Conception efficace des Aggregates

1ere partie : Modéliser un Aggregate unique

Vaughn Vernon: vvernon@shiftmethod.com

Traduction : Fabien Tregan (fabien[at]tregan.fr)

NDLT : La vaste majorité de la documentation sur Domain Driven Design n'est pas disponible en français. Elle contient des termes anglais qui ont un sens précis dans le contexte précis de DDD. J'ai utilisé pour les noter la même convention que j'utilise sur mes projets pour les noms du domaine métier : Ils sont dans la langue utilisée pour l'***Ubiquitous Language*** (ici l'anglais), les mots commencent par des majuscules et sont en italique. Pour distinguer les mots du domaine de DDD de ceux du domaine de l'exemple qui illustre cette documentation, ceux de DDD sont en gras.

Regrouper avec soin les ***Entities*** et les ***Value Objects*** en ***Aggregates*** aux frontières cohérentes peut sembler à priori être une tâche rapide. Mais de toutes les tactiques que [DDD] propose, il s'agit d'un des aspects les moins bien compris.

Pour aborder le problème, il peut être intéressant de commencer par étudier certaines questions qui reviennent fréquemment : L'***Aggregate*** est-il un moyen de regrouper un graphe d'objets sémantiquement liés sous un parent commun ? Si oui, y a-t-il une limite à se fixer pour la taille des graphes ? Etant donné que les instances d'***Aggregates*** peuvent référencer d'autres instances d'***Aggregates***, doivent-elles pouvoir être naviguées en profondeur en modifiant les objets au passage ? Et que veulent dire les concepts d'***Invariants*** et de ***Consistency Boundary*** ? C'est la réponse à cette dernière question qui influencera beaucoup la réponse aux autres.

Il y a différentes façons de mal modéliser les ***Aggregates***. On peut se laisser emporter par la facilité de la composition, au risque de voir les ***Aggregates*** devenir trop gros. A l'inverse, on pourrait arriver à des ***Aggregates*** trop dépouillés qui ne parviendraient plus à assurer la protection des vrais ***Invariants***. Comme nous le verrons, il est impératif à la fois d'éviter ces deux extrêmes, et de concentrer son attention sur les règles métier.

Conception d'une application de gestion Scrum

La meilleur façon d'expliquer les ***Aggregates*** est de suivre un exemple. Notre société imaginaire développe un logiciel pour supporter les projets Scrum : ProjectOvation. Il suit le modèle complet de la gestion de projet Scrum, avec ses *Produits*, son *Product Owner*, l'*Equipe*, les *Backlog Items*, la *Release En Cours* et les *Sprints*. L'objectif est de supporter l'ensemble de Scrum. Ceci aura l'avantage de nous fournir un domaine métier qui devrait vous être familier ; la terminologie de Scrum est le point de départ de notre ***Ubiquitous Language***. ProjectOvation sera disponible par abonnement, en SaaS. Chaque organisation cliente est enregistrée comme un *Tenant*.

La société a rassemblé un groupe d'experts Scrum et de développeurs Java[1] talentueux, mais leur expérience de DDD est encore assez limitée. L'équipe va forcément faire quelques erreurs au cours de son apprentissage, mais ils vont progresser tout comme nous pouvons le faire. Leurs tentatives pourront nous aider à reconnaître et à corriger des situations défavorables que nous pourrions avoir crées sur nos propres projets.

Les concepts de ce domaine, et les problématiques de performance et de scalabilité qui l'accompagnent, sont plus complexes que ce qu'aucun de membre de l'équipe n'a rencontré jusqu'à présent. Pour répondre à ces difficultés, une des outils tactiques de DDD qu'ils vont employer sera les ***Aggregates***.

Comment l'équipe devrait-elle choisir les meilleurs regroupements d'objets ? Le principe d'***Aggregate*** parle de composition, et fait allusion au masquage d'information, ce que

[1] Bien que les exemples utilisent Java et Hibernate, cette documentation sera applicable indifférement à des projets C# et Nhibernate.

l'équipe comprend bien. Il parle aussi de ***Consistency Boundaries*** et de ***Transactions***, mais ils ne n'en sont pas vraiment préoccupé pour l'instant : le système de persistance qu'ils ont choisi aide à gérer les commits unitaires des données. Cette erreur cruciale d'attention portée aux conseils donnés par le concept d'***Aggregate*** va les obliger à revenir en arrière. L'équipe avait pris en considérations les affirmations suivantes dans l'***Ubiquitous Language*** :

- Les produits ont des backlog items, des releases et des sprints
- De nouveaux blacklog items du produit sont planifiés
- De nouvelles releases sont planifiées
- De nouveaux sprints sont planifiés
- Un backlog item peut être planifié pour une release
- Un backlog item planifié peut faire partie des engagement d'un sprint

A partir de là, ils avaient un premier aperçu du modèle et ont commencé la conception. Voyons comment cela s'est passé.

Premier essai : Regroupement dans un grand Aggregate.

L'équipe s'est particulièrement intéressé aux mots « les produits ont » dans les phrases précédentes. Certains ont pensé que cela semblait décrire une composition : un ensemble d'objets interconnectés en un graph. Il a été considéré comme important de maintenir le cycle de vie de ces objets ensemble. L'équipe a alors retenu les règles de cohérence suivantes :

- Si un *Backlog Item* fait partie des engagements d'un *Sprint*, il ne faut pas permettre de retirer ce *Backlog Item* du système.
- Si un *Backlog Item* fait partie des engagements d'un *Sprint*, il ne faut pas permettre de retirer ce *Sprint* du système.
- Si un *Backlog Item* est planifié pour une *Release*, il ne faut pas permettre de retirer cette *Release* du système.
- Si un *Backlog Item* est planifié pour une *Release*, il ne faut pas permettre de retirer ce *Backlog Item* du système.

Le *Product* a été modélisé comme un gros ***Aggregate***. L'objet racine *Product* contient toutes les instances de *Backlog Item*, des *Releases*, et des *Sprints* qui lui sont associés. L'interface permet de se prémunir de toute suppression inappropriée par le client. Le modèle est représenté par le code suivant et la Figure 2 :

```
public class Product extends ConcurrencySafeEntity {
private Set<BacklogItem> backlogItems;
private String description;
private String name;
private ProductId productId;
private Set<Release> releases;
private Set<Sprint> sprints;
private TenantId tenantId;
...
}
```

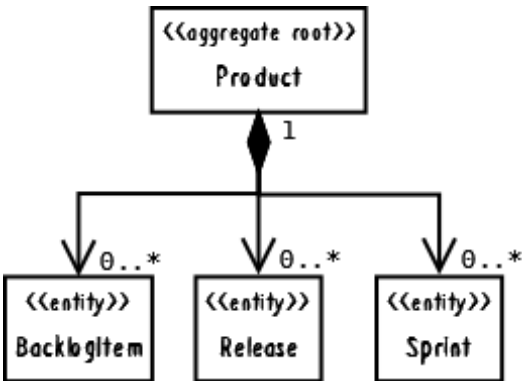

Figure 1 : *Product* modélisé comme un gros *Aggregate.*

Le gros ***Aggregate*** avait l'air intéressant, mais il n'était pas vraiment efficace. Une fois l'application déployée dans son environnement multi-utilisateurs, on a commencé à voir apparaître fréquemment des erreurs transactionnelles. Regardons de plus près quelques schéma d'utilisation et comment se passent les interactions avec notre implémentation de la solution. Notre instance d'***Aggregate*** emploi la concurrence optimiste pour protéger les objets persistants des modifications simultanées conflictuelles par différents clients, ce qui évite l'emploi de verrous gérés par la base de données. Les objets comportent un numéro de version qui est incrémenté à chaque modification et vérifié avant la persistance en base. Si la version de l'objet en base est plus élevée que celle de l'objet du client, l'objet client est considéré comme périmé et la mise à jour est rejetée.

Intéressons-nous à un scénario courrant d'utilisation multi-utilisateur :

- Deux utilisateurs, Bill et Joe consultent le même *Product* marqué en version 1 et commence à travailler dessus.

- Bill crée un nouveau *Backlog Item* et soumet ses changements. La version du *Product* passe à 2.
- Joe planifie une nouvelle *Release* et essai de la sauvegarder, mais la sauvegarde échoue parce qu'il était parti de la version 1.

Les mécanismes de persistances sont utilisés plus ou moins de cette façon pour gérer les accès concurrents[2]. On pourrait rétorquer que la stratégie de gestion de la concurrence peut être modifiée, nous y reviendrons un peu plus tard. Cette approche est en fait importante pour protéger les invariants de modifications concurrentes.

Ce problème de consistance est apparut avec seulement deux utilisateurs. Ajoutons quelques utilisateurs, et cela va devenir un réel problème. Dans Scrum, plusieurs utilisateurs font fréquemment en parallèle ce type de modification durant le sprint planning et le déroulement du sprint. Devoir fréquemment rejeter toutes les modifications sauf une serait complètement inacceptable.

Rien, à part la planification d'un nouveau *Backlog Item*, ne devrait logiquement interférer avec la planification d'une nouvelle *Release*. Alors pourquoi la sauvegarde de Joe a-t-elle échouée ? Le cœur du problème et que le gros ***Aggregate*** a été conçu avec en tête de faux invariants et non avec de réelles règles métier. Ces faux invariants sont des contraintes artificielles imposées par les développeurs. Mais l'équipe a d'autres possibilités pour empêcher les suppressions inappropriées sans rendre l'application arbitrairement restrictive. En plus des problèmes transactionnels, la conception apporte aussi des problèmes de performance et de scalabilité.

Second essai : Plusieurs Aggregate.

Etudions maintenant un autre modèle, représenté par la Figure 2,

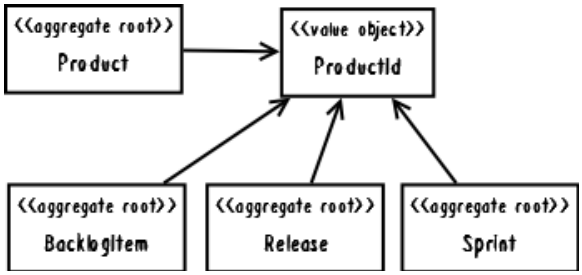

Figure 2 : *Product* et concepts associés modélisés comme des *Aggregates* distincts.

Casser le gros Aggregate en quatre va modifier certaines méthodes du contrat sur la classe *Product*. Avec le modèle en gros ***Aggregate***, la signature des méthodes ressemblait à ceci :

```
public class Product ... {
   ...
   public void planBacklogItem(
      String aSummary, String aCategory,
      BacklogItemType aType, StoryPoints aStoryPoints)
{
      ...
   }

   public void scheduleRelease(
      String aName, String aDescription,
      Date aBegins, Date anEnds) {
      ...
   }

   public void scheduleSprint(
      String aName, String aGoals,
      Date aBegins, Date anEnds) {
      ...
   }
   ...
}
```

Toutes ces méthodes sont des commandes [CQS]. Ceci veut dire que comme elles modifient l'état du *Product* en ajoutant un nouvel élément à l'une de ses collections, leur type de retour est `void`. Mais avec le modèle en ***Aggregates*** multiples, on obtient :

```
public class Product ... {
   ...
   public BacklogItem planBacklogItem(
      String aSummary, String aCategory,
      BacklogItemType aType, StoryPoints aStoryPoints)
{
      ...
   }

   public Release scheduleRelease(
      String aName, String aDescription,
      Date aBegins, Date anEnds) {
      ...
   }

   public Sprint scheduleSprint(
      String aName, String aGoals,
      Date aBegins, Date anEnds) {
      ...
   }
   ...
}
```

Dans cette conception, les méthodes ont un contrat de requête [CQS] et agissent comme des factories. Elles créent chacune une nouvelle instance d'***Aggregate*** et renvoient maintenant une référence vers celle-ci. Quand un client veut planifier un ***Backlog Item***, l'***Application Service*** transactionnel doit alors effectuer ceci :

[2] Par exemple, Hibernate implémente la concurrence optimiste de cette façon. La même chose peut être vraie pour une base de données de type clef-valeur puisque l'Aggregate est souvent sérialisé comme une valeur atomique.

```
public class ProductBacklogItemService ... {
   ...
   @Transactional
   public void planProductBacklogItem(
      String aTenantId, String aProductId,
      String aSummary, String aCategory,
      String aBacklogItemType, String aStoryPoints) {

      Product product =
         productRepository.productOfId(
            new TenantId(aTenantId),
            new ProductId(aProductId));

      BacklogItem plannedBacklogItem =
         product.planBacklogItem(
            aSummary,
            aCategory,

   BacklogItemType.valueOf(aBacklogItemType),
                StoryPoints.valueOf(aStoryPoints));

      backlogItemRepository.add(plannedBacklogItem);
   }
   ...
}
```

Nous avons donc résolu les problèmes d'échec de transaction en les faisant disparaître du modèle. Des instances de *Backlog Item*, *Release* et de *Sprint* peuvent maintenant être crées par des utilisateurs concurrents en toute sécurité. Plutôt convaincant.

Cependant, malgré leurs avantages sur la gestion des transactions, les quatre ***Aggregates*** sont moins pratique du point de vue de l'utilisation par le client. Peut-être serait-il possible d'améliorer le gros ***Aggregate*** pour en éliminer les problèmes de concurrence ? En positionnant à `false` l'option de mapping `optimistic-lock` d'Hibernate, l'effet induit d'échec de transaction disparaît. Il n'y a pas d'invariant sur le nombre total d'instance de `BacklogItem`, `Release` ou de `Sprint` créés, alors pourquoi ne pas laisser les collections grossir et ignorer ces modifications au niveau de la classe `Product` ? Quel serait le coût additionnel de la conservation du gros ***Aggregate*** ? Le problème est que sa croissance pourrait le rendre incontrôlable. Avant d'étudier pourquoi dans le détail, intéressons-nous au conseil dont l'équipe avait le plus besoin :

> Règle : Modéliser les vrais ***Invariants*** dans les ***Consistency Boundaries***

Lorsque l'on essai de découvrir les ***Aggregate*** dans un ***Bounded Context***, il est nécessaire de comprendre les vrais invariants du modèle. Ce n'est qu'avec cette connaissance que l'on pourra déterminer les objets qui devront être regroupé dans un ***Aggregate*** donné.

Un ***Invariant*** est une règle métier à laquelle l'état du système doit se conformer. Il existe plusieurs type de conformité : L'une est transactionnelle, elle est considérée comme immédiate et atomique. L'autre est finale, elle intervient après la stabilisation de l'état du système. Quand il s'agit d'***Invariant,*** c'est de la conformité transactionnelle dont on parle.

Prenons par exemple l'invariant `c = a + b`.

Alors, si `a` vaut 2 et que `b` vaut 3, `c` doit valoir 5. Etant données la règle et les hypothèses, si `c` vaut autre chose que 5, alors un invariant du système est enfreint. Pour s'assurer que c reste conforme, on modélise une frontière autour de ces attributs :

```
AggregateType1 {
  int a; int b; int c;
  operations...
}
```

La logique interne de la ***Consistency Boundary*** assure que tout à l'intérieur vérifie un sous-ensemble des règles métier d'invariance, quelles que soient les opérations effectuées. La conformité de tout ce qui est à l'extérieur de la frontière ne concerne pas l'***Aggregate***. ***Aggregate*** devient synonyme de frontière de la conformité transactionnelle. Dans cet exemple simple, `AggregateType1` a trois attributs de type primitifs, mais un ***Aggregate*** peut comporter des attributs de type référence.

Lorsque l'on utilise un mécanisme standard de persistance, on utilise une unique transaction[3] pour gérer la conformité. Un ***Aggregate*** correctement conçu peut être modifié de n'importe quelle manière requise par le métier et garder ses ***Invariants*** vérifiés, le tout en une seule transaction. Un ***Bounded Context*** correctement conçu modifie dans tous les cas une seule instance d'***Aggregate*** par transaction. Il en ressort que l'on ne peut pas travailler correctement sur la conception des ***Aggregates*** sans analyser les transactions.

Limiter les modifications à un ***Aggregate*** par transaction peut sembler un peu trop strict. Mais il s'agit d'un principe général qui doit pouvoir être suivi dans la plupart des cas. Il est à la base de la raison même de l'utilisation des ***Aggregates***.

[3] La transaction peut être gérée en appliquant le pattern Unité de Travail (Unit of Work)

Puisque les ***Aggregates*** doivent être conçus avec en tête la conformité de l'état du système, l'interface utilisateur devrait consacrer chacune de ses requêtes à l'exécution d'une seule commande sur une instance d'***Aggregate***. Si une requête utilisateur essai de faire trop de choses, l'application sera forcée de modifier plusieurs instances à la fois.

Nous avons donc vu que les ***Aggregates*** servent à délimiter un périmètre de conformité, et non à obtenir un graphe d'objets. Certains ***Invariants*** du monde réel seront plus complexes que ceux que nous avons vus. Malgré cela, les ***Invariants*** courants poseront moins de contraintes au modèle, permettant de concevoir de petit ***Aggregates***.

> Règle : Concevoir de petits ***Aggregates***

Nous pouvons maintenant approfondir la question du coût additionnel qu'il y aurait à garder le gros Aggregate. Même si l'on garantit que chaque transaction va réussir, celui-ci limite les performances et la scalabilité. Au fur et à mesure que notre société va développer son marché, le nombre de *Tenants* va augmenter. Quand chacun d'entre eux va s'investir de plus en plus dans l'utilisation de *ProjectOvation*, ils vont héberger de plus en plus de projets et des artéfacts de management qui vont avec. Il en résultera un grand nombre de ***Products***, ***Backlogs Items***, ***Releases***, de ***Sprints***… Les performances et la scalabilité sont des exigences non fonctionnelles qui ne peuvent être ignorées.

Au regarde des performances et de la scalabilité, que se passe-t-il quand un *User* d'un des *Tenants* veut ajouter un simple Backlog Item à un Product qui a plusieurs années et comporte déjà des milliers de Backlog Items ? Supposons un mécanisme de persistance capable de lazy loading (Hibernate). Nous ne chargerons probablement jamais l'ensemble des Backlog Items, Release et Sprints en une seule fois. Mais pour ajouter un seul Backlog Item à la collection déjà imposante, il faudrait charger des milliers d'instances existantes. Et c'est encore pire si le mécanisme de persistance ne supporte pas la lazy loading. Même en faisant attention à l'occupation mémoire, il faudra parfois charger plusieurs collections. Par exemple lorsque l'on panifie un Backlog Item pour une Release ou un Sprint, tous les Backlog Items et toutes les Releases (ou les Sprints) devront être chargés.

Observons la Figure 3 qui représente plus en détail le gros ***Aggregate***. Ne nous laissons pas abuser par les « 0..* » ; le nombre des associations ne sera pratiquement jamais zéro, et continuera d'augmente avec le temps. On finira probablement par charger des milliers d'objets en mémoire pour effectuer une opération relativement simple. Ceci peut se produire de nombreuses fois en parallèle quand des centaines ou des milliers de *Tenants* auront chacun plusieurs équipes et projets. Et la situation ne fera que s'aggraver avec le temps.

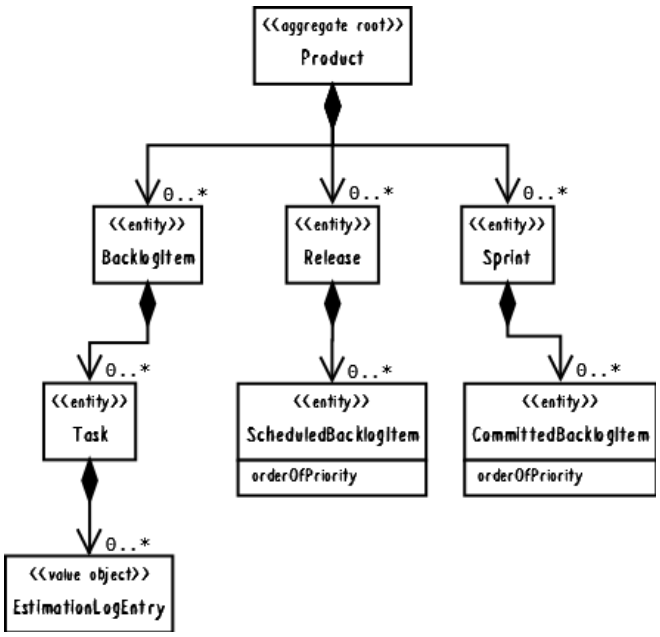

Figure 3 : Avec ce modèle, plusieurs collections de taille considérable sont chargées pour beaucoup de petites opérations simples.

Ce gros ***Aggregate*** ne donnera ni des performances ni des possibilités d'extension satisfaisantes. Il deviendra probablement un cauchemar menant à l'échec. Il était défaillant depuis le début, parce que ce sont de faux ***Invariants*** et le désire de se ramener à une composition sécurisante qui ont guidé la conception, et non la réussite des transactions, les performances et la scalabilité.

Mais si nous devons concevoir de petits ***Aggregates***, que veut dire « petit » ? A l'extrême, on aurait un ***Aggregate*** avec uniquement un identifiant global et un attribut unique, ce qui n'est la recommandation (sauf dans le cas où ce serait réellement ce dont un ***Aggregate*** particulier aurait besoin). En pratique, il faut limiter l'***Aggregate*** à une ***Root Entity*** et à l'ensemble minimal des attributs et/ou de

propriétés de type ***Value***[4] ; c'est à dire ceux qui sont requis et pas plus.

Quels sont ceux qui sont requis ? La réponse est simple : ceux qui doivent rester en accord entre eux, même si les experts métier n'expriment pas la règle. Par exemple, Product a les attributs `name` et `description`. On n'imagine pas que `name` et `description` soient en désaccord. Lorsque l'on change le nom, on veut probablement changer la description. Si l'on change l'un et pas l'autre, c'est probablement que l'on veut corriger un faut ou juste faire que l'un complète mieux l'autre. Même si les experts métier ne considèreront pas qu'il y a une règle, elle existe implicitement.

Que faire si vous pensez devoir modéliser une sous-partie comme une ***Entity*** ? D'abord, demandez-vous si cette sous-partie doit pouvoir évoluer en elle-même dans le temps, ou si elle peut être simplement remplacée quand une modification est nécessaire. Si les instances peuvent être tout simplement remplacées, on choisira un ***Value Object*** et non une ***Entity***. Parfois, des sous-parties de type ***Entity*** sont nécessaires. Mais si l'on y regarde de prêt, beaucoup de concepts que l'on trouve modélisé comme des ***Entities*** pourraient en fait être remaniés en ***Value Objetcs***. Choisir un type ***Value Object*** plutôt qu'une ***Entity*** ne veut pas dire que l'***Aggregate*** devient immuable, puisque la ***Root Entity*** elle-même mue lorsque l'un de ses propriété de type ***Value*** est remplacée.

Il y a un gros avantage à limiter les sous-partie à des ***Values***. Certains mécanismes de persistance pourront sérialiser les ***Values*** directement avec l'a ***Root Entity***, alors que les ***Entities*** auront souvent besoin d'être stockées en un second temps. Le coût est plus élevé avec des ***Entities*** : par exemple les jointures SQL sont nécessaires pour les lire avec Hibernate. Lire simplement la colonne d'une table est plus rapide. Les Value Objects sont plus petits et plus sûrs à utiliser (ils sont moins sujets aux bugs). Comme ils sont immuables, il est plus simple d'écrire les tests unitaires qui les valide.

Sur un projet pour le secteur financier qui utilisait [Qi4j], Niclas [Hedhman] a rapporté que l'équipe avait pu concevoir approximativement 70% des ***Aggregates*** comme des ***Root Entity*** ne contenant que des propriétés de type ***Value Objects***. Les 30% restants contenaient juste deux ou trois entités au total. Ceci ne veut pas dire que tous les models de domaines auront une répartition de 70/30, mais qu'une large partie des ***Aggregates*** peut se limiter à une seule ***Entity***, la ***Root Entity***.

La discussion sur les ***Aggregates*** dans [DDD] donne un exemple dans lequel de multiples ***Entities*** se justifient. Une commande se voit affecter un total maximal admissible, et la somme de toutes les lignes ne doit pas dépasser ce total. Là règle devient complexe lorsque l'on autorise plusieurs utilisateurs à ajouter simultanément des lignes. Aucun ajout individuel ne peut dépasser la limité, mais un ensemble d'ajouts concurrents le pourrait. Je ne répèterai pas la solution ici, mais je veux insister sur le fait que la plupart du temps, les ***Invariants*** du modèle métier sont plus simple à gérer que ceux de cet exemple. Garder cela en tête permet de modéliser les Aggregates avec un nombre minimal d'attributs.

Les petits ***Aggregates*** améliorent les de performances, mais ils sont aussi plus enclins au succès transactionnel. Les conflits empêchant une mise à jour deviennent rares. Le domaine n'aura que rarement de vrais ***Invariants*** obligeant à choisir la solution d'une grande composition d'***Aggregates***. Il sera judicieux de limiter la taille des ***Aggregates***. Si vous rencontrez à l'occasion une vraie règle de conformité, ajoutez éventuellement quelques entités, au besoin une collection, mais continuez à vous obliger à garder le résultat aussi petit que possible.

Se méfier des Use Cases

Les analystes métier jouent un rôle important dans la fourniture des spécifications Use Cases. Comme un travail important est effectué pour l'écriture d'une spécification longue et détaillée, il aura un impact sur beaucoup de vos décisions de conception. Mais il ne faut pas oublier que les Use Cases écrits de cette manière ne sont pas porteurs de la vision de l'équipe dans laquelle les experts métier et les développeurs travaillent en relation étroite. Il faudra faire un travail de réconciliation des Use Cases avec notre modèle, y compris au niveau des choix de conceptions des ***Aggregates***. Un problème typique que l'on

[4] Une propriété de type ***Value*** est un attribut qui référence un ***Value Object***. Je le différencie des attributs simples comme les Strings ou les types numériques, comme le fait Ward Cunningham quand il décrit Whole Value. Voir http://fit.c2.com/wiki.cgi?WholeValue

rencontre est qu'un use case particulier implique la modification de plusieurs instances d'***Aggregates***. Dans ce cas, il nous faut déterminer si l'intention de l'utilisation spécifiée comme un seul Use Case peut être persistée en plusieurs transactions ou s'il faut qu'elle le soit de façon atomique. Si l'on est dans le second cas, il faut se poser des question : Peut importante la qualité de la spécification, elle n'est pas en accord avec les vrais ***Aggregates*** de notre modèle.

Si les limites de l'Aggregate son alignées sur de réelles contraintes métier, et que les analystes métier spécifient ce qui est représenté dans la Figure 4. Si l'on regarde les différentes possibilité d'ordonnancement des requêtes, on voit que dans certains cas deux requêtes sur trois vont échouer.[5]

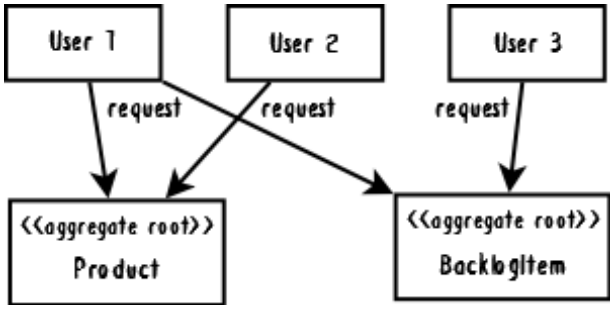

Figure 4 : Des conflits potentiels d'accès aux ressources existent entre trois utilisateurs qui veullent accéder aux deux même instances d'*Aggregates*, causant l'échèque de nombreuses transactions.

Que nous apprend le fait de se retrouver dans cette situation sur notre conception ? La réponse à cette question peut nous amener à redécouvrir notre modèle. Avoir besoin de maintenir en conformité plusieurs ***Aggregates*** peut être une indication que l'équipe a raté un ***Invariant***. Il se peut que vous finissiez par remanier ces différents Aggregates en un nouveau concept avec un nouveau nom qui adressera la nouvelle règle métier. (Evidemment, il est possible que seules des parties des anciens ***Aggregates*** se retrouvent regroupées dans le nouveau)

Il se peut, donc, qu'un nouveau Use Case nous force à repenser un ***Aggregate***. Mais il convient de rester critique : former un nouvel ***Aggregate*** à partir de plusieurs existants peut mener à un nommer un nouveau concept, mais si la modélisation de ce nouveau concept vous mène à un gros ***Aggregate***, elle mènera à tous les problèmes du gros ***Aggregate***.

Quelle autre approche pourrait-elle s'avérer intéressante ? Ce n'est pas parce qu'un nouvel Use Case appèle à maintenir la cohérence en une transaction unitaire que c'est ce que vous devriez faire. Souvent, dans ce type de cas, l'objectif métier peut être atteint avec une *conformité finale* des différents ***Aggregates***. L'équipe doit examiner de façon critique les Use Case et remettre en question ses hypothèses, surtout lorsque les suivre telles quelles sont écrites conduiraient à une conception non satisfaisante. L'équipe peut être menée à réécrire des Use Cases (ou au moins à les ré-interpréter si elle se frotte à un analyste métier peu coopérant). Le nouvel Use Case spécifie une *conformité finale*, ainsi qu'un *délais acceptable de mise en conformité*. C'est un des problème adressé par la seconde partie de cette essai.

Au programme de la seconde partie

La première partie se concentre sur la conception d'un ensemble de petits ***Aggregates*** et de ce qu'ils contiennent. Dans certains cas, il faudra des références et une cohérences entre des ***Aggregates***, en particulier lorsque l'on veut garder ceux-ci petits. La seconde partie de cet essai explique comment les ***Aggregates*** peuvent en référencer d'autres, et comment obtenir une conformité finale.

Remerciements

Eric Evans et Paul Rayner ont effectué plusieurs relectures détaillées de cet essai. J'ai aussi reçu le retour de Udi Dahan, Greg Young, Jimmy Nilsson, Niclas Hedhman, et Rickard Öberg.

Références

[CQS] Martin Fowler explains Bertrand Meyer's Command-Query Separation: http://martinfowler.com/bliki/CommandQuerySeparation.html

[DDD] Eric Evans; Domain-Driven Design - Tackling Complexity in the Heart of Software; 2003, Addison-Wesley, ISBN 0-321-12521-5.

[Hedhman] Niclas Hedhman; http://www.jroller.com/niclas/

[5] Ne pas confondre ceci avec les cas où certains Use Cases décrivent des modifications dans lesquelles plusieurs ***Aggregates*** sont modifiés aux cours de transactions successives, ce qui est authorisé. Une intention utilisateur ne dois pas être confondue avec une transaction. Nous ne parlons ici que des Use Cases qui implique la modification de plusieurs ***Aggregates*** au cours d'une seule transaction.

[Qi4j] Rickard Öberg, Niclas Hedhman ; Qi4j framework ; http://qi4j.org/

Biographie

Vaughn Vernon est consultant vétéran, proposant ses services en architecture, développement, accompagnement et en formation.

Cet essai en trois parties est à l'origine du livre Implementing Domaine Driven Design [AJOUTER L'ISBN ET L'EDITEUR]. Sa présentation au QCon San Francisco 2010 sur le Context Mapping est disponible sur le site de la communauté DDD : http://dddcommunity.org/library/vernon_2010.
Vaughn blogue sur http://vaughnvernon.co/, et vous pouvez le contacter à vvernon@shiftmethod.com