Rapport 2

Équipe B6

I. Scénarios non réalisés

Tous les scénarios n'ont pu être réalisés, par manque de temps. Certains scénarios

abandonnés concernent des comportements mineurs de fonctionnalités implantées. C'est le cas des scénarios 8 et 30, qui demandent la modification d'un compte utilisateur et l'enregistrement dans celui-ci de coordonnées bancaires. Le scénario 15, qui demande l'envoi d'un courriel après l'achat, est dans une situation semblable.

D'autres scénarios se rapportent à des fonctionnalités en cours d'implantation, mais dont plusieurs fonctions importantes sont toujours manquantes. Par exemple, les scénarios 10, 11, 13 et 14 sont tous liés au panier d'achat, mais n'ont pu être réalisés.

Le plus grand groupe de scénarios non réalisés est celui réalisant la fonctionnalité de revente de billets. Aucun travail n'a été fait sur cette fonctionnalité. Les scénarios 19, 20, 22, 23, 27 et 28 restent donc à faire.

Finalement, les scénarios restant sont indépendants les uns des autres. Ils concernent des fonctionnalités mineures, non essentielles pour le fonctionnement du système, mais pouvant améliorer l'expérience de l'utilisateur. Cette catégorie inclut le scénario 18, qui permet aux utilisateurs d'acheter un groupe de places assises; le scénario 29, qui permet de retirer de l'inventaire les billets non-vendus des matchs annulés; le scénario 32 qui suggère à l'utilisateur des billets basés sur certains qu'il a déjà acheté; le scénario 33, qui permet de visualiser l'emplacement du siège associé au billet dans le stade.

Les autres scénarios ont tous été réalisés. Il s'agit de scénarios relatifs à la consultation de données, à la recherche de billets, à l'achat de billets, à la gestion des comptes utilisateurs et à l'ajout de données dans le système par l'administrateur.

II. Architecture logicielle

Le style architectural principal de notre système est l'architecture en couches. Notre système dispose de 3 couches au sens strict, c'est à dire au sens où la couche n reçoit des instructions de la couche n+1 et en envoie à la couche n-1. Il s'agit, dans l'ordre, de la couche présentation, de la couche applicative et de la couche de persistance.

La couche présentation suit une forme architecturale de style web-mvc. Elle est chargée de la génération de vues HTML à envoyer aux clients et de la réception des requêtes de ceux-ci. Ces contrôleurs redirigent ensuite les requêtes au service applicatif approprié. La technologie employée est Spring-MVC.

La couche applicative est principalement composée des services applicatifs et des repositories. Les repositories génèrent les objets du domaine qui encapsulent la logique d'affaire. Ils le font en s'adressant à la couche de persistance chez qui ils récupèrent les données. Les services applicatifs ont pour responsabilité de réaliser les différents cas d'utilisation du système. Ainsi, ils coordonnent les appels au domaine. À la fin d'un service, le repository retourne les données à la couche de persistance pour qu'elles y soient sauvegardées.

La couche de persistance s'assure de la sauvegarde des données dans un fichier XML. Sa responsabilité est de charger les données appropriées en mémoire, d'enregistrer les modifications qui sont faites aux données et de les réécrire sur le disque au moment opportun.

Notre domaine d'affaire est enrobé dans la couche applicative d'une manière qui rappelle le style architectural en onion[1]. Ce style architectural place le domaine au centre du système, puis l'enrobe avec les préoccupations de l'infrastructure. Dans notre cas, c'est la couche applicative qui se charge de recevoir et d'envoyer les commandes aux autres composantes du système, rendant le domaine d'affaire absolument indépendant de celles-ci.

Finalement, notons que nos services sont découpés selon le principe du CQRS (Command-Query Responsability Segregation)[2]. Les services qui opèrent en mode

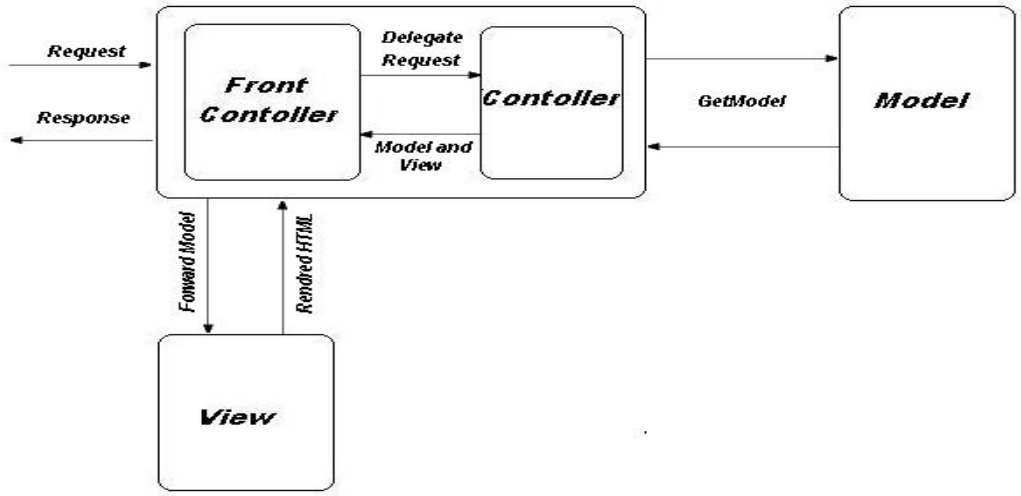
[1] http://jeffreypalermo.com/blog/the-onion-architecture-part-1/
[2] http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/

query (lecture des données pour l'affichage dans les vues), ne s'encombrent pas du modèle d'affaire, qui est inutile dans de tels cas de figure. Ils récupèrent directement les données des Daos sans passés par les repositories. Les services qui opèrent en mode command (modification des données du système) manipulent les données à travers les objets d'affaires qui sont mis à leur disposition par les repositories. De cette manière, les données sont toujours modifiées en respectant les règles d'affaire applicables (utilisation du domaine pour les commands) et les objets d'affaires n'ont pas à désencapsuler leurs données avec des accesseurs (non utilisation du domaine pour les query).

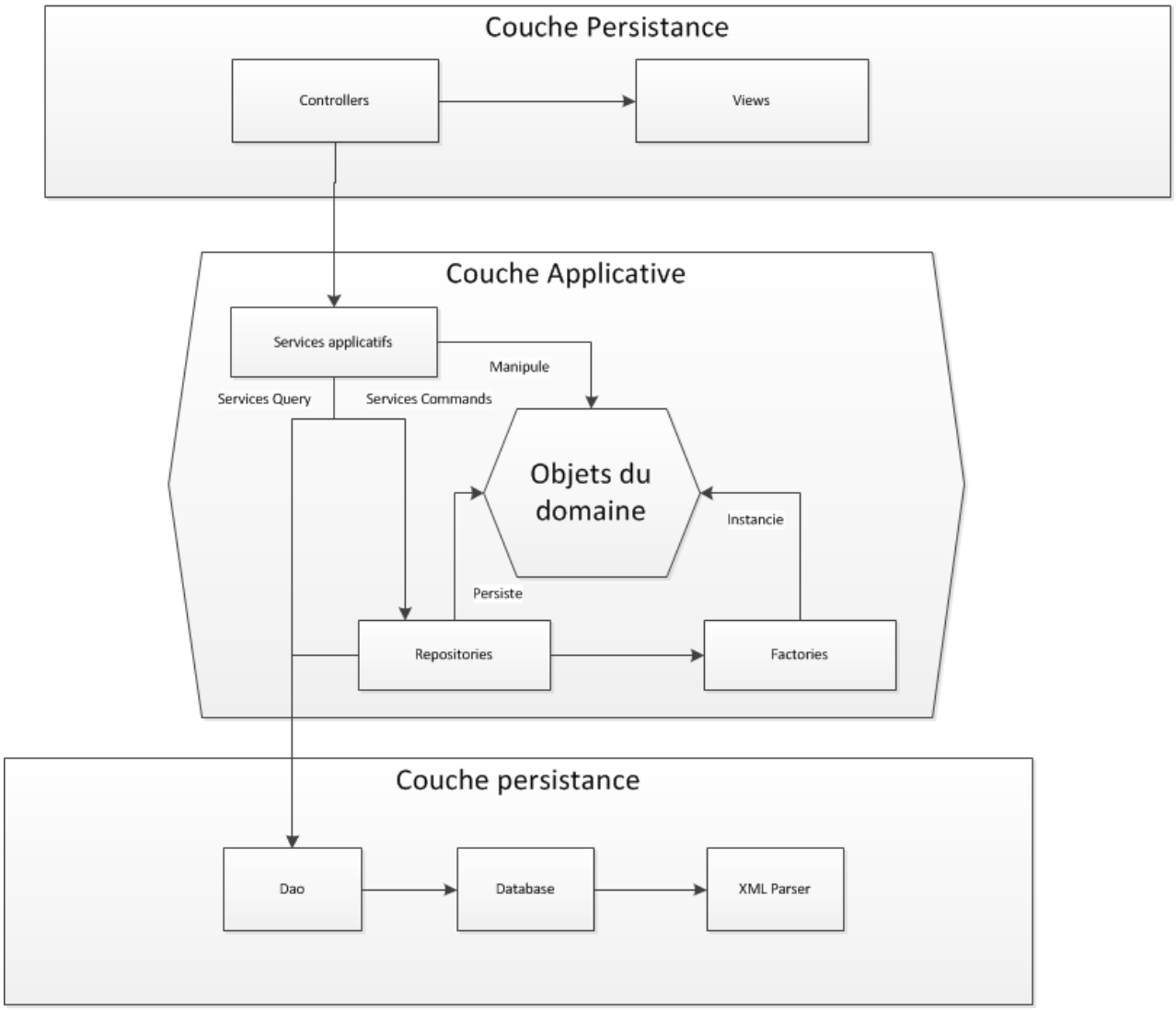
Les pages qui suivent illustrent les concepts discutés avec différents schéma.

Architecture

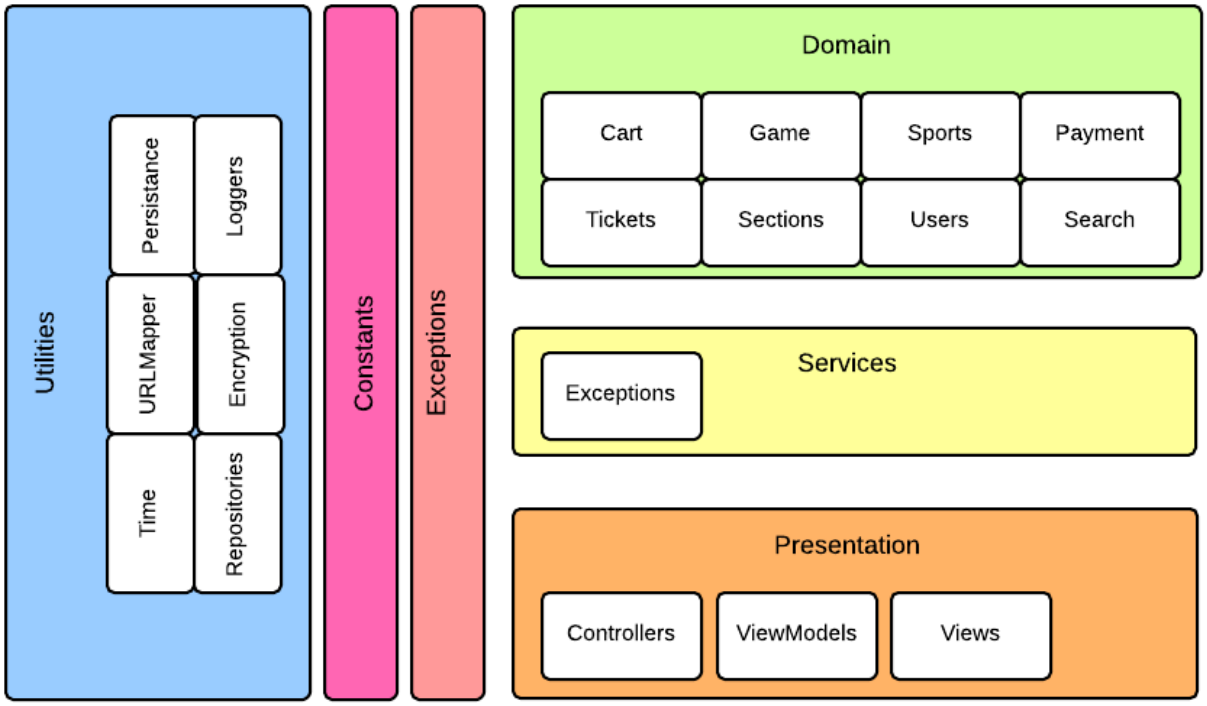
Spring MVC



Architecture en couches



Répartition des couches de *packages*



Suite à l'itération 3, nous avons remarqué lors de l'ajout de fonctionnalités, que notre architecture telle qu'elle était, nous prédisposait à des cycles de dépendances. Nous avons donc procédé à une restructuration de nos *packages* tels qu'il fût recommandé afin de regrouper les éléments par «composant du domaine» (Cart, Sports, Tickets, etc.) et non par «concept technique» (tel que factories, mapper, etc.).
Cette nouvelle façon de revoir l'architecture nous a permis de remarquer que certains composants transcendaient l'architecture verticalement, dans certains cas de façon volontaire tel que pour le *package* «Utilities» et involontairement dans d'autres, tel que pour «Constants» et «Exceptions».

III. Patrons de conception

Cette section discute des patrons de conception utilisés dans le projet.

Adapteur

Afin de simplifier les opérations sur les fichiers XML, la classe XmlExtractor a été mise en place afin d'encapsuler la logique de recherche et d'extraction de noeud du fichier XML. Elle retourne des objets simples, faciles à convertir avec un minimum de code.

Factory

Utilisée dans deux contextes différents. Premièrement, dans les *repository* afin d'y instancier les objets du domaine, qu'ils soient nouveaux ou récupérés de la persistance. Deuxièmement, ils sont utilisés par la couche web afin de construire les *ViewModel* qui seront utilisés par les vues.

Repository

Les *repositories* sont utilisés par les services de *Commands* afin d'obtenir les objets d'affaire qu'ils manipulent. Le *repository* fait le pont entre la couche applicative et la couche données. Chaque entité du domaine dispose de son propre *repository*.

Multi-Singleton

Bien qu'un mauvais patron de conception, l'utilisation du singleton est parfois nécessaire. Aussi, nous avons réservé son emploi à un seul endroit où il nous semblait essentiel. Nous l'utilisons pour obtenir l'unique instance de l'objet base de données de la persistance XML et ainsi avoir plusieurs *DAO* pouvant utiliser la même source de données.

Notre singleton est «multiple», car nous utilisons deux sources de données: un fichier XML stocke les informations relatives aux comptes utilisateurs et un autre celles relatives aux sports, matchs et billets. Lorsqu'ils demandent accès à une source de données, les *DAO* spécifient le nom de la source qu'ils veulent utiliser. Si le besoin s'en fait sentir, l'ajout de nouvelles sources de données est facile.

State

Le patron *state* a été utilisé pour conserver l'état interne des objets d'affaires games et tickets. En effet, ces deux objets, pour pouvoir être sauvegardés, doivent être assignés une et une seule fois à leur objet parent (au sport dans le cas de la partie et à la partie dans le cas du billet). Le *state* enregistre l'état d'assignation. Si une seconde assignation est tentée, celle-ci échoue. Si une sauvegarde est demandée sur l'objet sans que celui-ci n'ait été assigné, elle échoue également.

Façade

Nos services applicatifs offrent des services aux contrôleurs, masquant la

complexité du système se cachant derrière eux. De cette manière, ils se comportent comme des façades.

Template method

Des objets CreditCard sont utilisés afin de valider les informations de paiement fournies par l'acheteur. Les différents types de carte de crédit effectuent les mêmes traitements mais avec des implémentations différentes pour valider ces informations de paiement. Il a donc été décidé d'utiliser une *Template method* afin de factoriser le code.

IV. Utilisations de l'aspect

Journaliser les ventes

À chaque transaction, des billets sont retirés du système et nous croyons pertinent d'enregistrer ces transactions en dehors du système. Puisque la persistance de ces transactions revient au même principe qu'une journalisation conventionnelle, nous avons décidé d'utiliser la POA (Programmation Orientée Aspect) pour effectuer cette partie du traitement. Nous avons utilisé la POA pour intercepter la méthode du ticketService qui retire les billets du système. Nous utilisons les tickets passés en paramètre afin de les inscrire dans un fichier texte à l'aide de log4j.

De cette manière, l'historique des ventes pourra être consulté à l'externe et le traitement de ces transactions est transparent dans le ticketService.

Journalisation dans les contrôlleurs

La POA a été utilisée à deux fins dans le code des contrôleurs : pour la journalisation des exceptions, et pour la journalisation de certains de ses comportements.

La journalisation des exceptions apparait comme l'utilisation la plus classique de l'AOP dans un tel projet. Un point de coupure est inséré à chacune des méthodes qui comporte des exceptions. Si la méthode a généré une exception lors de l'exécution, l'aspect retrouve l'exception en question et produit une journalisation via log4j. Si aucune erreur n'est lancée, aucune journalisation n'est produite.

Nous avons jugé que certaines méthodes des contrôleurs méritaient d'être

journalisées de par leur nature critique. Par exemple, lorsqu'un administrateur ajoute des parties ou des billets, une journalisation est produite par l'aspect correspondant. De cette manière, il est aisé de retracer le cours des évènements qui ont précédé une éventuelle erreur. Cette journalisation peut aussi être utilisée à des fins de consultation.

Le «transfert» de ce code de journalisation vers un aspect rend le code des contrôleurs beaucoup plus simple et épuré, et donc beaucoup plus facile à travailler et à maintenir.

Gestion de l'administration

Suite à une rétrospective que nous avons faites sur notre section d'administration du site web, nous avons remarqué la présence redondance de code lors de la vérification des permissions des usagers.

Ainsi, nous avons décidé d'incorporer l'aspect «sécurité» dans notre code afin de valider les permissions des usagers. Nous avons donc utilisé pour ce faire un point de coupure sur toutes les méthodes contenues à l'intérieur du *package* administration. Pour chaque appel fait à une méthode d'administration, nous avons fait un *around* sur celle-ci afin de vérifier dans la session que l'utilisateur est bien un administrateur et sinon, le rediriger à une page d'erreur.

De cette façon, nous vérifions l'identité de l'utilisateur avant même que la méthode soit exécutée. Ce fonctionnement ressemble beaucoup à la façon dont Spring Security semble fonctionner.

V. Pistes d'amélioration de la conception

Suite à la restructuration de notre architecture, nous avons remarqué que la façon dont nous avions fait notre architecture précédemment entraînait certains cycles de dépendances sans que nous pouvions le remarquer. Nous avons donc dû trouver une façon d'améliorer notre architecture tout en ayant pas de cycle de dépendances.

Afin d'éviter ces cycles, nous avons eu à faire certains compromis tel que de créer des *packages* verticaux transcendants notre application tel que «Exceptions» et «Constants» afin que les couches de notre applications puissent y avoir accès.

Ceci a engendré de la division dans notre code où certains *packages* tel que «Exceptions» qui pouvaient rester localisés (comme dans le *package* «Services») le sont restés afin de préserver le plus possible le style en couche.

Avec plus de temps, il aurait sans doute été préférable d'améliorer notre architecture afin d'éviter que de telles choses soient présentes.

VI. Installation

1. Prérequis

Afin de pouvoir installer et exécuter le projet, il est nécessaire d'installer Git et Maven sur le poste de travail.

2. Clonage et exécution

Les commandes suivantes sont à entrer dans un Terminal afin de cloner et d'exécuter le projet :

```
git clone -b remise_finale
https://github.com/Archi2013/ulaval-sports-tickets.git
cd ulaval-sports-tickets/
mvn jetty:run
```

Enfin, ouvrir un navigateur web et aller à l'adresse : http://localhost:8080 .

3. Utilisation

Sur la page d'accueil se trouve trois lien. Le logo ramène à la page d'accueil peu importe où on se trouve sur le site.

Le lien «Se connecter» répond au scénario 1 et 2.

Le lien «Sports» redirige vers la liste des sports, la vue qui permet de répondre aux scénarios d'utilisation 3, 5 et 6. Il permet aussi de vérifier l'ajout d'un match pour le scénario 35. Si vous êtes connecté, vous pourrez acheter des billets en consultants les billets disponibles pour un match X. L'achat vous dirigera automatiquement vers le panier d'achat. Cela répond au scénario 12 et 17.

Le lien «Recherche» permet d'accéder a une page de recherche de match en fonction de ces préférences. Si vous êtes connecté, vous pourrez sauvegarder celle-ci. Cela répond au scénario 4 et 17.

Lorsque vous êtes connecté avec un utilisateurs X, vous aurez accès a votre panier d'achat qui contiendra vos achat non complété. Vous pourrez alors procédé jusqu'à l'achat de vos billets à l'aide d'une carte de crédit. Cela répond au scénario 9, 7, 16 et 34.

Quand connecté avec un compte administrateur [admin / admin], un lien supplémentaire «Administration» sera visible. Il permet l'ajout des matchs et des billets. Cela permet de répondre au scénario d'utilisation 25 et 26.
N.B. : Pour l'ajout de billets de type «avec siège», il faut que le nom de section soit «Section 100» ou «Section 200».

Annexe I

Scénario 35

Intitulé : En tant qu'administrateur, je peux ajouter un match au calendrier d'un sport afin de pouvoir rendre des billets disponibles pour ce match.

Critère A: L'administrateur peut choisir d'ajouter une partie.
Critère B: L'administrateur peut entrer les détails d'une partie (date, sport, équipes....).
Critère C: La partie est créée et mémorisée par le système, sans aucun billet au départ .