Université de Rennes

istic Informatique Électronique

TD/TP de Contrôle continu

L'objectif de ces TD et TP est d'appliquer une approche complète de modélisation en mode projet, partant d'un cahier des charges réaliste et allant jusqu'à une application Web utilisant le framework OpenXava.

1 Cahier des charges : Paris en ligne

Le système doit prendre en charge différents types de paris sportifs, dans un premier temps football, basket, tennis et courses de chevaux. Différentes options sont prises en charge pour différents sports, et sont déterminées par les règles de chaque sport. Par exemple, des options différentes sont disponibles pour les matchs de football qui ont deux mi-temps, par rapport aux matchs de tennis qui ont entre deux et cinq sets, par rapport aux courses de chevaux. Les informations en temps réel sur tous les événements sportifs en cours à un moment donné est fournie par un certain nombre sources de données de tiers qui sont mises à jour en permanence.
Le système permet de faire ce que nous appelons un pari simple pour chaque sport. Par exemple, pour les matchs de football, un parieur peut choisir l'issue générale du match : qui sera le vainqueur ou si le match se terminera par un match nul ; en revanche, les matchs de tennis, de basket-ball ou les courses de chevaux ne peuvent généralement pas se terminer par un match nul. Dans un premier temps, les paris seront pris exclusivement avant le début de l'événement.
Au-delà des paris simples, chaque sport permet de prévoir une série d'autres aspects d'un événement sportif. Ceux-ci sont pris en charge par ce que nous appelons les paris avancés. Les chances de gagner ces paris avancés sont généralement plus faibles et nécessitent une gestion du risque plus sophistiquée, à la fois avant et pendant la compétition. Un type de pari avancé consiste à deviner le score exact d'un match. Un autre consiste à deviner le score à des intervalles spécifiques du match (par exemple, la mi-temps d'un match de football ou les sets individuels d'un match de tennis). Les extensions aux paris simples peuvent en outre impliquer les écarts (par exemple, une équipe de football doit gagner par un certain nombre de buts).
Les paris sont placés en ligne par un parieur, qui bénéficie au moment de son inscription d'un capital initial de 10,000 jetons. La manière de recharger ces jetons sera définie ultérieurement. Le parieur sélectionne un événement sportif, un type de pari, le résultat parié, le montant qu'il veut parier en jetons, et soumet ce ticket au système, qui débite son compte du montant du pari. En cas de réussite du pari, le parieur sera crédité de sa mise multipliée par la cote, obtenue pour chaque évènement par un algorithme qui sera fourni dans un second temps.
Le bookmaker détermine quels événements, et leurs variantes, seront proposés par le système. Il définit tous les paramètres relatifs à ces événements, tels que ceux qui sont utilisés pour déterminer les cotes pour chaque événement, les écarts sur les scores, etc. Le bookmaker peut limiter le montant maximal d'un pari, plafonner les gains pour un pari réussi, etc.

2 Etude à réaliser (par binômes)

Modéliser ce système avec UML en respectant la méthode proposée en cours pour analyser un cahier des charges. Les diagrammes UML demandés sont ceux du niveau étude des besoins/analyse et concernent :

- les cas d'utilisation;
- un diagramme de classes avec quelques diagrammes d'objets associés si nécessaire;
- pour chaque cas d'utilisation, un diagramme de séquence système nominal, et des diagrammes de séquence exceptionnels pouvant servir de test système
- des diagrammes d'états pour les classes pour lesquelles cela est pertinent

Utiliser l'outil  https://dotuml.com/index.html pour mettre au propre vos diagrammes. Vous pouvez utiliser DotUML directement en ligne, ou bien comme un plug-in de VS Code. Rendez les diagrammes obtenus à votre encadrant de TD selon ses instructions.

3 Réalisation pratique avec OpenXava (par binômes)

1. Identifiez un petit sous-ensemble de votre modèle (typiquement lié à un cas d'utilisation simple du point de vue d'un parieur) et traduisez-le en Java dans le framework OpenXava, afin d'obtenir une ébauche d'application web déployée en local sur votre poste de travail. On s'appuiera sur le TD précédent pour ce qui est de la mise en œuvre des associations.

2. Sur la base des descriptions textuelles DotUML, écrivez un générateur de code Java pour OpenXava. Testez sur l'exemple de la question précédente que vous obtenez bien le résultat souhaité.

3. Appliquer votre générateur OpenXava à l'ensemble de votre projet pour obtenir un POC d'une application de paris en ligne du point de vue d'un parieur (le bookmaker et les sources de données seront dans un premier temps simulés). *NB : si vous n'avez pu mener à bien la question précédente, faites ce travail à la main*. A ce stade vous avez tout à fait le droit de modifier votre diagramme d'analyse si cette étape de génération de code vous a permis d'y identifier des problèmes.

4. Préparez une petite démonstration de votre projet pour votre encadrant de TP.

5. Rendez le résultat de l'ensemble de ce travail (modèles, générateur de code, code) à votre encadrant selon ses instructions (accès à votre dépôts GitLab, envoi d'un zip, etc.).

NB : cette série de 3 TD et de 4 TP constitue un projet **noté** (contrôle continu) dans lequel vous devrez gérer votre temps de manière autonome pour arriver au résultat. Nous vous recommandons toutefois un mode assez agile avec 3 ou même 4 itérations.

Annexe 1 : Exemple de code OpenXava

```
package com.yourcompany.invoicing.model;

import javax.persistence.*;
import org.openxava.annotations.*;
import lombok.*;

@Entity  // This marks Customer class as an entity
@Getter @Setter // This makes all fields below publicly accessible
public class Customer {

    @Id  // The number property is the key property. Keys are required by default
    @Column(length=6)  // The column length is used at the UI level and the DB level
    int number;

    @Column(length=50)  // The column length is used at the UI level and the DB
level
    @Required  // A validation error will be shown if the name property is left
empty
    String name;

}
```

Annexe 2 : OpenXava

OpenXava est un des outils que vous allez devoir utiliser en TP. Il vous permettra de définir des interactions entre les différentes classes métier sans avoir à les implémenter. Ci-dessous sont proposés quelques pointeurs pour vous aider à prendre en main l'outil.

Etape 1 : Lancement d'OpenXava

Vérifier que le framework n'est pas disponible sur les machines de l'ISTIC (habituellement dans l'espace share) ; sinon, vous pouvez télécharger l'application directement sur le site : https://www.openxava.org/

Il vous faudra remplir un petit formulaire avant de pouvoir accéder au téléchargement de l'application.

Une fois téléchargée sur votre espace personnel, vous pourrez lancer le framework.

L'outil est en fait un IDE Eclipse (un peu comme VS Code mais très utilisé dans la communauté Java principalement puisque beaucoup d'outils sont développés pour venir s'interfacer avec et aider au développement) modifié qui permet d'utiliser des annotations dans le code qui seront pris en compte lors de la compilation pour générer du code automatiquement.

Par exemple, en utilisant les annotations **@getter** et **@setter**, les méthodes de modifications et d'accessibilités des attributs seront automatiquement générées pour l'ensemble des attributs.

Etape 2 : Prise en main et première application

Nous allons nous baser très rapidement sur l'exemple fourni dans le getting-started (https://www.openxava.org/OpenXavaDoc/docs/getting-started_en.html) afin de lancer un projet OpenXava.

Le principe est simplement de créer 2 classes métier : un client, qui sera défini par un numéro de client servant d'identifiant et un nom, et un produit, défini par un numéro servant d'identifiant et une description.

Le code du client vous est fourni en (vous n'avez plus qu'à copier/coller le code dans une classe Java).

A partir de là, vous pouvez lancer la compilation et aller voir ce qu'il se passe dans l'application web générée.

La dernière ligne de la console vous donne le moyen d'accéder à l'application (par défaut dans un navigateur web http://localhost:8080/*<nom_de_votre_projet>*). Vous pourrez y enregistrer de nouveaux clients. Vous pouvez maintenant stopper cette application (le bouton rouge de la console dans l'interface graphique d'OpenXava).

Ce qui va nous intéresser beaucoup plus ici ce sont les annotations utilisées.
Tout d'abord **@Entity** est l'annotation à utiliser sur toutes

les classes qui seront gérées par OpenXava. Les classes précédées d'un **@Entity** seront visibles depuis l'interface web de l'application.

Les 2 annotations citées précédemment **@Getter** et **@Setter** permettent de générer automatiquement les méthodes associées (mise à jour et consultation) aux attributs.

Ensuite, plus spécifiquement lié à l'affichage et la gestion en interne des différents éléments :

L'annotation **@Id** permet la gestion en interne de l'application web pour pouvoir aller chercher les différents objets qui ont été créés.
L'annotation **@Column** avec son paramètre length permet simplement de spécifier l'affichage dans l'interface web.
Enfin l'annotation **@Required** force l'utilisateur à définir cet attribut pour pouvoir créer son objet.

Etape 3 : Mettre en place les associations des exercices précédents

Exerice 1 :

Vous utiliserez l'annotation **@ManyToOne** (avec l'option fetch=FetchType.LAZY).
L'annotation en elle-même permet d'aller chercher tous les éléments de la classe A et d'associer un seul par objets instances de la classe B.
Vous pouvez relancer l'application et expérimenter sur ce qui peut être fait ou non.

Une fois l'application lancée, vous pouvez créer des objets A et B mais… :
* Pouvez-vous supprimer un objet de la classe A qui n'a pas de rôle b associé ?
* Pouvez-vous supprimer un objet de la classe A à qui un objet de la classe B est associé ?
(en fait, c'est un choix qui a été fait pour être sûr que les utilisateurs ne fassent pas n'importent quoi, il suffirait de voir si pour tout objet de la classe B, le a est l'objet que l'on veut supprimer si c'est le cas, on le met à null sinon on ne change rien).

Exercice 2 :

Vous utiliserez l'annotation **@ManyToOne** (avec les options fetch=FetchType.LAZY et optional=false).

Les objets de la classe A (respectivement B) peuvent-ils exister sans objet de la classe B (respectivement A) associé ?

Est-ce que cette application respecte le modèle proposé ci-dessus ?
Que faudrait-il changer pour y arriver ? Essayez et voyez si vous réussissez.

Est-ce que l'on peut avoir plusieurs instance de B associées à la même instance de A ?

Exercice 3 : La composition

Cela se met en place dans OpenXava par l'utilisation de 2 annotations majoritairement :

@Embeddable à la place de l'annotation **@Entity** sur la classe qui fera parti de l'autre,

et un attribut annoté **@Embedded** dans la classe qui mènera la création et la destruction des objets.

Essayez et remarquez que la classe B n'existe pas dans le panneau à gauche. Vous ne pourrez pas consulter les instances qui ont été créées de manière indépendantes puisqu'elles doivent toutes être associées à un objet de la classe A.

Vous pouvez remarquer également qu'il n'y a pas d'identifiant (annotation **@Id** sur un attribut de B), vous pouvez alors réutilisez les mêmes noms dans les différents champs, à chaque fois, c'est bien une nouvelle instance qui sera créée et gérée de manière indépendante.

Pour en apprendre d'avantage sur OpenXava et ses possibilités, n'hésitez pas à consulter les pages des leçons (qui commencent juste après le Getting Started).

La première se trouvant à l'adresse suivante :
https://www.openxava.org/OpenXavaDoc/docs/basic-domain-model1_en.html