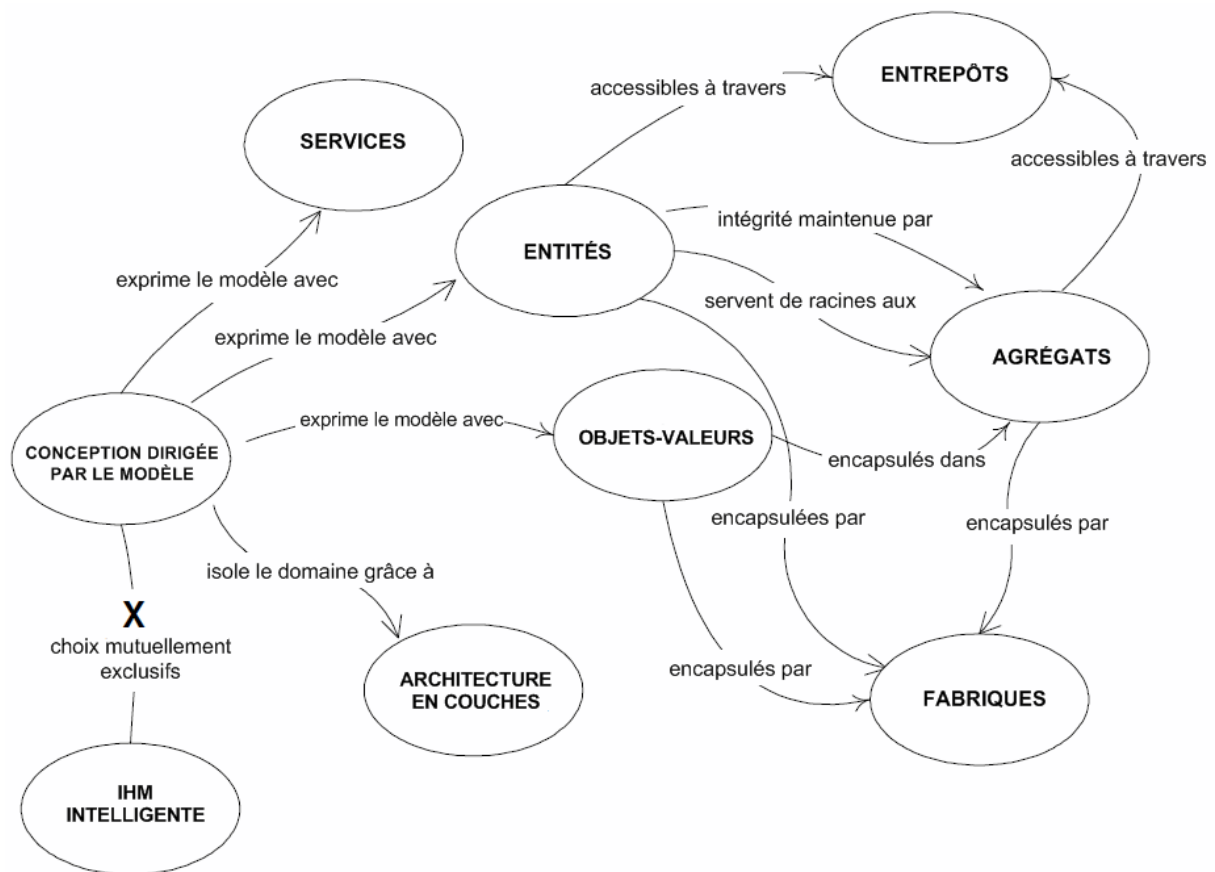


Domain-Driven Design *Vite fait*



par Abel Avram & Floyd Marinescu

édité par : Dan Bergh Johnsson, Vladimir Gitlevich

traduction : Guillaume Lebur

Domain-Driven Design

Vite fait

Deuxième partie : notions avancées

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

4	Refactorer pour une vision plus profonde	5
	Refactorer en continu.....	5
	Mettre au jour les concepts clés	7
5	Préserver l'intégrité du modèle	12
	Contexte borné.....	13
	Intégration continue.....	16
	Carte de Contexte.....	17
	Noyau partagé	18
	Client-Fournisseur	19
	Conformiste	21
	Couche anticorruption	23
	Chemins séparés	25
	Service Hôte ouvert.....	26
	Distillation	27
6	DDD compte aujourd'hui : une interview d'Eric Evans.....	32
	Lexique français-anglais des termes DDD	38

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. Ca 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 refactorée. 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

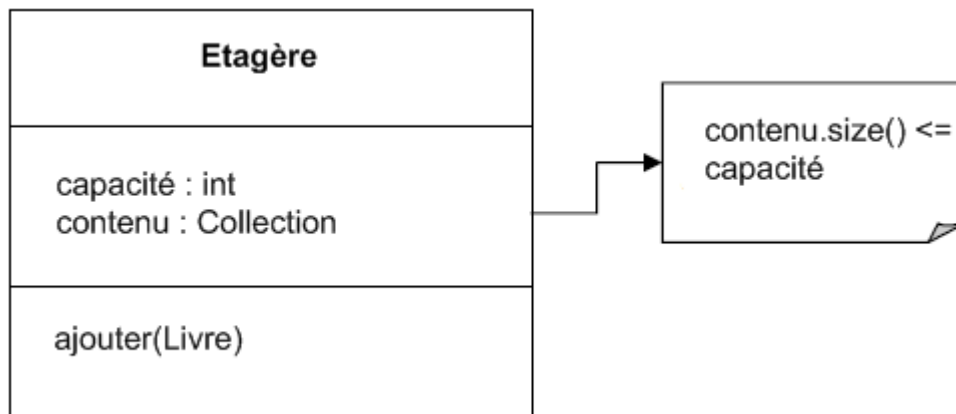
totalément mal compris. Tout ça fait partie de l'apprentissage d'un nouveau domaine. Mais à mesure que nous construisons notre Langage omniprésent, les concepts clés finissent par s'y intégrer. C'est là qu'on doit commencer à chercher des concepts implicites.

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

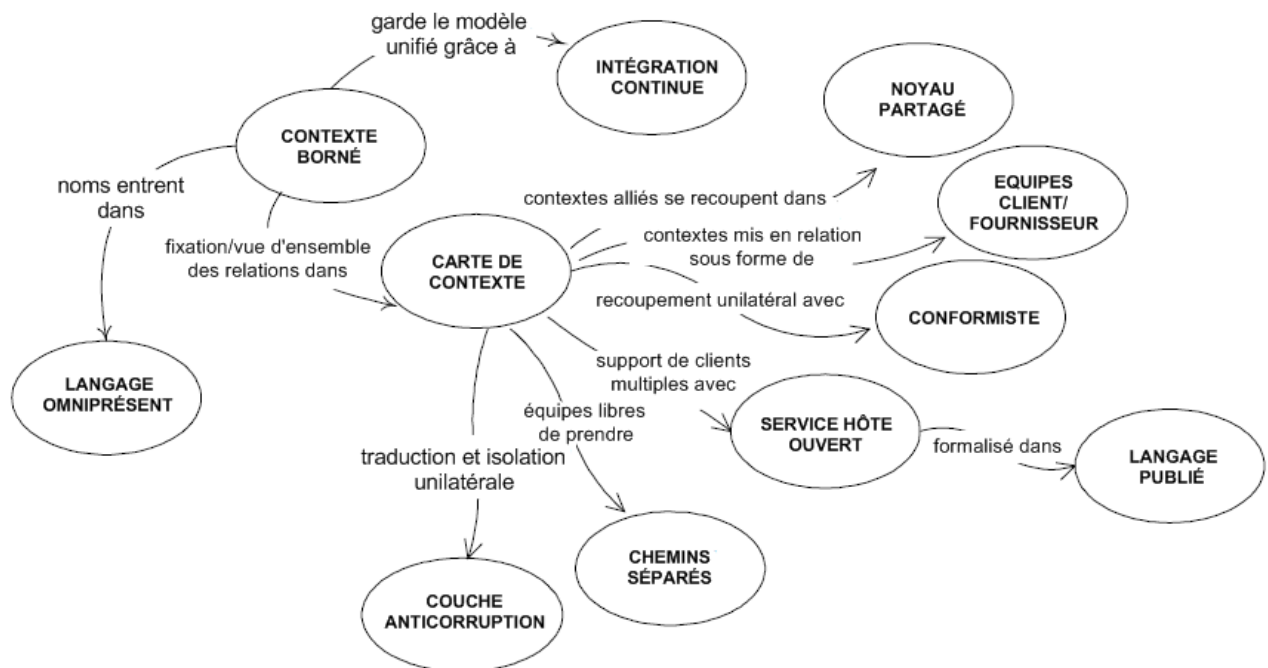
Ce chapitre traite des gros projets qui nécessitent les efforts combinés de multiples équipes. On se retrouve face à un nouvel ensemble de défis lorsqu'on confie à plusieurs équipes, dans des conditions de management et de coordination diverses, la tâche de développer un projet. Les projets d'entreprise 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 historique¹, 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

¹ 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.

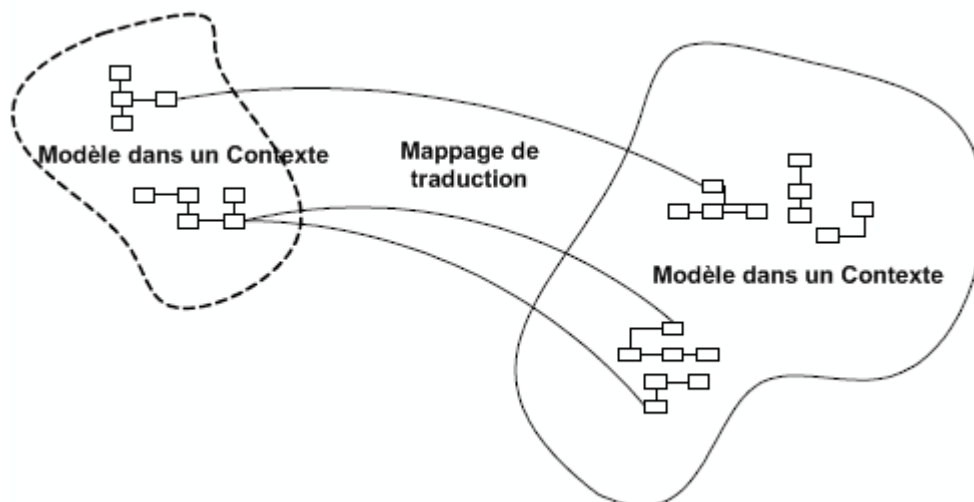
Un modèle n'est pas entièrement défini dès le début. Il est créé, puis il évolue continuellement sur la base de nouvelles perspectives dans le domaine et de retours d'informations du processus de développement. Cela veut dire que de nouveaux concepts risquent d'entrer dans le modèle, et que de nouveaux 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.



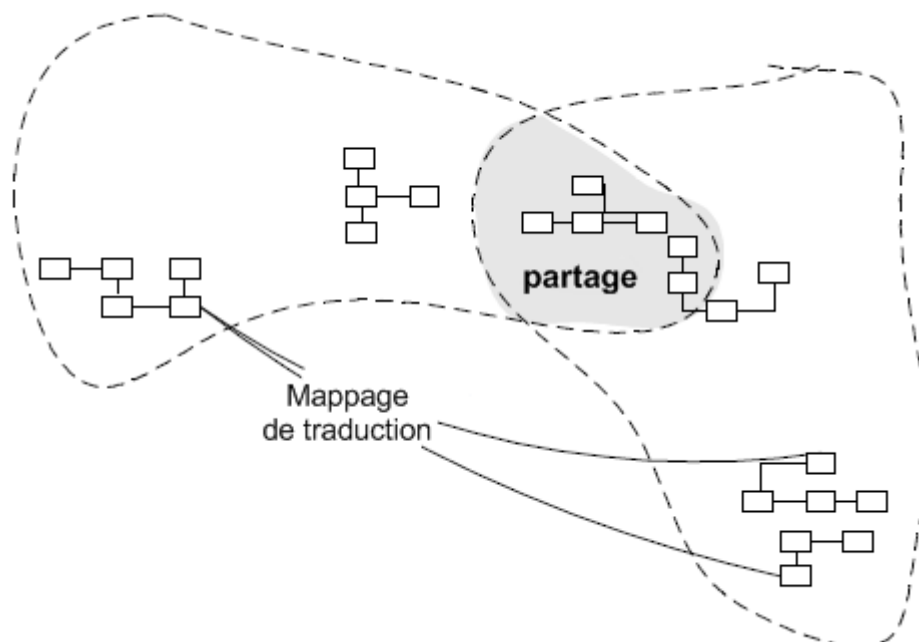
Il ne suffit pas d'avoir des modèles unifiés distincts. Ils doivent être intégrés, parce que chacune des fonctionnalités d'un modèle n'est qu'une partie du système entier. À la fin, les pièces doivent être assemblées, et tout le système doit fonctionner correctement. Si les contextes ne sont pas clairement 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 prêtera l'attention nécessaire et écoutera 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 cliente 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.

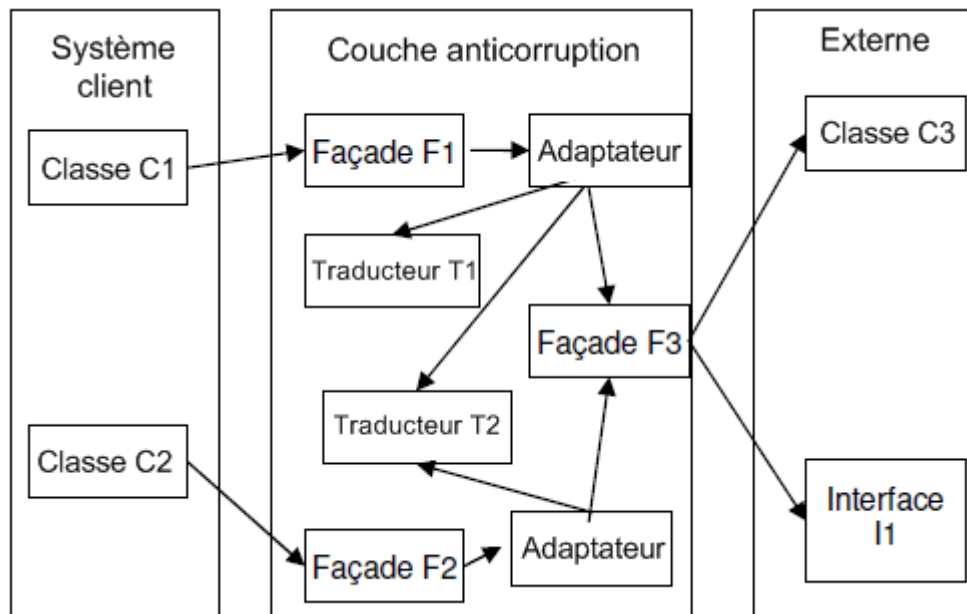
Couche anticorruption

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 généralité, 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. Ca 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

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>
Contexte borné	<i>Bounded Context</i>
Couche anticorruption	<i>Anticorruption Layer</i>
Noyau partagé	<i>Shared Kernel</i>
Service Hôte ouvert	<i>Open Host Service</i>
Sous-domaine générique	<i>Generic Subdomain</i>