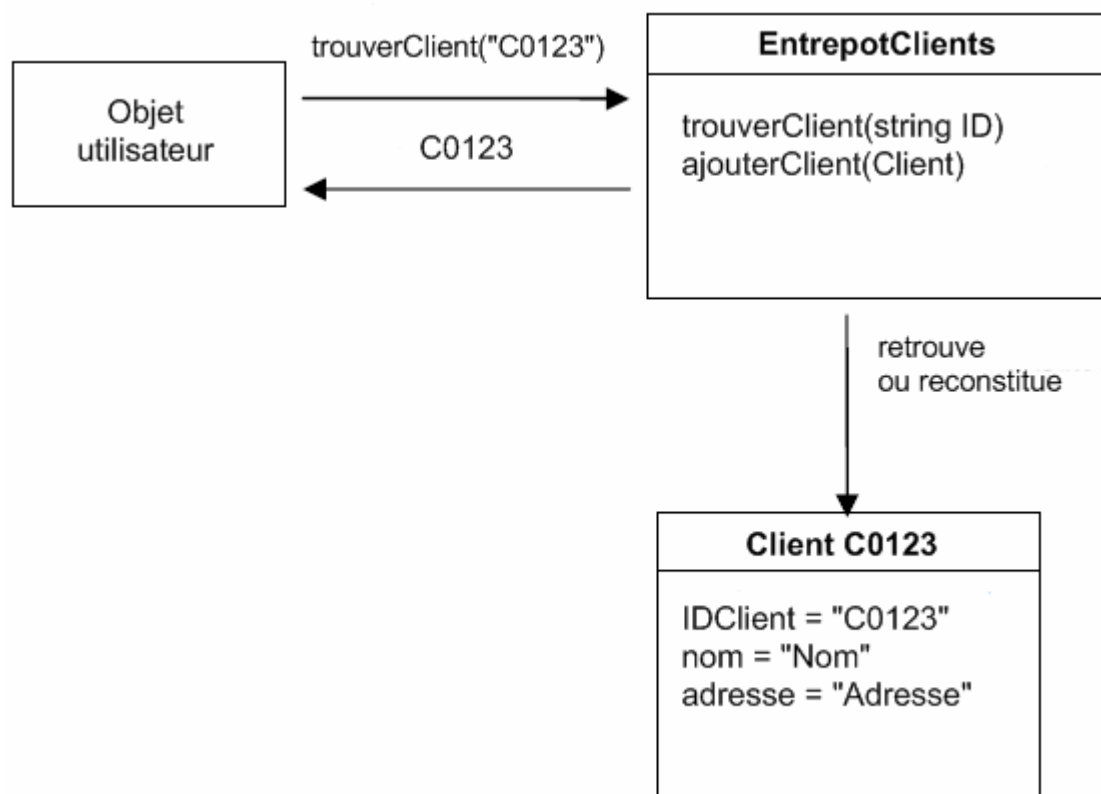


Pour chaque type d'objet qui nécessite un accès global, créez un objet qui puisse procurer l'illusion d'une collection en mémoire de tous les objets de ce type. Organisez-en l'accès à travers une interface globale connue de tous. Fournissez des méthodes pour ajouter et enlever des objets, qui encapsuleront les véritables insertions et suppressions de données dans le stockage des données. Proposez des méthodes qui sélectionnent des objets en se basant sur des critères et qui retournent des objets ou collections d'objets pleinement instanciés dont les valeurs satisfont les critères ; ce faisant, vous encapsulerez la technologie réelle de stockage et de requêtage. Fournissez des Entrepôts seulement pour les racines d'agrégats qui nécessitent réellement un accès direct. Conservez un client focalisé sur le modèle, en déléguant tout stockage ou accès aux données aux Entrepôts.

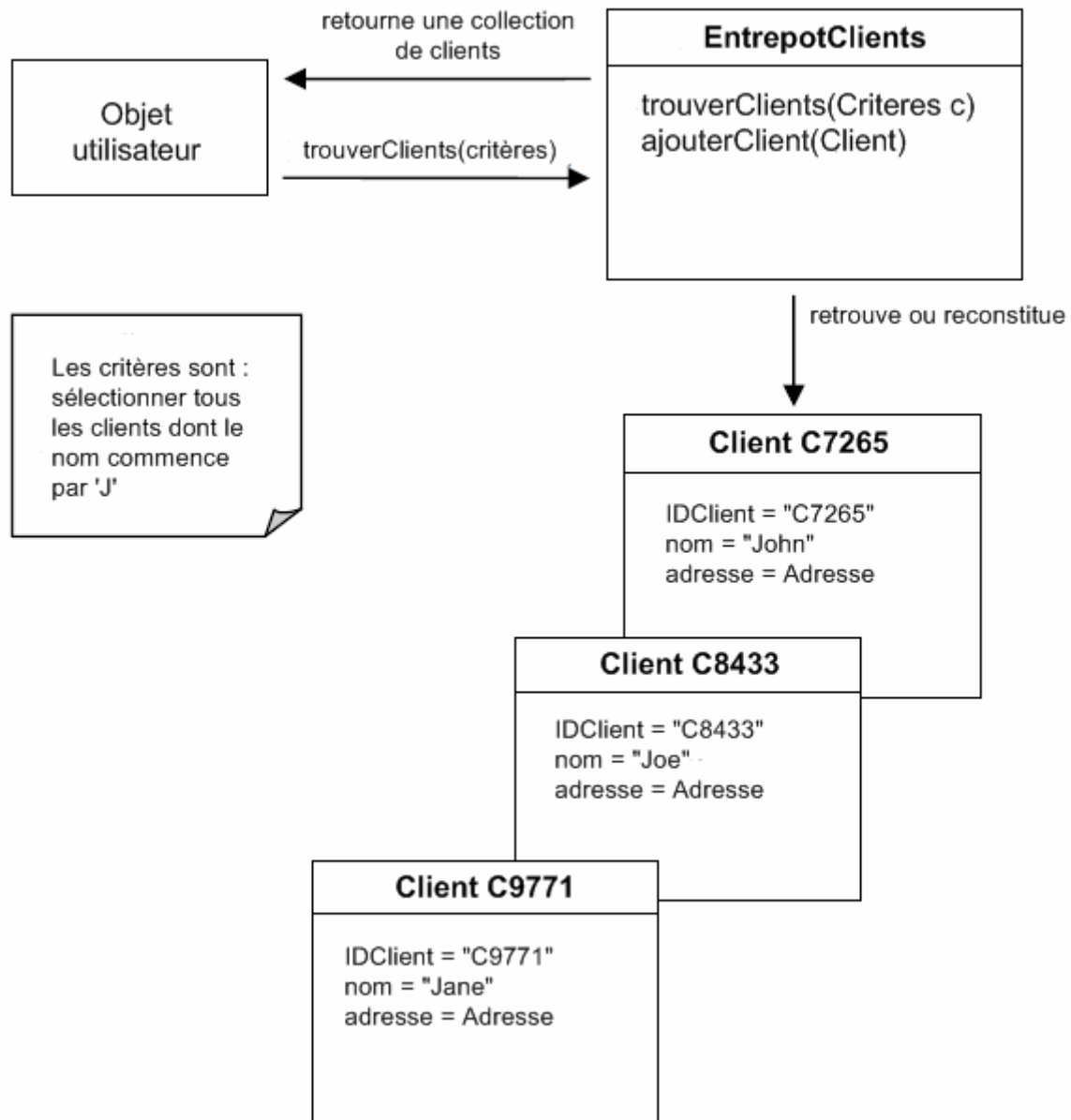
Un Entrepôt peut contenir des informations détaillées dont il se sert pour accéder à l'infrastructure, mais son interface devrait être simple. Un Entrepôt doit avoir un jeu de méthodes utilisées pour récupérer des objets. Le client appelle une de ces méthodes

et lui passe un ou plusieurs paramètres qui représentent les critères de sélection utilisés pour choisir un objet ou un ensemble d'objets correspondants. Une Entité peut facilement être demandée en passant son identité. D'autres critères de sélection peuvent être constitués d'un ensemble d'attributs de l'objet. L'Entrepôt va comparer tous les objets à cet ensemble et retournera ceux qui satisfont aux critères. L'interface de l'Entrepôt pourrait aussi contenir des méthodes utilisées pour effectuer des calculs supplémentaires comme le nombre d'objets d'un certain type.

On notera que l'implémentation d'un entrepôt peut être étroitement assimilée à de l'infrastructure, mais que l'interface de l'entrepôt est du pur modèle du domaine.

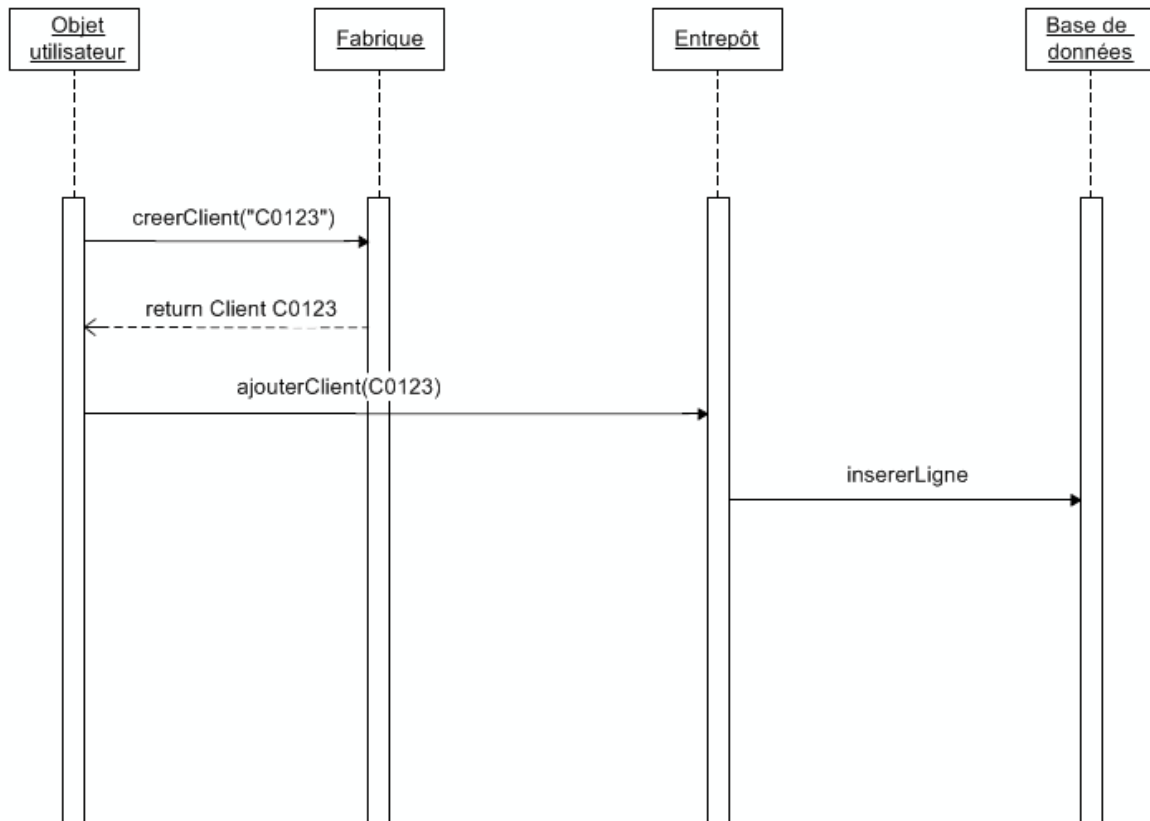


Une autre option est de préciser des critères de sélection sous forme de Spécification. La Spécification permet de définir des critères plus complexes, comme dans l'exemple suivant :



Il y a une relation entre Fabrique et Entrepôt. Ce sont tous deux des patterns de conception dirigée par le modèle, et ils nous aident tous les deux à gérer le cycle de vie des objets du domaine. Tandis que la Fabrique est concernée par la création d'objets, l'Entrepôt se charge des objets déjà existants. L'Entrepôt peut mettre en cache des objets localement, mais très souvent il a besoin de les récupérer dans un lieu de stockage persistant. Les objets sont soit créés en utilisant un constructeur, soit on les passe à une Fabrique pour qu'ils y soient construits. C'est pour cette raison que l'Entrepôt peut être vu comme une Fabrique, car il crée des objets. Ce n'est pas une création *ex nihilo*, mais la reconstitution d'un objet qui a existé. On ne doit pas mélanger Entrepôt et Fabrique. La fabrique crée de nouveaux objets, tandis que l'Entrepôt retrouve des objets déjà créés. Quand un nouvel objet doit être ajouté à

l'Entrepôt, il devrait d'abord être créé par la Fabrique, et ensuite donné à l'Entrepôt qui le stockera comme dans l'exemple ci-dessous.



Une autre façon de qualifier cela est de dire que les Fabriques sont « purement domaine » alors que les Entrepôts peuvent contenir des liens avec l'infrastructure, par exemple la base de données.

4

Refactorer pour une vision plus profonde

Refactorer en continu

Jusqu'ici nous avons parlé du domaine, et de l'importance de créer un modèle qui exprime le domaine. Nous avons donné quelques conseils sur les techniques à utiliser pour créer un modèle utile. Le modèle doit être étroitement associé au domaine d'où il provient. Nous avons aussi dit que le design du code doit être construit autour du modèle, et que le modèle lui-même devait être amélioré à partir des décisions de design. Concevoir sans modèle peut mener à un logiciel qui ne sera pas fidèle au domaine qu'il sert, et n'aura peut-être pas le comportement attendu. A l'inverse, modéliser sans le feedback de la conception et sans impliquer les développeurs nous conduira vers un modèle qui n'est pas bien compris par ceux qui doivent l'implémenter, et peut ne pas être adapté aux technologies utilisées.

Pendant le processus de conception et de développement, nous devons nous arrêter de temps en temps et jeter un œil sur le code. Peut-être que c'est alors le bon moment pour un refactoring. Le refactoring est le processus de reconception du code en vue de l'améliorer sans changer le comportement de l'application. Le refactoring se déroule généralement en petites étapes contrôlables et précautionneuses, pour ne pas casser des fonctionnalités ou introduire des bugs. Après tout, l'objectif du refactoring est de parfaire le code, pas de le détériorer. Les tests automatisés sont d'une grande aide pour s'assurer que nous n'avons rien endommagé.

Il y a de nombreuses manières de refactorer du code. Il y a même des patterns de refactoring. Ces patterns constituent une approche automatisée du refactoring. Il existe des outils basés sur ces patterns, qui rendent la vie du développeur bien plus facile

qu'elle n'était. Sans ces outils, il peut être très difficile de refactorer. Ce genre de refactoring concerne plus le code et sa qualité.

Il existe un autre type de refactoring, qui est lié au domaine et à son modèle. Parfois on a de nouvelles idées sur le domaine, quelque chose devient plus clair, ou on découvre une relation entre deux éléments. Tout ceci doit être inclus dans la conception via du refactoring. Il est très important d'avoir un code expressif, facile à lire et à comprendre. A la lecture du code, on devrait être capable de dire ce qu'il fait, mais aussi pourquoi il le fait. C'est à cette seule condition que le code peut véritablement capturer la substance du modèle.

Le refactoring technique, celui basé sur des patterns, peut être organisé et structuré. Un refactoring pour une vision plus profonde ne peut pas s'effectuer de la même manière. On ne peut pas créer des patterns pour ça. La complexité et la variété des modèles ne nous offrent pas la possibilité d'aborder la modélisation de façon mécanique. Un bon modèle est le produit d'une réflexion profonde, de la perspicacité, de l'expérience, et du flair.

Une des premières choses qu'on nous apprend sur la modélisation, c'est de lire les spécifications fonctionnelles et de chercher les noms et les verbes. Les noms sont convertis en classes, tandis que les verbes deviennent des méthodes. C'est une simplification, et cela mène à un modèle superficiel. Tous les modèles manquent de profondeur au début, et nous devrions les refactorer en vue d'une compréhension toujours plus fine.

La conception doit être flexible. Un design rigide résiste au refactoring. Du code qui n'a pas été construit dans un esprit de flexibilité, c'est du code avec lequel il est difficile de travailler. A chaque besoin de changement, vous allez devoir vous battre avec le code, et les choses à refactorer prendront facilement beaucoup de temps.

Utiliser un ensemble éprouvé de blocs de construction de base ainsi qu'un langage cohérent assainit déjà l'effort de développement. Ça nous laisse avec le défi de trouver un modèle incisif, un modèle qui capture les préoccupations subtiles des experts du domaine et peut guider une conception pragmatique. Un modèle qui se débarrasse du superficiel et capture l'essentiel est un modèle profond. Cela devrait mettre le logiciel davantage au diapason du mode de pensée des experts du domaine, et le rendre plus réceptif aux besoins des utilisateurs.

Traditionnellement, le refactoring est décrit en termes de transformations de code avec des motivations techniques. Le refactoring peut aussi être motivé par une avancée nouvelle dans le domaine et le raffinement correspondant du modèle ou de son expression dans le code.

Les modèles du domaine sophistiqués sont rarement développés autrement que par un processus itératif de refactoring, qui suppose une implication étroite des experts métier et des développeurs intéressés par l'apprentissage du domaine.

Mettre au jour les concepts clés

Le refactoring se fait par petites étapes. Aussi, le résultat est une série de petites améliorations. Il y a des fois où de nombreux petits changements ajoutent très peu de valeur au design, et d'autres où quelques changements font une grande différence. C'est ce qu'on appelle une Avancée majeure.

Au début, on a un modèle superficiel et grossier. Ensuite nous le raffinons, lui et le design, en nous basant sur une connaissance plus profonde du domaine, sur une meilleure compréhension de ses enjeux. On y ajoute des abstractions. Puis la conception est refactorisée. Chaque affinage ajoute de la clarté au design. Cela crée ensuite les conditions pour une Avancée majeure.

Une Avancée majeure implique souvent un changement dans la façon dont nous pensons et voyons le modèle. C'est une source de grand progrès dans le projet, mais cela a aussi ses inconvénients. Une Avancée majeure peut impliquer une grande quantité de refactoring. Cela signifie du temps et des ressources, choses dont, semble-t-il, nous manquons toujours. C'est également risqué, car un vaste refactoring peut introduire des changements dans le comportement de l'application.

Pour réaliser une Avancée majeure, il nous faut rendre explicites les concepts implicites. Quand on parle aux experts du domaine, on échange beaucoup d'idées et de connaissances. Certains concepts se frayent un chemin jusqu'au Langage omniprésent, mais d'autres passent inaperçus dans un premier temps. Ce sont des concepts implicites, utilisés pour expliquer d'autres concepts déjà présents dans le modèle. Pendant le processus d'affinage de la conception, quelques-uns de ces concepts implicites attirent notre attention. On découvre que certains d'entre eux jouent un rôle clé dans la conception. C'est à ce moment-là que nous devrions rendre ces concepts explicites. On devrait leur créer des classes et des relations. Quand ça se produit, il se peut qu'on ait la chance d'être face à une Avancée majeure.

Les concepts implicites ne devraient pas le rester. Si ce sont des concepts du domaine, ils devraient être présents dans le modèle et dans la conception. Comment les reconnaître ? La première façon de découvrir des concepts implicites, c'est d'écouter

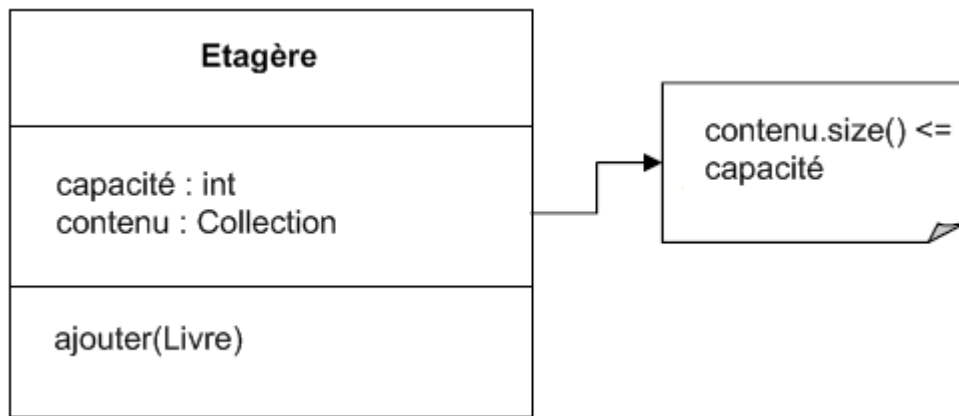
le langage. Le langage qu'on utilise pendant la modélisation et la conception contient beaucoup d'informations sur le domaine. Au début il peut ne pas y en avoir tant que ça, ou une partie de l'information peut ne pas être utilisée correctement. Il se peut que certains des concepts ne soient pas pleinement compris, ou même qu'on les ait totalement mal compris. Tout ça fait partie de l'apprentissage d'un nouveau domaine. Mais à mesure que nous construisons notre Langage omniprésent, les concepts clés finissent par s'y intégrer. C'est là qu'on doit commencer à chercher des concepts implicites.

Parfois, il y a des parties du design qui ne sont pas si claires que ça. Un ensemble de relations rend le cheminement des traitements dur à suivre. Ou bien les procédures font quelque chose de complexe et de difficilement compréhensible. Il y a des maladresses dans le design. C'est un bon endroit où chercher des concepts cachés. Probablement que quelque chose manque. Si un concept clé manque au puzzle, les autres vont devoir remplacer ses fonctionnalités. Cela va dilater certains objets, leur ajoutant un comportement qui n'est pas censé s'y trouver. C'est la clarté du design qui va en souffrir. Essayez de voir s'il y a un concept manquant et si vous en trouvez un, rendez-le explicite. Refactorisez le design pour le rendre plus simple et plus souple.

Lorsqu'on bâtit la connaissance, il est possible qu'on se heurte à des contradictions. Ce que dit un expert du domaine peut sembler aller à l'encontre de ce qu'un deuxième soutient. Une spécification peut sembler en contredire une autre. Certaines contradictions n'en sont pas vraiment, ce sont plutôt des manières différentes de voir la même chose, ou simplement un manque de précision dans les explications. Nous devons essayer de réconcilier les contradictions. Parfois, ça met au jour des concepts importants. Même si ce n'est pas le cas, c'est tout de même important de le faire pour maintenir un modèle bien clair.

Une autre façon évidente de dénicher des concepts du domaine est d'utiliser la littérature du domaine. Il existe des livres sur à peu près n'importe quel sujet. Ils renferment des tas de connaissances sur leurs domaines respectifs. Généralement, les livres ne contiennent pas de modèle des domaines qu'ils présentent : l'information qui s'y trouve doit être traitée, distillée et raffinée. Néanmoins, cette information est de grande valeur et offre une vue approfondie du domaine.

Il y a d'autres concepts qui sont très utiles lorsqu'on les rend explicites : Contrainte, Processus et Spécification. Une Contrainte est une manière simple d'exprimer un invariant. Quoi qu'il arrive aux données des objets, l'invariant est respecté. On peut faire ça simplement en mettant la logique de l'invariant dans une Contrainte. Ce qui suit en est un exemple simple. Son but est d'expliquer le concept, pas de présenter l'approche préconisée dans ce cas nominal.



On peut ajouter des livres à une étagère, mais on ne devrait jamais pouvoir en ajouter plus que sa capacité. On peut voir ça comme faisant partie du comportement de l'Etagère, comme dans le code Java qui suit.

```
public class Etagere {
    private int capacite = 20;
    private Collection contenu;
    public void ajouter(Livre livre) {
        if(contenu.size() + 1 <= capacite) {
            contenu.add(livre);
        } else {
            throw new IllegalArgumentException(
                "L'étagère a atteint sa limite.");
        }
    }
}
```

Nous pouvons refactorer le code en extrayant la contrainte dans une méthode séparée.

```
public class Etagere {
    private int capacite = 20;
    private Collection contenu;
    public void ajouter(Livre livre) {
        if(espaceEstDisponible()) {
            contenu.add(livre);
        } else {
            throw new IllegalArgumentException(
                "L'étagère a atteint sa limite.");
        }
    }
    private boolean espaceEstDisponible() {
        return contenu.size() < capacite;
    }
}
```

}

Placer la contrainte dans une méthode séparée a l'avantage de la rendre explicite. C'est facile à lire et tout le monde remarquera que la méthode `ajouter()` est sujette à cette contrainte. Il y aura aussi de la place pour une évolution lorsqu'on voudra ajouter plus de logique aux méthodes, si la contrainte devient plus complexe.

Les Processus sont généralement exprimés dans du code à travers des procédures. Nous n'allons pas utiliser une approche procédurale, puisque nous utilisons un langage orienté objet ; il nous faut donc choisir un objet pour le processus, et y ajouter un comportement. La meilleure manière d'implémenter les processus est d'utiliser un Service. S'il y a différentes manières de mener à bien le processus, alors nous pouvons encapsuler l'algorithme dans un objet et utiliser une Stratégie. Tous les processus ne sont pas destinés à être rendus explicites. Le bon moment pour implémenter explicitement un processus, c'est quand le Langage omniprésent le mentionne expressément.

La dernière méthode pour rendre les concepts explicites dont nous traiterons ici est la Spécification. Pour faire simple, on utilise une Spécification pour tester un objet afin de voir s'il satisfait certains critères.

La couche domaine contient des règles métier qui s'appliquent aux Entités et aux Objets Valeurs. Ces règles sont classiquement intégrées dans les objets auxquels elles s'appliquent. Certaines d'entre elles sont juste un ensemble de questions dont la réponse est « oui » ou « non ». Ce genre de règle peut être exprimé par une série d'opérations logiques appliquées sur des valeurs booléennes, et le résultat final est aussi un booléen. Un exemple de ça, c'est le test effectué sur un objet Client pour voir s'il est éligible à un certain crédit. La règle peut être exprimée sous forme de méthode appelée `estEligible()`, et rattachée à l'objet Client. Mais cette règle n'est pas une simple méthode qui opère strictement sur des données du Client. L'évaluation de la règle implique de vérifier les références du client, de voir s'il a payé ses dettes dans le passé, d'examiner s'il a des soldes négatifs, etc. Ces règles métier peuvent être grosses et complexes et faire gonfler l'objet au point qu'il ne serve plus son objectif d'origine. A ce moment on pourrait être tenté de déplacer l'intégralité de la règle au niveau application, car elle semble s'étendre au-delà du niveau domaine. En fait, il est temps de refactorer.

La règle doit être encapsulée dans un objet séparé qui devient la Spécification du Client, et qu'on doit laisser dans la couche domaine. Le nouvel objet va contenir une série de méthodes booléennes qui testent si un objet Client particulier est éligible pour un crédit ou pas. Chaque méthode joue le rôle d'un petit test, et toutes les méthodes combinées donnent la réponse à la question d'origine. Si la règle métier n'est pas

comprise dans un unique objet Spécification, le code qui correspond finira éparpillé dans bon nombre d'objets, ce qui le rendra incohérent.

La Spécification est utilisée pour tester des objets pour voir s'ils répondent à un besoin, ou s'ils sont prêts à remplir un objectif. On peut aussi s'en servir pour sélectionner un objet particulier dans une collection, ou comme condition durant la création d'un objet.

Souvent, il y a une Spécification séparée pour vérifier que chaque règle simple est satisfaite, et ensuite on combine un certain nombre d'entre elles dans une spécification composite qui exprime la règle complexe, comme ceci :

```
Client client =
entrepotClients.trouverClient(identiteClient);
...
Specification clientEligiblePourRemboursement = new
Specification(
    new clientAPayeSesDettesDansLePasse(),
    new clientNAPasDeSoldesNegatifs());
if(clientEligiblePourRemboursement.estSatisfaitePar(client)
{
    serviceRemboursement.envoyerRemboursementA(client);
}
```

Il est plus simple de tester des règles élémentaires, et juste en regardant ce code, la définition d'un client éligible à un remboursement devient évidente.

5

Préserver l'intégrité du modèle

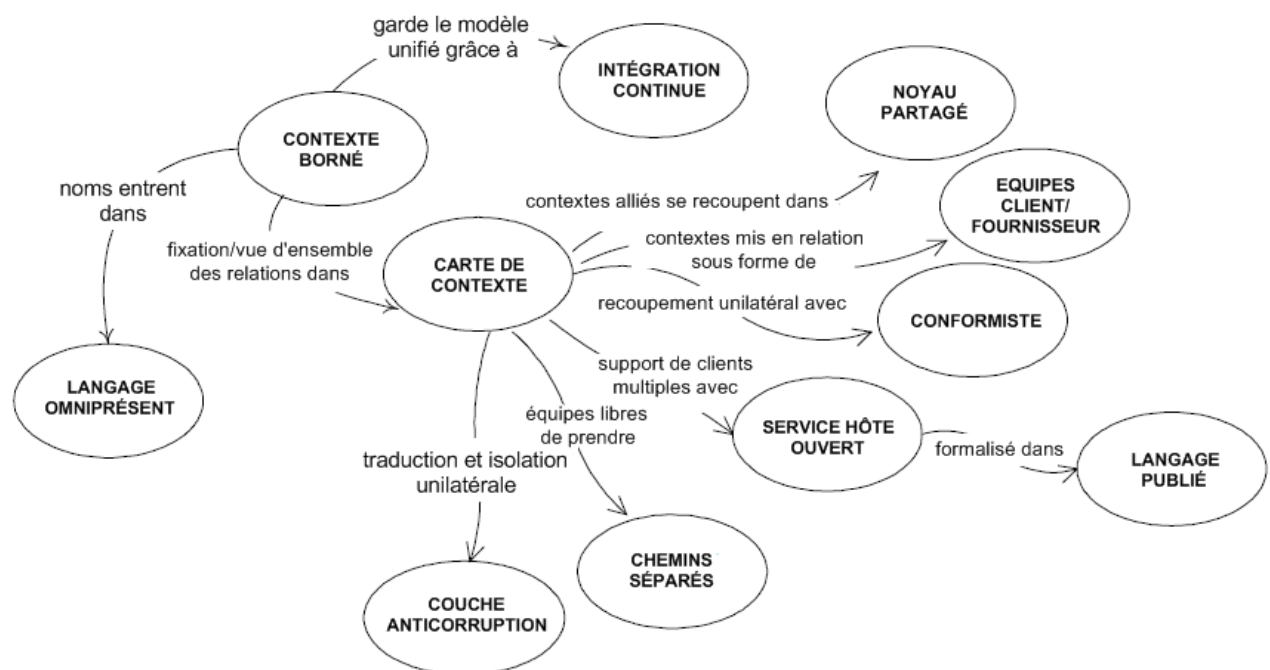
Ce chapitre traite des gros projets qui nécessitent les efforts combinés de multiples équipes. On se retrouve face à un nouvel ensemble de défis lorsqu'on confie à plusieurs équipes, dans des conditions de management et de coordination diverses, la tâche de développer un projet. Les projets d'entreprise sont généralement vastes, ils emploient des technologies et des ressources variées. La conception de tels projets devrait tout de même être basée sur un modèle du domaine, et on doit prendre les mesures qui correspondent pour s'assurer de la réussite du projet.

Quand plusieurs équipes travaillent sur un projet, le développement du code est fait en parallèle, chaque équipe se voyant assigner une partie spécifique du modèle. Ces parties ne sont pas indépendantes, mais plus ou moins interconnectées. On part d'un seul gros modèle, et tout le monde reçoit une partie de celui-ci à implémenter. Supposons qu'une équipe a créé un module, et le rend disponible à l'usage des autres équipes. Un développeur d'une autre équipe commence à utiliser le module, et découvre qu'il manque à ce dernier une fonctionnalité requise par son propre module. Il ajoute cette fonctionnalité et archive le code pour qu'il puisse être utilisé par tous. Ce qu'il n'a peut-être pas réalisé, c'est que ça constitue en réalité un changement du modèle, et il est tout à fait possible que ce changement casse des fonctionnalités de l'application. Ça peut arriver d'autant plus facilement que personne ne prend tout à fait le temps de comprendre le modèle en entier. Chacun maîtrise son propre pré carré, mais ne connaît pas les autres zones suffisamment en détail.

Il suffit de pas grand-chose pour partir d'un bon modèle et le faire progresser vers un modèle incohérent. La première nécessité pour un modèle est qu'il soit cohérent, avec des termes fixés et aucune contradiction. La cohérence interne d'un modèle est appelée *unification*. Un projet d'entreprise peut avoir un modèle qui recouvre l'intégralité du domaine de l'entreprise, sans contradictions et sans termes qui se chevauchent. Le modèle d'entreprise unifié est un idéal difficile à atteindre, et parfois ça ne vaut même pas la peine d'essayer. Ce genre de projet requiert les efforts combinés de nombreuses équipes. Les équipes ont besoin d'un degré élevé d'indépendance dans le processus de développement, parce qu'elles n'ont pas le temps de se rencontrer et de débattre de la

conception constamment. La coordination de telles équipes est une tâche épouvantable. Il se peut qu'elles appartiennent à des services différents et qu'elles aient des managements séparés. Lorsque le design du modèle évolue en partie indépendamment, on doit faire face à l'éventualité d'une perte d'intégrité du modèle. Si on essaye de préserver l'intégrité du modèle en maintenant un gros modèle unifié pour tout le projet d'entreprise, ça ne marchera pas. La solution n'est pas si évidente, parce qu'elle est le contraire de tout ce que nous avons appris jusqu'ici. Au lieu d'essayer de garder un gros modèle qui va s'effondrer plus tard, nous devrions le diviser sciemment en plusieurs modèles. Plusieurs modèles bien intégrés peuvent évoluer indépendamment tant qu'ils obéissent au contrat auquel ils sont liés. Chaque modèle doit avoir une frontière clairement délimitée, et les liaisons entre modèles doivent être définies avec précision.

Nous allons décrire un ensemble de techniques utilisées pour maintenir l'intégrité du modèle. Le dessin qui suit présente ces techniques et les relations qui existent entre elles.



Contexte borné

Tout modèle a un contexte. Quand on a affaire à un seul modèle, le contexte est implicite. On n'a pas besoin de le définir. Quand on crée une application censée

interagir avec un autre logiciel, par exemple avec une application historique², il est clair que la nouvelle application possède ses propres modèle et contexte, qui sont séparés du modèle et du contexte historiques. Les deux ne peuvent pas être combinés, mélangés, ou confondus. Mais lorsqu'on travaille sur une grosse application d'entreprise, il nous faut définir un contexte pour chaque modèle que nous créons.

Dans tout gros projet, plusieurs modèles entrent en jeu. Mais quand on essaie de combiner des morceaux de code basés sur des modèles distincts, le logiciel devient buggé, peu fiable, et difficile à comprendre. La communication entre les membres de l'équipe devient confuse. On ne distingue souvent pas bien dans quel contexte un modèle donné ne devrait pas être appliqué.

Il n'y a pas de formule mathématique pour diviser un gros modèle en modèles plus petits. Essayez de placer dans le même modèle les éléments qui sont liés, et qui forment un concept naturel. *Un modèle doit être assez petit pour pouvoir être assigné à une seule équipe.* La coopération et la communication au sein d'une même équipe sont plus fluides et complètes, ce qui favorise le travail des développeurs sur le même modèle. Le contexte d'un modèle est l'ensemble de conditions qu'on doit appliquer pour s'assurer que les termes utilisés dans le modèle prennent un sens précis.

L'idée de base, c'est de définir le périmètre d'un modèle, de tracer les limites de son contexte, puis de faire tout ce qui est possible pour préserver son unité. C'est difficile de maintenir un modèle dans un état pur quand il s'étend sur l'intégralité du projet d'entreprise, mais c'est beaucoup plus facile quand il se limite à une zone précise. Définissez explicitement le contexte dans lequel le modèle s'applique. Posez des bornes explicites en termes d'organisation d'équipe, d'utilisation au sein de parties spécifiques de l'application, et de manifestations physiques comme les bases de code et les schémas de bases de données. Essayez de préserver un modèle strictement cohérent à l'intérieur de ces frontières, et ne vous laissez pas distraire ou embrouiller par des problèmes extérieurs.

Un Contexte borné n'est pas un Module. Un Contexte borné fournit le cadre logique à l'intérieur duquel le modèle évolue. Les Modules sont utilisés pour organiser les éléments d'un modèle, donc le Contexte borné englobe le Module.

Quand différentes équipes sont amenées à travailler sur le même modèle, elles doivent faire très attention à ne pas se marcher sur les pieds. Nous devons être constamment conscients que des changements dans le modèle peuvent casser des fonctionnalités existantes. Lorsqu'on utilise des modèles multiples, chacun peut travailler librement sur sa propre partie. Nous connaissons tous les limites de notre modèle, et nous restons

² Legacy application

à l'intérieur de ses frontières. Nous devons juste nous assurer que le modèle reste pur, cohérent et unifié. Chaque modèle supporte plus facilement le refactoring, sans répercussion sur les autres modèles. La conception peut être affinée et distillée afin d'obtenir le maximum de pureté.

Il y a un prix à payer pour avoir des modèles multiples. On doit définir les frontières et les relations entre les différents modèles. Cela demande du travail en plus et un effort de conception supplémentaire, et il y aura peut-être des traductions à faire entre les modèles. On ne pourra pas transférer des objets entre deux modèles, ni invoquer librement un comportement comme s'il n'y avait pas de frontière. Mais ce n'est pas une tâche très difficile, et les bénéfices justifient qu'on se donne cette peine.

Par exemple, nous voulons créer une application d'e-commerce pour vendre des choses sur Internet. Cette application permet aux clients de s'enregistrer, et collecte leurs données personnelles, dont le numéro de carte de crédit. Les données sont conservées dans une base relationnelle. Les clients peuvent se logger, parcourir le site pour chercher des produits, et passer commande. L'application va devoir publier un événement à chaque fois qu'une commande est passée, pour que quelqu'un puisse expédier l'article demandé. Nous voulons aussi construire une interface de reporting pour créer des rapports, en vue de surveiller le statut des produits disponibles, les achats qui intéressent les clients, ce qu'ils n'aiment pas, etc. Au début, nous commençons avec un modèle qui couvre tout le domaine de l'e-commerce. Nous avons cette tentation parce qu'après tout, on nous a demandé de créer une seule grosse application. Mais si nous examinons avec plus d'attention la tâche qui nous occupe, nous découvrons que l'application de boutique en ligne n'est pas vraiment liée à celle de reporting. Elles ont des responsabilités différentes, et elles risquent même de devoir utiliser des technologies différentes. La seule chose vraiment commune est que les données clients et produits sont conservées dans la base de données, et que les deux applications y accèdent.

L'approche préconisée est de créer un modèle séparé pour chacun des domaines, un pour la boutique en ligne, et un pour le reporting. Ils peuvent tous deux évoluer librement sans grande préoccupation de l'autre, et même devenir des applications séparées. Il peut s'avérer que l'application de reporting ait besoin de données particulières que l'application de vente en ligne devra enregistrer en base, mais à part ça elles peuvent se développer indépendamment.

On a besoin d'un système de messagerie qui informe le personnel de l'entrepôt des commandes qui sont passées, pour qu'ils puissent envoyer les marchandises commandées. Le personnel du service expéditions va utiliser une application qui lui donne des informations détaillées sur l'article acheté, la quantité, l'adresse du client, et les conditions de livraison. Nul besoin que le modèle d'e-commerce couvre les deux

domaines d'activité. Il est beaucoup plus simple que l'application de boutique en ligne envoie des Objets Valeurs contenant les informations de commande à l'entrepôt en utilisant des messages asynchrones. Il y a indéniablement deux modèles qui peuvent être développés séparément, et nous devons juste nous assurer que l'interface entre les deux marche bien.

Intégration continue

Une fois qu'un Contexte borné a été défini, nous devons le maintenir dans un état sain. Quand un certain nombre de gens travaillent dans le même Contexte borné, le modèle a une forte tendance à se fragmenter. Plus l'équipe est grosse, plus le problème est de taille, mais même trois ou quatre personnes peuvent rencontrer de sérieux ennuis. Pour autant, vouloir décomposer le système en contextes de plus en plus petits finit par provoquer la perte d'un niveau d'intégration et de cohérence très utile.

Même quand une équipe travaille dans un Contexte borné, il y a de la place pour l'erreur. Il nous faut communiquer à l'intérieur de l'équipe pour vérifier que nous comprenons tous le rôle joué par chaque élément du modèle. Si quelqu'un ne saisit pas bien les relations entre objets, il risque de modifier le code d'une manière qui rentre en contradiction avec l'intention d'origine. Il est facile de se tromper de cette manière quand on ne reste pas à 100% concentré sur la pureté du modèle. Un membre de l'équipe peut écrire du code qui duplique un code existant sans le savoir, ou ajouter du code en doublon au lieu de modifier le code actuel, de peur de casser une fonctionnalité existante.

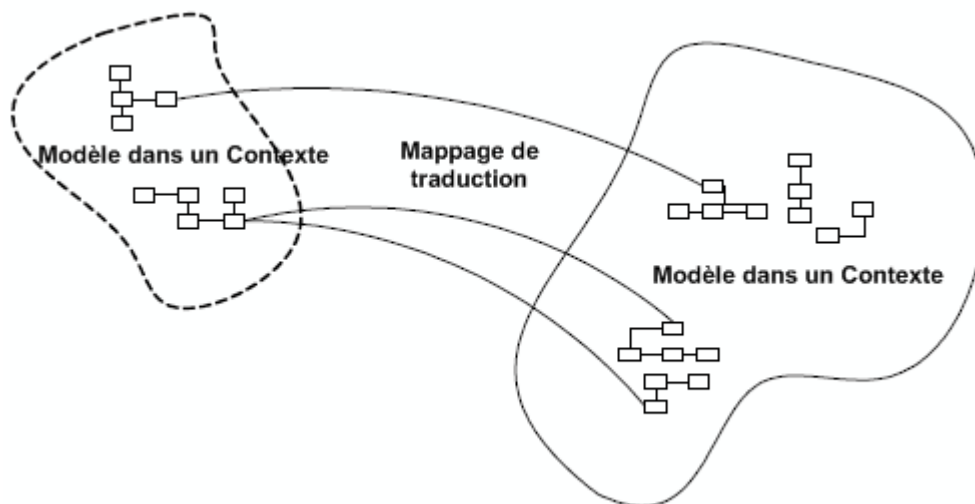
Un modèle n'est pas entièrement défini dès le début. Il est créé, puis il évolue continuellement sur la base de nouvelles perspectives dans le domaine et de retours d'informations du processus de développement. Cela veut dire que de nouveaux concepts risquent d'entrer dans le modèle, et que de nouveaux éléments sont ajoutés au code. Tout cela doit être intégré dans le modèle unifié, et implémenté en conséquence dans le code. C'est pourquoi l'Intégration continue est un procédé nécessaire dans le cadre d'un Contexte borné. Il nous faut un processus d'intégration qui nous assure que tous les éléments ajoutés s'intègrent harmonieusement dans le reste du modèle, et sont correctement implémentés dans le code. Nous avons besoin d'une procédure pour fusionner le code. Plus tôt nous fusionnons le code, mieux c'est. Pour une petite équipe seule, on recommande une intégration quotidienne. Il nous faut aussi mettre en place un processus de compilation. Le code fusionné doit être automatiquement compilé pour pouvoir être testé. Une autre condition nécessaire est de pratiquer des tests automatisés. Si l'équipe possède un outil de test, et a créé une suite de tests, ceux-

ci peuvent être joués après chaque compilation, ainsi toute erreur est signalée. On peut facilement modifier le code pour réparer les erreurs indiquées parce qu'elles sont prises en charge tôt, et ensuite le processus d'intégration, compilation et tests repart.

L'Intégration continue se base sur l'intégration de nouveaux concepts dans le modèle, qui font leur chemin dans l'implémentation où ils sont ensuite testés. Toute incohérence du modèle peut être repérée dans l'implémentation. L'Intégration continue s'applique à un Contexte borné, elle n'est pas utilisée pour traiter les relations entre Contextes voisins.

Carte de Contexte

Une application d'entreprise a de multiples modèles, et chaque modèle a son propre Contexte borné. Il est conseillé d'utiliser le contexte comme base de l'organisation d'équipe. Les personnes d'une même équipe communiquent plus facilement, et ils travaillent plus efficacement à intégrer le modèle et l'implémentation. Même si chaque équipe travaille sur son modèle, il est bon que tout le monde ait une idée du tableau d'ensemble. Une Carte de Contexte est un document qui met en évidence les différents Contextes bornés et leurs liaisons. Une Carte de Contexte peut être un diagramme comme celui-ci-dessous, ou n'importe quel document écrit. Le niveau de détail peut varier. Ce qui est important, c'est que tous ceux qui travaillent sur le projet la partagent et la comprennent.



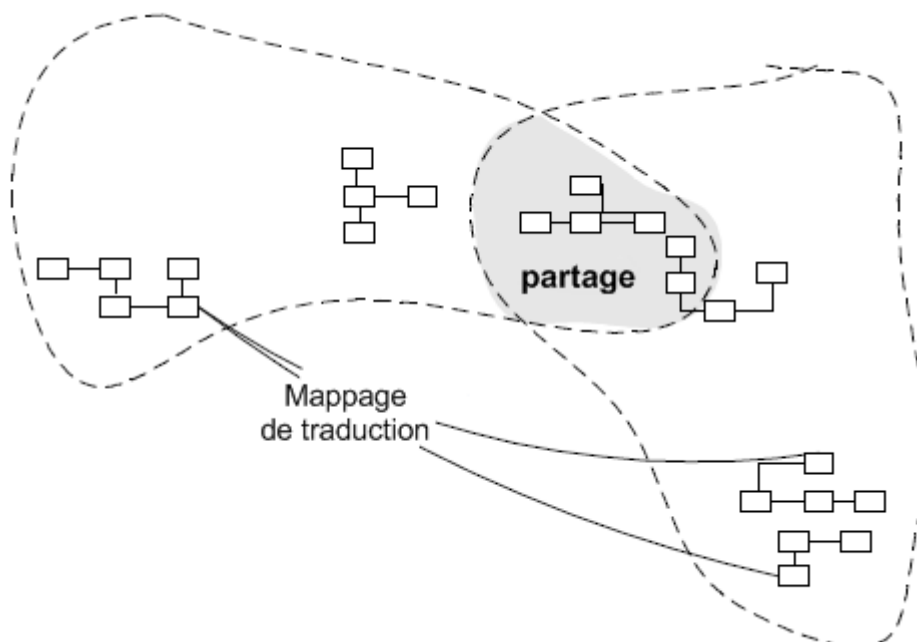
Il ne suffit pas d'avoir des modèles unifiés distincts. Ils doivent être intégrés, parce que chacune des fonctionnalités d'un modèle n'est qu'une partie du système entier. A la fin, les pièces doivent être assemblées, et tout le système doit fonctionner correctement. Si les contextes ne sont pas clairement définis, il est possible qu'ils se

chevauchent. Si les liaisons entre contextes ne sont pas mises en évidence, il y a des chances qu'elles ne marchent pas quand le système sera intégré.

Chaque Contexte borné doit avoir un nom qui fait partie du Langage omniprésent. Cela favorise beaucoup la communication d'équipe lorsqu'on parle du système dans sa globalité. Tout le monde devrait connaître les limites de chaque contexte et les mappages entre contextes et code. Une pratique courante consiste à définir les contextes, puis créer les modules de chaque contexte, et utiliser une convention de nommage pour indiquer le contexte auquel chaque module appartient.

Dans les pages qui suivent, nous allons parler de l'interaction entre différents contextes. Nous présenterons une série de patterns qui peuvent être utilisés pour créer des Cartes de Contexte où les contextes ont des rôles clairs et où leurs relations sont montrées. Noyau partagé et Client-Fournisseur sont des patterns qui comportent un haut niveau d'interaction entre contextes. Chemins séparés est un pattern qu'on utilise quand on veut que les contextes soient fortement indépendants et évoluent séparément. Il y a deux autres patterns qui traitent de l'interaction entre un système et un système historique ou un système externe, ce sont les Services Hôtes ouverts et les Couches Anticorruption.

Noyau partagé



Quand l'intégration fonctionnelle est limitée, le coût de l'Intégration continue peut être jugé trop élevé. Ca s'avère particulièrement vrai lorsque les équipes n'ont pas les compétences ou l'organisation politiques nécessaires pour maintenir une intégration

continue, ou quand une équipe unique est simplement trop grosse et peu maniable. Alors, on peut définir des Contextes bornés séparés et former plusieurs équipes.

Des équipes non coordonnées qui travaillent sur des applications étroitement liées peuvent faire la course chacun de leur côté pendant un moment, mais ce qu'elles produisent risque de ne pas bien s'assembler. Elles peuvent finir par passer plus de temps sur des couches de traduction et des réajustements qu'elles n'en auraient passé sur de l'Intégration continue le cas échéant, ce qui se traduit par des efforts faits en double et la perte des bénéfices d'un Langage omniprésent commun.

Par conséquent, désignez un sous-ensemble du modèle du domaine que les deux équipes s'accordent à partager. Bien sûr cela comprend, en plus de cette portion du modèle, la sous-partie du code ou de la conception de la base de données qui va avec. Ce morceau explicitement partagé possède un statut spécial, et ne devrait pas être modifié sans consultation de l'autre équipe.

Intégrez un système fonctionnel fréquemment, mais un peu moins souvent que le rythme d'intégration continue de chaque équipe. Lors de ces intégrations, jouez les tests des deux équipes.

L'objectif du Noyau partagé est d'éviter les doublons, tout en gardant deux contextes séparés. Développer sur un Noyau partagé demande beaucoup de précaution. Les deux équipes peuvent modifier le code du noyau, et elles doivent intégrer leurs changements. Si les équipes utilisent des copies séparées du code du noyau, elles doivent fusionner le code aussi souvent que possible, au moins une fois par semaine. Un harnais de test devrait être mis en place, pour que chaque changement apporté au noyau soit testé immédiatement. Toute modification du noyau devrait être communiquée à l'autre équipe, et toutes les équipes devraient être informées et tenues au courant des nouvelles fonctionnalités.

Client-Fournisseur

Parfois, il arrive que deux sous-systèmes aient une relation particulière : l'un dépend beaucoup de l'autre. Les contextes dans lesquels ces deux sous-systèmes existent sont distincts, et le résultat des traitements d'un système est déversé dans l'autre. Ils n'ont pas de Noyau partagé, peut-être parce qu'il n'est conceptuellement pas correct d'en avoir un, ou même car il n'est pas techniquement possible que les deux sous-systèmes partagent du code commun. Les deux systèmes sont dans une relation Client-Fournisseur.

Revenons à un précédent exemple. Nous avons parlé plus haut des modèles impliqués dans une application d'e-commerce qui comporte aussi du reporting et un système de

messages. Nous avons déjà dit qu'il était bien mieux de créer des modèles séparés pour tous ces contextes, car un modèle unique serait un goulet d'étranglement permanent et une source de discorde dans le processus de développement. En supposant que nous nous soyons mis d'accord pour avoir des modèles séparés, quelle devrait être la relation entre le sous-système de boutique web et celui de reporting ? Le Noyau partagé ne semble pas être un bon choix. Les sous-systèmes vont très probablement utiliser des technologies différentes dans leur implémentation. L'un est une expérience purement navigateur, tandis que l'autre pourrait être une application avec une IHM riche. Même si l'application de reporting est faite avec une interface web, les concepts principaux des modèles précédemment cités sont différents. Il peut y avoir de l'empiètement, mais pas assez pour justifier un Noyau partagé. Nous allons donc choisir une voie différente. D'un autre côté, le sous-système de shopping en ligne ne dépend pas du tout de celui de reporting. Les utilisateurs de la boutique électronique sont des clients qui parcourent les articles sur le web et passent des commandes. Toutes les données sur les clients, les produits et les commandes sont mises en base. Et c'est tout. L'application de shopping en ligne ne s'intéresse pas vraiment à ce qui arrive aux dites données.

Dans le même temps, l'application de reporting, elle, s'y intéresse et a besoin des données enregistrées par la boutique en ligne. Elle a aussi besoin d'informations supplémentaires pour assurer le service de reporting qu'elle propose. Il se peut que les clients mettent des articles dans leur panier, mais en enlèvent avant de régler. Il se peut qu'ils visitent certains liens plus que d'autres. Ce genre d'informations n'a pas de sens pour l'application de shopping en ligne, mais elles pourraient vouloir dire beaucoup pour celle de reporting. Par conséquent, il faut que le sous-système fournisseur implémente des spécifications dont a besoin le sous-système client. C'est une des connexions qu'il peut y avoir entre les deux sous-systèmes.

Une autre exigence est liée à la base de données utilisée, et plus précisément à son schéma. Les deux applications vont se servir de la même base. Si le sous-système de boutique en ligne était le seul à y accéder, le schéma de base de données pourrait être modifié à tout moment pour refléter ses besoins. Mais le sous-système de reporting doit aussi accéder à la base, il a donc besoin de stabilité dans son schéma. Il est inimaginable que le schéma de la base ne change pas du tout pendant le processus de développement. Ça ne posera pas de souci à l'application de shopping en ligne, mais ça sera certainement un problème pour celle de reporting. Les deux équipes vont devoir communiquer, probablement travailler sur la base de données ensemble, et décider quand le changement doit être réalisé. Cela va représenter une limitation pour le sous-système de reporting, car cette équipe préférerait effectuer la modification rapidement et continuer à développer, au lieu d'attendre l'application de boutique en ligne. Si l'équipe boutique en ligne a le droit de veto, elle peut imposer des limites aux changements à faire sur la base de données, ce qui nuit à l'activité de l'équipe

reporting. Si l'équipe boutique en ligne peut agir indépendamment, elle rompra les accords tôt ou tard, et implémentera des changements auxquels l'équipe de reporting n'est pas préparée. C'est pourquoi ce pattern marche bien si les équipes sont chapeautées par un même management. Cela facilite le processus de décision, et crée une harmonie.

C'est quand on est face à ce genre de scénario que la pièce de théâtre doit commencer. L'équipe reporting doit jouer le rôle du client, tandis que l'équipe boutique en ligne doit endosser celui du fournisseur. Les deux équipes devraient se rencontrer régulièrement ou à la demande, et discuter comme un client le fait avec son fournisseur. L'équipe cliente présente ses besoins, et l'équipe fournisseur prépare ses plans en conséquence. Même si toutes les exigences du client devront être satisfaites au final, c'est au fournisseur d'en décider l'agenda de réalisation. Si certains besoins sont considérés comme vraiment importants, ils devraient être implémentés plus tôt, alors que d'autres exigences peuvent être reportées. L'équipe cliente aura aussi besoin que des données d'entrée et de la connaissance soient partagées par l'équipe fournisseur. Le processus circule dans une seule direction, mais c'est nécessaire dans certains cas.

Il faut que l'interface entre les deux sous-systèmes soit précisément définie. Une suite de tests de conformité devrait être créée et utilisée pour vérifier à tout moment si les spécifications de l'interface sont respectées. L'équipe fournisseur pourra travailler sur sa conception avec moins de réserve car le filet de sécurité de la suite de tests d'interface l'alertera à chaque fois qu'il y aura un problème.

Etablissez une relation client/fournisseur claire entre les deux équipes. Pendant les séances de planning, négociez et budgétez des tâches pour les exigences du client de sorte que chacun comprenne l'engagement et le calendrier.

Développez conjointement des tests d'acceptation automatisés qui valideront l'interface attendue. Ajoutez ces tests à la suite des tests de l'équipe fournisseur pour qu'ils soient joués comme faisant partie de son intégration continue. Ces tests rendront l'équipe fournisseur libre de faire des modifications sans craindre d'effets de bord dans l'application de l'équipe cliente.

Conformiste

Une relation Client-Fournisseur est viable quand les deux équipes ont un intérêt dans la relation. Le client est très dépendant du fournisseur, mais l'inverse n'est pas vrai. S'il y a un management pour faire fonctionner cela, le fournisseur prêterait l'attention nécessaire et écouterait les requêtes du client. Si le management n'a pas clairement

décidé comment les choses sont censées se passer entre les deux équipes, ou si le management est défaillant ou absent, le fournisseur commencera tout doucement à être plus préoccupé par son modèle et sa conception que par l'aide à apporter au client. Après tout, les membres de l'équipe fournisseur ont leurs propres deadlines. Même si ce sont des gens bien, volontaires pour aider l'autre équipe, la pression des délais aura son mot à dire, et l'équipe cliente va en souffrir. Cela arrive aussi quand les équipes appartiennent à des sociétés différentes. La communication est difficile, et la société qui fournit peut ne pas trouver beaucoup d'intérêt à s'investir dans cette relation. Elle va soit apporter une aide sporadique, soit refuser de coopérer tout court. Résultat, l'équipe client se retrouve toute seule, en essayant de se débrouiller du mieux qu'elle peut avec le modèle et la conception.

Quand deux équipes de développement ont une relation Client-Fournisseur dans laquelle l'équipe qui fournit n'est pas motivée pour répondre aux besoins de l'équipe cliente, cette dernière est démunie. L'altruisme peut inciter les développeurs fournisseurs à faire des promesses, mais il est peu probable qu'elles soient tenues. La croyance en ces bonnes intentions mène l'équipe cliente à faire de plans se basant sur des fonctionnalités qui ne seront jamais disponibles. Le projet client sera retardé jusqu'à ce que l'équipe finisse par apprendre à vivre avec ce qu'on lui donne. Une interface taillée pour les besoins de l'équipe cliente n'est pas près de voir le jour.

L'équipe cliente a peu d'options. La plus évidente est de se séparer du fournisseur et d'être complètement seule. Nous examinerons ceci plus tard dans le pattern Chemins séparés. Parfois, les bénéfices apportés par le sous-système du fournisseur n'en valent pas la peine. Il peut être plus simple de créer un modèle séparé, et de concevoir sans avoir à penser au modèle du fournisseur. Mais ce n'est pas toujours le cas.

Parfois il y a de la valeur dans le modèle du fournisseur, et une connexion doit être maintenue. Mais comme l'équipe fournisseur n'aide pas l'équipe cliente, cette dernière doit prendre des mesures pour se protéger des changements du modèle effectués par le fournisseur. Elle va devoir implémenter une couche de translation qui connecte les deux contextes. Il est aussi possible que le modèle du fournisseur soit mal conçu, rendant son utilisation malaisée. Le contexte client peut tout de même s'en servir, mais il devrait se protéger en utilisant une Couche anticorruption dont nous parlerons plus loin.

Si le client est obligé d'utiliser le modèle de l'équipe fournisseur, et si celui-ci est bien fait, c'est peut-être le moment de faire preuve de conformisme. L'équipe cliente peut adhérer au modèle du fournisseur et s'y conformer entièrement. Cela ressemble beaucoup au Noyau partagé, mais il y a une différence importante. L'équipe cliente ne peut pas apporter de changements au noyau. Elle peut simplement l'utiliser comme s'il faisait partie de son modèle, et construire par-dessus le code existant qui lui est fourni.

Il y a beaucoup de cas où ce genre de solution est viable. Lorsque quelqu'un fournit un composant riche, et une interface d'accès à celui-ci, nous pouvons construire notre modèle en y incluant ledit composant comme s'il nous appartenait. Si le composant a une petite interface, il pourrait s'avérer plus judicieux de créer simplement un adaptateur, et de faire la traduction entre notre modèle et le modèle du composant. Cela isolerait notre modèle, et nous pourrions le développer avec un grand degré de liberté.

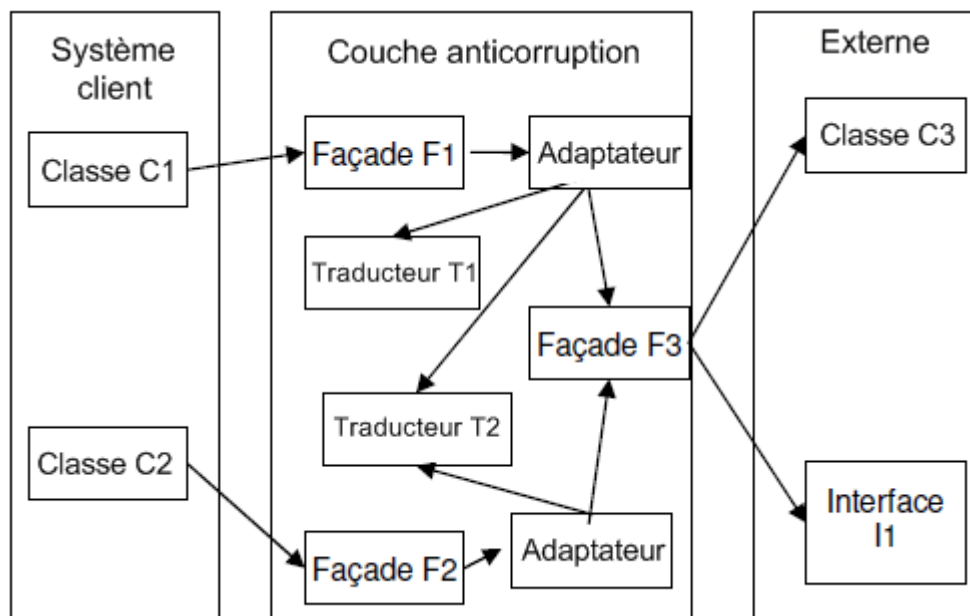
Couche anticorruption

Nous rencontrons souvent des circonstances où nous créons une application qui est obligée d'interagir avec du code logiciel historique ou une application séparée. C'est un défi supplémentaire pour le modélisateur du domaine. Beaucoup d'applications historiques n'ont pas été construites en utilisant des techniques de modélisation de domaine, et leur modèle est confus, broussailleux, il est difficile de le comprendre et de travailler avec. Même s'il a été bien fait, le modèle de l'application historique ne nous est pas d'une grande utilité, car notre modèle est probablement assez différent. Néanmoins, il faut qu'il y ait un certain niveau d'intégration entre notre modèle et le modèle historique, car cela fait partie des prérequis pour pouvoir utiliser la vieille application.

Notre système client peut interagir avec un système externe de différentes manières. L'une d'entre elles est de passer par des connexions réseau. Les deux applications doivent utiliser les mêmes protocoles de communication réseau, et il faut que le client adhère à l'interface utilisée par le système externe. Une autre méthode d'interaction est la base de données. Le système externe travaille avec des données stockées dans une base. Le client est censé accéder à la même base. Dans ces deux cas, nous avons affaire à des données primitives qui sont transférées entre les systèmes. Bien que cela paraisse assez simple, en vérité les données primitives ne contiennent aucune information sur les modèles. On ne peut pas prendre des données dans une base et les traiter entièrement comme des données primitives. Il y a beaucoup de sémantique cachée derrière les données. Une base de données relationnelle contient des données primitives reliées à d'autres, ce qui crée une toile de relations. La sémantique des données est très importante et doit être prise en considération : l'application cliente ne peut pas accéder à la base et y écrire sans comprendre la signification des données utilisées. Il faut bien voir que des parties du modèle externe sont reflétées dans la base de données, et elles viennent s'intégrer dans notre modèle.

Il y a des chances pour que le modèle externe altère le modèle client, si on laisse faire cela. Nous ne pouvons pas ignorer l'interaction avec le modèle externe, mais nous devrions faire attention à isoler notre propre modèle de celui-ci. Nous devrions construire une Couche anticorruption qui se dresse entre notre modèle client et l'extérieur. Du point de vue de notre modèle, la Couche anticorruption est une partie naturelle du modèle, elle ne paraît pas être quelque chose d'étranger. Elle opère avec des concepts et des actions familières à notre modèle. Mais la Couche anticorruption dialogue avec le modèle externe en utilisant le langage externe, pas le langage client. Cette couche agit comme un traducteur dans les deux sens entre deux domaines et langages. Le meilleur résultat possible, c'est que le modèle client reste pur et cohérent sans être contaminé par le modèle externe.

Comment implémenter la Couche anticorruption ? Une très bonne solution consiste à voir la couche comme un Service du point de vue du modèle client. Un Service est très facile à utiliser parce qu'il fait abstraction de l'autre système et nous permet de nous adresser à lui dans nos propres termes. C'est le Service qui va faire la traduction requise, donc notre domaine reste isolé. En ce qui concerne l'implémentation concrète, le Service va être conçu comme une Façade (voir Design Pattern de Gamma et al., 1995). D'autre part, la Couche anticorruption va très probablement avoir besoin d'un Adaptateur. L'Adaptateur vous permet de convertir l'interface d'une classe en une autre qui sera comprise par le client. Dans notre cas, l'Adaptateur n'enrobe pas nécessairement une classe, car son travail consiste à assurer la traduction entre deux systèmes.



La couche anticorruption peut contenir plus d'un Service. Pour chaque Service il y a une Façade qui correspond, et à chaque Façade on adjoint un Adaptateur. On ne doit

pas utiliser un seul Adaptateur pour tous les Services, parce qu'on l'encombrerait de fonctionnalités hétéroclites.

Il nous reste encore un composant à ajouter. L'Adaptateur s'occupe d'envelopper le comportement du système externe, mais nous avons aussi besoin de convertir des objets et des données. Cela se fait au moyen d'un traducteur. Ca peut être un objet très simple, avec peu de fonctionnalités, qui répond au besoin basique de traduire des données. Si le système externe a une interface complexe, il peut s'avérer plus judicieux d'ajouter une Façade supplémentaire entre les adaptateurs et cette interface externe. Cela va simplifier le protocole de l'Adaptateur, et le séparer de l'autre système.

Chemins séparés

Jusqu'ici, nous avons cherché des façons d'intégrer des sous-systèmes, de les faire travailler ensemble, et ce de telle manière que le modèle et la conception restent sains. Cela demande des efforts et des compromis. Les équipes qui travaillent sur ces sous-systèmes doivent passer un temps considérable à régler les relations entre eux. Ils peuvent être obligés de faire constamment des fusions de leur code, et d'effectuer des tests pour s'assurer qu'ils n'ont rien cassé. Parfois, il arrive qu'une des équipes doive passer un temps considérable simplement pour implémenter quelques exigences dont l'autre équipe a besoin. Il faut aussi faire des compromis. C'est une chose de développer quand on est indépendant, de choisir les concepts et associations librement ; c'en est une tout autre de devoir s'assurer que notre modèle s'intègre dans le framework d'un autre système. On va peut-être devoir transformer le modèle juste pour qu'il fonctionne avec l'autre sous-système. Ou introduire des couches spéciales qui assurent les traductions entre les deux sous-systèmes. Il y a des fois où on est obligé de faire ça, mais en certaines occasions on peut emprunter une voie différente. Nous devons évaluer précisément les bénéfices de l'intégration des deux systèmes et l'utiliser seulement si on en tire une vraie valeur. Si nous arrivons à la conclusion que l'intégration apporte plus d'ennuis qu'elle n'est utile, alors on devrait opter pour des Chemins séparés.

Le pattern Chemins séparés concerne le cas où une application d'entreprise peut être constituée de plusieurs applications plus petites qui n'ont pas grand-chose ou rien en commun d'un point de vue modélisation. Il y a un seul jeu de spécifications et, vu de l'utilisateur, il s'agit d'une seule application, mais côté conception et modélisation, cela peut être fait en utilisant des modèles séparés avec des implémentations distinctes. Ce qu'on devrait faire, c'est regarder les spécifications et voir si elles peuvent être

divisées en deux ensembles ou plus qui n'ont pas grand-chose en commun. Si c'est réalisable, alors on peut créer des Contextes bornés séparés et les modéliser indépendamment. Cela a l'avantage de nous donner la liberté de choisir les technologies utilisées pour l'implémentation. Les applications qu'on crée peuvent partager une IHM fine commune qui agit comme un portail, avec des liens ou des boutons qui servent à accéder à chaque application. C'est une intégration mineure qui a trait à l'organisation des applications plutôt qu'au modèle sous-jacent.

Avant de partir sur des Chemins séparés, nous devons nous assurer que nous ne reviendrons pas à un système intégré. Les modèles développés indépendamment sont très difficiles à réintégrer. Ils ont si peu en commun que ça n'en vaut simplement pas la peine.

Service Hôte ouvert

Lorsqu'on essaie d'intégrer deux sous-systèmes, on crée souvent une couche de traduction entre les deux. Cette couche agit comme un tampon entre le sous-système client et le sous-système externe avec lequel on essaie de s'intégrer. Cette couche peut être plus ou moins cohérente, selon la complexité des relations et la manière dont le sous-système externe a été conçu. Si ce dernier s'avère être utilisé non pas par un sous-système client, mais par plusieurs, on doit créer une couche de traduction pour chacun d'entre eux. Toutes ces couches vont répéter la même tâche de traduction, et contenir du code similaire.

Quand un sous-système doit être intégré avec beaucoup d'autres, fabriquer un traducteur sur mesure pour chacun peut enliser l'équipe. Il y a de plus en plus de choses à maintenir, et de plus en plus de choses dont il faut se soucier quand des modifications sont faites.

La solution est de voir le sous-système externe comme un fournisseur de services. Si nous pouvons l'enrober d'un ensemble de Services, alors tous les autres sous-systèmes accéderont à ces Services, et nous n'aurons pas besoin de couche de traduction. La difficulté est que chaque sous-système peut avoir besoin d'interagir de manière spécifique avec le sous-système externe, et créer un ensemble de Services cohérent peut s'avérer problématique.

Définissez un protocole qui donne accès à votre sous-système comme un ensemble de Services. Ouvrez le protocole pour que tous ceux qui doivent s'intégrer avec lui puissent l'utiliser. Améliorez et étendez le protocole pour gérer de nouvelles nécessités d'intégration, sauf quand une équipe a des besoins idiosyncratiques. Dans ce cas,

utilisez un traducteur exceptionnel séparé pour élargir le protocole à cette situation particulière tout en gardant un protocole partagé simple et cohérent.

Distillation

La distillation est le procédé qui consiste à séparer les substances qui composent un mélange. Le but de la distillation est d'extraire une substance particulière du mélange. Pendant le processus de distillation, on peut obtenir des sous-produits, et ils peuvent aussi avoir un intérêt.

Un gros domaine aura un gros modèle, même après que nous l'ayons raffiné et créé beaucoup d'abstractions. Il peut rester volumineux même suite à de nombreux refactorings. Dans une situation comme celle-là, il est peut-être temps de distiller. L'idée est de définir un Cœur de Domaine qui représente l'essence du domaine. Les sous-produits du processus de distillation seront des Sous-domaines génériques qui vont contenir d'autres parties du domaine.

Dans la conception d'un gros système, il y a tellement de composants auxiliaires, tous compliqués et tous absolument nécessaires au succès, que l'essence du modèle du domaine, le vrai capital métier, peut être obscurci et négligé.

Quand on travaille avec un modèle vaste, on devrait essayer de séparer les concepts essentiels des concepts génériques. Au début, nous avons donné l'exemple d'un système de surveillance du trafic aérien. Nous avons dit qu'un Plan de Vol contenait la Route qu'un avion est destiné à suivre. La Route semble être un concept continuellement présent dans ce système. Mais en réalité, ce concept est générique et pas essentiel. Le concept de Route est utilisé dans de nombreux domaines, et un modèle générique peut être conçu pour le décrire.

L'essence de la surveillance de trafic aérien se situe ailleurs. Le système de contrôle connaît la route que l'avion doit suivre, mais il reçoit aussi des informations d'un réseau de radars qui détectent l'avion dans le ciel. Ces données montrent le véritable chemin suivi par l'avion, et il est généralement différent du chemin prescrit. Le système va devoir calculer la trajectoire de l'avion en se basant sur ses paramètres de vol actuels, les caractéristiques de l'avion et la météo. La trajectoire est un chemin à 4 dimensions qui décrit complètement la route que l'avion va suivre au cours du temps. La trajectoire peut être calculée pour les deux minutes qui suivent, les quelques dizaines de minutes à venir ou les deux prochaines heures. Chacun de ces calculs favorise le processus de prise de décision. Tout l'intérêt de calculer la trajectoire de l'avion est de voir s'il y a une chance pour que le chemin de cet avion en croise un

autre. Dans le voisinage des aéroports, lors des décollages et des atterrissages, beaucoup d'avions font des cercles en l'air ou manœuvrent. Si un avion dérive de sa route prévue, il est fort possible qu'un crash se produise. Le système de contrôle du trafic aérien va calculer les trajectoires des avions, et diffuser une alerte s'il y a la possibilité d'une intersection. Les aiguilleurs du ciel vont devoir prendre des décisions rapides, dirigeant les avions afin d'éviter la collision. Quand les avions sont plus éloignés, les trajectoires sont calculées sur de plus longues périodes, et il y a plus de temps pour réagir.

Le module qui synthétise la trajectoire de l'avion à partir des données disponibles constitue ici le cœur du système métier. Il devrait être désigné comme étant le cœur de domaine. Le modèle de routage relève plus d'un domaine générique.

Le Cœur de Domaine d'un système dépend de la manière dont nous regardons ce système. Un système de routage simple verra la Route et ses dépendances comme un élément central de la conception. Le système de surveillance du trafic aérien considèrera la Route comme un sous-domaine générique. Le Cœur de Domaine d'une application peut devenir un sous-domaine générique chez une autre. Il est important d'identifier correctement le Cœur, et de déterminer les relations qu'il entretient avec les autres parties du modèle.

Faites réduire le modèle. Trouvez le Cœur de Domaine et proposez un moyen de le distinguer facilement de la masse de modèles et de code auxiliaires. Mettez l'accent sur les concepts les plus valables et les plus spécialisés. Adoptez un cœur relativement petit.

Mettez vos meilleurs talents sur le Cœur de Domaine, et recrutez en conséquence. Concentrez vos efforts sur le Cœur pour trouver un modèle approfondi et développer un design souple – suffisamment pour satisfaire la vision du système. Justifiez les investissements sur toute autre partie en les mettant en regard du bénéfice apporté au Cœur distillé.

Il est important d'assigner aux meilleurs développeurs la tâche d'implémenter le Cœur de Domaine. Les développeurs ont généralement tendance à aimer les technologies, à apprendre le meilleur et tout dernier langage, à être attirés plus par l'infrastructure que par la logique métier. La logique métier d'un domaine semble être ennuyeuse et peu gratifiante pour eux. Après tout, à quoi bon apprendre les spécificités des trajectoires d'avions ? Quand le projet sera terminé, toute cette connaissance sera de l'histoire ancienne, et aura très peu d'intérêt. Mais la logique métier du domaine est au centre de celui-ci. Des erreurs dans la conception et l'implémentation du cœur peuvent mener à l'abandon complet du projet. Si la logique métier centrale ne fait pas son travail, toutes les paillettes et dorures technologiques ne vaudront rien.

Un Cœur de Domaine ne se crée généralement pas d'un seul coup. Il y a un processus d'affinage et des refactorings successifs sont nécessaires avant que le Cœur n'émerge plus clairement. Nous devons instaurer le Cœur comme pièce centrale du design, et délimiter ses frontières. Il nous faut par ailleurs repenser les autres éléments du modèle en relation avec le nouveau Cœur. Ceux-ci peuvent eux aussi avoir besoin d'être refactorés, et des fonctionnalités peuvent nécessiter des changements.

Certaines autres parties du modèle ajoutent de la complexité sans pour autant capturer ni communiquer une connaissance spécialisée. Tout ce qui est sans rapport avec le sujet rend le Cœur de Domaine plus difficile à discerner et à comprendre. Le modèle s'encrasse de principes généraux que tout le monde connaît, ou de détails appartenant à des spécialités qui ne sont pas votre point de mire principal mais jouent un rôle auxiliaire. Pourtant, quelle que soit leur généricité, ces autres éléments sont essentiels au fonctionnement du système et à l'expression complète du modèle.

Identifiez les sous-domaines cohérents qui ne sont pas la motivation principale de votre projet. Extrayez les modèles génériques de ces sous-domaines et placez-les dans des Modules séparés. N'y laissez aucune trace de vos spécialités. Une fois qu'ils ont été séparés, donnez à leur développement permanent une priorité plus basse qu'au Cœur de Domaine, et évitez d'assigner les développeurs du cœur à ces tâches (parce qu'ils y gagneront peu de connaissance métier). Considérez aussi des solutions disponibles dans le commerce ou des modèles déjà publiés pour ces Sous-domaines génériques.

Tout domaine utilise des concepts qui sont utilisés par d'autres domaines. L'argent et les concepts qui s'y rattachent comme les devises et les taux de change se retrouvent dans différents systèmes. Les graphiques et diagrammes sont un autre concept largement répandu, très complexe en lui-même mais qui peut être utilisé dans beaucoup d'applications.

Il y a différentes manières d'implémenter un Sous-domaine générique :

1. **Solution du commerce.** Celle-ci a l'avantage que toute la solution ait déjà été réalisée par quelqu'un d'autre. Il y a toujours une courbe d'apprentissage liée au produit, et cette solution implique des dépendances. Si le code est buggé, vous devez attendre qu'il soit corrigé. Vous devez aussi utiliser des compilateurs et des versions de bibliothèques spécifiques. L'intégration ne se fait pas aussi facilement comparé à un système maison.
2. **Sous-traitance.** La conception et l'implémentation sont confiées à une autre équipe, probablement d'une société différente. Cela vous permet de vous concentrer sur le Cœur de Domaine, et vous enlève le fardeau d'un autre domaine à traiter. Il y a toujours le désagrément de devoir intégrer le code sous-

traité. L'interface utilisée pour dialoguer avec le sous-domaine doit être définie et communiquée à l'autre équipe.

3. **Modèle existant.** Une solution pratique consiste à utiliser un modèle déjà créé. Il existe certains livres qui ont publié des patterns d'analyse, et ils peuvent être utilisés comme source d'inspiration pour nos sous-domaines. Il se peut qu'il ne soit pas possible de recopier les patterns *ad litteram*, mais beaucoup d'entre eux peuvent être utilisés avec de petites modifications.
4. **Implémentation maison.** Cette solution a l'avantage d'accomplir le meilleur niveau d'intégration. Cela veut bien entendu dire des efforts supplémentaires, y compris le fardeau de la maintenance.

6

DDD compte aujourd'hui : une interview d'Eric Evans

InfoQ.com interviewe le fondateur de Domain Driven Design, Eric Evans, pour replacer DDD dans le contexte d'aujourd'hui :

Pourquoi DDD est-il aujourd'hui plus important que jamais ?

Fondamentalement, DDD est le principe selon lequel nous devrions nous concentrer sur les enjeux profonds du domaine dans lequel nos utilisateurs sont impliqués, selon lequel le meilleur de nos esprits devrait être dévoué à la compréhension ce domaine, et à la collaboration avec les experts du domaine pour réussir à accoucher d'une forme conceptuelle que nous pouvons utiliser pour bâtir des logiciels flexibles et puissants.

C'est un principe qui ne passera jamais de mode. Il s'applique à chaque fois que l'on opère dans un domaine complexe et très élaboré.

La tendance sur le long terme est d'appliquer l'informatique à des problèmes de plus en plus compliqués, de plus en plus profondément au cœur de ces métiers. Il me semble que cette tendance s'est interrompue pendant quelques années, au moment où nous connaissions l'explosion du web. L'attention était détournée d'une logique riche et de solutions approfondies, tant il y avait de valeur dans le simple fait de mettre sur le web des données avec un comportement très basique. Il y avait beaucoup à faire en la matière, et même réaliser des choses simples sur le web a été difficile pendant un moment, donc cela a absorbé tout l'effort de développement.

Mais maintenant que le niveau basique d'utilisation du web a largement été assimilé, les projets commencent à se faire de nouveau plus ambitieux sur la logique métier.

Très récemment, les plateformes de développement web ont commencé à être suffisamment mûres pour rendre le développement web assez productif pour DDD, et

il y a un certain nombre de tendances positives. Par exemple, l'approche SOA, quand elle est bien utilisée, nous fournit une façon très pratique d'isoler le domaine.

Dans le même temps, les processus Agiles ont eu assez d'influence pour que la plupart des projets actuels aient au moins l'intention de faire des itérations, de travailler étroitement avec les partenaires métier, d'appliquer l'intégration continue, et de travailler dans un environnement fortement communicant.

DDD paraît donc de plus en plus important pour le futur prévisible, et il semble que des fondations soient en place.

Les plateformes technologiques (Java, .NET, Ruby et autres) sont en constante évolution. Comment Domain Driven Design se situe-t-il par rapport à cela ?

En fait, les nouvelles technologies et les nouveaux processus devraient être jugés sur leur capacité à aider les équipes à se concentrer sur leur domaine, plutôt que de les en distraire. DDD n'est pas spécifique à une plateforme technologique, mais certaines plateformes donnent des moyens plus expressifs de créer de la logique métier, et certaines plateformes ont moins d'encombrements parasites. Vis-à-vis de ce dernier caractère, les quelques dernières années montrent une direction optimiste, particulièrement après l'affreuse fin des années 90.

Java a été le choix par défaut ces dernières années ; pour ce qui est de son expressivité, elle est typique des langages orienté objet. En ce qui concerne les encombrements parasites, le langage de base n'est pas trop mauvais. Il offre la *garbage collection*, ce qui en pratique s'avère essentiel. (Contrairement au C++, qui exigeait d'accorder trop d'attention aux détails de bas niveau.) Il y a du fouillis dans la syntaxe Java, mais il y a toujours moyen de rendre lisibles des *plain old java objects* (POJOs). Et certaines des innovations de syntaxe de Java 5 favorisent la lisibilité.

Mais quand le framework J2EE est paru pour la première fois, il enterrait littéralement cette expressivité basique sous des montagnes de code du framework. En suivant les premières conventions (comme EJB Home, les accesseurs préfixés par get/set pour toutes les variables, etc.), on produisait des objets affreux. Les outils étaient si lourds que le simple fait de les faire marcher absorbait toute la capacité des équipes de développement. Et il était si difficile de changer les objets, une fois que le gigantesque fatras de code généré et de XML avait été régurgité, que les gens ne faisaient pas beaucoup de modifications. Cette plateforme rendait une modélisation du domaine efficace pratiquement impossible.

Il faut ajouter à cela l'impératif de produire des IHM web convoyées par http et html (qui n'ont pas été conçus dans cet objectif), en utilisant des outils de première génération quelque peu primitifs. Pendant cette période, créer et maintenir une interface utilisateur décente était devenue si difficile qu'il restait peu d'attention à

accorder à la conception de fonctionnalités internes complexes. L'ironie de la chose, c'est qu'au moment même où la technologie objet prenait le pouvoir, la modélisation et la conception sophistiquées prenaient un sévère coup sur la tête.

La situation était similaire sur la plateforme .NET, où certaines problématiques étaient un peu mieux traitées, et d'autres un peu moins bien.

Ce fut une période décourageante, mais la tendance s'est inversée au cours des quatre dernières années environ. D'abord, si l'on regarde Java, il y a eu la convergence d'une nouvelle sophistication dans la communauté sur la manière d'utiliser les frameworks de façon sélective, et d'une ménagerie de nouveaux frameworks (open source pour la plupart) qui s'améliorent incrémentalement. Des frameworks comme Hibernate et Spring gèrent des tâches spécifiques que J2EE essayait de traiter, mais de façon bien plus légère. Des approches comme AJAX tentent de s'attaquer au problème de l'interface utilisateur de manière moins laborieuse. Et les projets sont beaucoup plus intelligents maintenant dans leurs choix d'utiliser les composants de J2EE qui leur apportent de la valeur, en y incorporant certains de ces autres éléments plus récents. C'est pendant cette ère que le terme POJO a été proposé.

Le résultat, c'est une diminution incrémentale mais sensible des efforts techniques des projets, et une amélioration distincte dans l'isolation de la logique métier du reste du système afin qu'elle puisse être écrite en termes de POJOs. Ça ne produit pas automatiquement une conception dirigée par le domaine, mais cela en fait une opportunité réaliste.

Voilà pour le monde Java. Ensuite, vous avez les nouveaux arrivants comme Ruby. Ruby a une syntaxe très expressive, et sur ce plan fondamental ce devrait être un très bon langage pour DDD (bien que je n'aie pas encore entendu parler de beaucoup d'utilisations réelles de DDD dans ce genre d'applications.) Rails a généré beaucoup d'excitation parce qu'il semble finalement rendre la création d'interfaces web aussi facile que les IHM l'étaient au début des années 90, avant le Web. Aujourd'hui, cette capacité a surtout été exploitée dans la construction d'applications web faisant partie de celles, très nombreuses, qui n'ont pas une richesse de domaine très importante derrière elles ; et pour cause, même celles-ci étaient douloureusement compliquées dans le passé. Mais j'ai espoir qu'avec la diminution de l'aspect implémentation IHM du problème, les gens y verront une opportunité de focaliser davantage leur attention sur le domaine. Si jamais l'utilisation de Ruby commence à prendre cette direction, je pense qu'il pourrait constituer une excellente plateforme pour DDD. (Quelques pans d'infrastructure devraient probablement être comblés.)

Sur des problématiques plus pointues, il y a les efforts dans le champ des *domain-specific languages* (DSLs), dont je pense depuis longtemps qu'ils pourraient être le prochain grand pas pour DDD. A cette date, nous n'avons toujours pas d'outil qui nous

donne réellement ce que nous attendons. Mais les gens font plus d'expériences que jamais dans ce domaine, et cela me donne espoir.

Aujourd'hui, pour autant que je sache, la plupart des gens qui tentent d'appliquer DDD travaillent en Java ou .NET, et quelques-uns en Smalltalk. Donc c'est la mouvance positive du monde Java qui profite des effets immédiats.

Que s'est-il passé dans la communauté DDD depuis que vous avez écrit votre livre ?

Quelque chose qui me passionne, c'est quand les gens prennent les principes dont j'ai parlé dans mon livre et les utilisent d'une manière que je n'aurais jamais soupçonnée. Il y a par exemple l'utilisation du design stratégique chez StatOil, la compagnie nationale de pétrole norvégienne. Les architectes là-bas ont écrit un retour d'expérience à ce propos. (Vous pouvez le lire à l'adresse <http://domaindrivendesign.org/articles/>.)

Entre autres, ils ont pris la carte de contexte et l'ont appliquée à l'évaluation de logiciels du marché dans des prises de décisions achat/réalisation.

Pour citer un exemple assez différent, certains d'entre nous ont exploré d'autres problématiques en développant une bibliothèque de code Java contenant quelques objets du domaine fondamentaux dont ont besoin beaucoup de projets. Les gens peuvent consulter tout ça à l'adresse :

<http://timeandmoney.domainlanguage.com>

Nous avons exploré, par exemple, jusqu'où l'on peut pousser l'idée d'un langage *fluent* spécifique au domaine, tout en continuant à implémenter les objets en Java.

Il y a pas mal de choses qui se passent. J'apprécie toujours quand des personnes me contactent pour me faire part de ce qu'ils font.

Avez-vous des conseils pour les gens qui essaient d'apprendre DDD aujourd'hui ?

Lisez mon livre ! ;-) Essayez aussi d'utiliser Timeandmoney dans votre projet. Un de nos objectifs d'origine était de proposer un bon exemple grâce auquel les gens pourraient apprendre en l'utilisant.

Une chose qu'il faut avoir en tête, c'est que DDD est principalement pratiqué au niveau de l'équipe, donc au besoin vous devrez peut-être vous faire évangéliste. Sur un plan plus réaliste, vous pourriez aussi chercher un projet où les gens font déjà un effort pour faire du DDD.

Gardez à l'esprit certains pièges de la modélisation de domaine :

- 1) Gardez la main. Les modélisateurs ont besoin de coder.

- 2) Concentrez-vous sur des scénarios concrets. La réflexion abstraite doit être ancrée dans des cas concrets.
- 3) N'essayez pas d'appliquer DDD à tout. Tracez une carte de contexte et décidez où vous allez faire des efforts sur DDD ou pas. Ensuite, ne vous en souciez pas hors de ces limites.
- 4) Expérimentez beaucoup et attendez-vous à faire beaucoup d'erreurs. La modélisation est un processus créatif.

A propos d'Eric Evans

Eric Evans est l'auteur de "Domain-Driven Design: Tackling Complexity in Software," Addison-Wesley 2004.

Depuis le début des années 90, il a travaillé sur de nombreux projets, développant de larges systèmes métier en objet, avec de nombreuses approches et résultats différents. Le livre est une synthèse de cette expérience. Il présente un système de techniques de modélisation et conception que des équipes ont utilisées avec succès pour aligner des systèmes logiciels complexes sur les besoins métier et garder des projets agiles quand les systèmes deviennent plus gros.

Eric dirige maintenant « Domain Language », un groupe de consulting qui coache et forme les équipes à appliquer Domain Driven Design, et les aide à rendre leur travail de développement plus productif et plus utile pour le métier.

Lexique français-anglais des termes DDD

Agrégat	<i>Aggregate</i>
Avancée majeure	<i>Breakthrough</i>
Carte de Contexte	<i>Context Map</i>
Chemins séparés	<i>Separate Ways</i>
Cœur de Domaine	<i>Core Domain</i>
Conception dirigée par le modèle	<i>Model-Driven Design</i>
Contexte borné	<i>Bounded Context</i>
Couche anticorruption	<i>Anticorruption Layer</i>
Entité	<i>Entity</i>
Entrepôt	<i>Repository</i>
Fabrique	<i>Factory</i>
Fabrique abstraite	<i>Abstract Factory</i>
Langage omniprésent	<i>Ubiquitous Language</i>
Méthode de fabrication	<i>Factory Method</i>
Modèle du domaine	<i>Domain Model</i>
Noyau partagé	<i>Shared Kernel</i>
Objet-Valeur	<i>Value Object</i>
Racine d'agrégat	<i>Aggregate Root</i>
Service Hôte ouvert	<i>Open Host Service</i>
Sous-domaine générique	<i>Generic Subdomain</i>