DM7 POCA
Domain-Driven Design 2

Rendu de Team Galaxy: Robin Arnoux, Thierry Chhoa, Kévin Dang, Natacha Guijarro, Rémy Phol Asa, Alexandre Sabri.

1 - Illustrer le principe « For every traversable association in the model, there is a mechanism in the software with the same properties. » en listant, pour chaque association dans votre modèle, sa contrepartie dans l'implémentation.

Association Client -> Devis (one to many) :

“Un client peut simuler et souscrire à autant de devis qu’il le souhaite. On doit pouvoir identifier pour chaque devis quel est le client qui l’a émis et qui y est souscrit (ou non).” (modèle)“

“Une instance de la classe Quote possède un membre customer de type Customer. Ce membre customer est initialisé par l’intermédiaire de TypeORM lors de l’instanciation, la table Quote en base possédant une clé étrangère faisant référence au client ayant émis le devis. A l’aide du repository géré par TypeORM il est possible de retrouver en base et d’instancier des objets pour tous les devis d’un client.” (design)

2 - Prenons pour hypothèse que votre modèle contienne une liste de contrats vendus, et une liste des garanties possibles. Tous les contrats n’ont pas la même liste de garanties. Il y a une relation bidirectionnelle many-to-many entre les 2 : une garantie peut porter sur plusieurs contrats; un contrat peut comporter plusieurs garanties. La relation a-t-elle besoin d'être traversée dans les 2 sens ? Quel design choisissez-vous ?

Un contrat doit pouvoir avoir connaissance des garanties qu’il comporte, cela est indispensable pour permettre plusieurs actions qui sont au cœur du modèle. Entre autres : permettre à un client de consulter toutes les garanties comportées dans un de ses contrats, permettre à l’assureur de déterminer si un sinistre est couvert par un contrat précis et dans quelle mesure, …

Concernant l’autre sens de la relation - permettre à une garantie de savoir sur quels contrats elle porte - elle est à notre sens nécessaire - et donc pour nous **la relation doit pouvoir être traversée dans les deux sens**.

En effet, si une garantie ne peut pas traverser la relation et connaître les contrats qui incluent, on ne peut pas ajouter naturellement de nombreuses fonctionnalités. Par exemple, si en tant qu'assureur on souhaite faire évoluer une garantie, alors tous les contrats qui la comprennent vont devoir être renégociés, il faut donc pouvoir identifier tous les contrat qui l'incluent pour notifier les clients souscripteurs de ses contrats et pouvoir rediscuter le contrat avec eux.

Si cette traversabilité dans les deux sens ne fait pas partie du design alors il faudrait parcourir l'intégralité de nos contrats pour identifier ceux qui comportent une garantie spécifique ce qui n'est pas souhaitable.

Design choisi :

Dans l'entité client (Customer) conserver une liste de toutes les garanties du contrat. Ce lien direct fait sens car on veut pouvoir accéder immédiatement aux clauses du contrat pour les afficher au client, et déterminer la couverture sur un sinistre, … Cette liste sera instanciée par TypeORM lors de la reconstitution de l'objet persisté en base : On ajoute une relation en base qui spécifie les association garantie / contrat via des clé étrangères vers leurs id.

On peut imaginer que l'entité contrat aurait également une liste de tous les contrats qui implémente , instanciée de la même manière, mais on peut vite alors se retrouver avec énormément de données référencées. Il semble plus pertinent plutôt d'utiliser cette même relation en base mentionnée plus haut via un repository managé par TypeORM pour récupérer tous les contrats qui incluent cette garantie (via son id).

3 - Quels sont les entités, les « value objects » et les services de votre modèle ?

Les entités de notre modèle sont les clients (Customer) , les devis (Quote), les sinistres.

Les "value objects" sont la localisation de l'utilisateur, le modèle de son véhicule (SpaceshipModel), la classe de son véhicule (SpaceshipClass) , son email, son numéro de téléphone.

Les services sont l'envoi de notification par mail, enregistrement d'un devis pour un utilisateur, suppression d'un devis, simulation d'un devis, déclaration d'un sinistre etc...

4 - Pouvez-vous donner un exemple d'agrégat dans votre modèle ?

Un exemple d'agrégat dans notre modèle est l'entité Customer qui détient l'entité Quote selon les critères suivant :

- L'entité Customer possède une identité unique;

- L’entité Customer est ultimement responsable de ses Quotes;
- On ne peut atteindre les Quotes que via le Customer concerné;
- Les Quotes peuvent être utilisés par des objets externes à l’agrégat mais ne garde pas une référence vers elle.
- Lorsqu’on supprimer un Customer, tous les Quotes lui appartenant le sont également.

5 - Donner un exemple d'invariant satisfait à l'intérieur d'un agrégat. Donner un exemple d'invariant global à l'application (qui n'est pas localisé dans un seul agrégat).

Invariant satisfait à l’intérieur d’un agrégat : Un Quote appartient toujours au client (entité Customer) qui l’a simulé et y a souscrit (ou non) et ce client ne peut pas être changé.

Invariant global à l'application : Nous ne sommes pas sur si la question demande un invariant satisfait par tous les agrégats ou un invariant plus global encore. Dans le premier cas, nous pourrions dire lorsqu'un agrégat est détruit, tous les objets qu'il contient le sont également. Dans le second cas, un client a accès à davantage de fonctionnalités qu'un visiteur anonyme

6 - Comment votre design permet-il d'annuler un contrat ?

Bien que cela ne soit pas actuellement implémenté dans notre design, voici comme nous avons décidé de modifier notre design pour permettre l’annulation d’un contrat.

Un client peut demander l’annulation d’un de ses contrats actifs à tout moment. Le contrat ne sera pas immédiatement annulé mais une demande d’annulation portant sur le contrat sera alors enregistrée. (On peut également envisager de renseigner le statut de cette demande - ex : en cours de traitement, traitée, … - Dès lors cette demande est traitée : en consultant les paiements effectué par le client relatif à ce contrat on peut alors le remboursement de la différence sur la dernière période de paiement qu’il a réglé sur la période où il ne sera désormais pas couvert (par exemple si le client paye à l’année et demande d'annuler son contrat après 3 mois on lui remboursera les 9 mois restants). Une date d’annulation sera enregistrée dans le contrat - cela empêchera de couvrir des sinistres qui seraient déclarés et auraient lieu après la date d’annulation.

Si le contrat à été souscrit il y a moins de 14 jours, l'intégralité de la somme payée est remboursée au client.

Une fois annulé le contrat est conservé mais sous le statut annulé afin de permettre au client et à l’assureur de consulter l’historique des paiements et remboursements ayant eu lieu dans le cadre de ce contrat.

7 - Considérons la règle métier suivante : « Un contrat est valide s'il a été signé et payé ». Comment votre design prend-il en compte cette règle métier ?

Actuellement, notre design empêche simplement au client d'arriver à l'étape de validation du contrat - et donc d'un point de vue technique sa création en tant que contrat valide et sa persistance en base - qu'une fois le contrat signé et validé. Cependant on peut souhaiter permettre plus de flexibilité à ce niveau - un paiement par virement par exemple peut prendre quelques jours à être validé - si l'on veut s'assurer d'une vérification humaine de la signature (et on peut imaginer qu'il serait demandé de fournir des scans de ses papiers d'identité par exemple) alors il y aura un certain délai avant que les deux étapes ne soient validés - délais pendant lequel on veut malgré tout garder une trace du contrat comme étant en cours de paiement / de signature pour ne pas perdre l'information que le client a souhaité souscrire à un devis ni la progression actuelle des étapes de paiement et de signature.

Pour pouvoir prendre en compte cette règle métier, il est donc possible d'ajouter un membre statut à l'entité contrat comme discuté plus haut sur la question sur l'annulation on pourrait donc avoir un statut en cours de validation - et implémenter l'invariant suivant : tant que le paiement et la signature n'ont pas eu lieu alors le statut du contrat ne peut pas être validé. Dès lors que paiement et signature sont confirmés le statut du contrat devient validé.

8 - Dans le cas où les objets sont persistés en base de données, quelles sont les deux étapes du cycle de vie d'un objet où une factory peut être utile ?

Une factory peut être utile au début du cycle de vie d'un objet lors de sa création. Elle permet de ne pas laisser au client la responsabilité d'assembler un objet en particulier lorsque celui-ci à de nombreuses dépendances vers d'autres types et/ou nécessite l'instanciation d'objets supplémentaires. Cela permet de plus de s'assurer que l'on crée toujours des objets qui sont cohérents avec le modèle et le design et de gérer facilement le comportement que l'on souhaite implémenter lorsque l'on tente de créer un objet incohérent via la factory.

Dans un second temps une factory peut être utilisée au milieu du cycle de vie de l'objet lorsqu'on souhaite reconstituer un objet déjà persisté en base de données depuis les informations renvoyées par la base. Cette tâche peut être directement celle du repository - qui assure la communication avec la base de donnée et le mapping data-objet - qui tient alors dans les faits un rôle de factory mais on peut aussi décider de déléguer ce rôle à une factory isolée du repository à laquelle il fera appel.

Sources : Eric Evans - Domain Driven Design (Chapitre 6)

9 - Lorsqu'un objet doit être persisté en base de données, il est nécessaire d'implémenter la sérialisation de cet objet. Où est-il envisageable de placer ce code ? (plusieurs possibilités) Quel endroit vous semble le plus judicieux ?

Pour pouvoir sérialiser un objet, il peut être envisageable d'implémenter le code responsable de la sérialisation d'un objet directement dans sa classe. Cela à l'avantage de permettre de s'assurer plus simplement que la sérialisation évolue en même temps que l'implémentation de la classe et reste cohérente avec elle. Certains language proposent des fonctionnalités rendant ce choix assez intuitif (ex : implémenter l'interface Serializable en java).

Cependant cela peut vite occuper beaucoup de place dans le code. Si on suppose de plus que l'on veut proposer plusieurs format de sérialisation (xml, json, …) en gardant toute l'implémentation dans les classes on va vite avoir du code dupliqué et on il y aura parfois plus de code de sérialisation que de code propre à l'implémentation de la classe elle même.

Une autre option qui nous semble plus judicieuse est de découpler l'implémentation de la sérialisation des classes - des informations nécessaire à la sérialisation de certains membres pourraient être ajoutés quand cela est nécessaire dans les classes mais la logique de sérialisation/désérialisation (et donc les choix de différents formats par exemple) devraient en être séparés. Dans certains langages comme java, cela est implémentable très intuitivement tandis que ce choix est moins évident dans d'autres (par exemple C++ ne dispose pas de mécanisme dit de mirroring, l'automatisation de la sérialisation comme en java n'est donc pas possible).