SC Project

Master ALMA 2eme année

Rapport de projet

Encadrants : M.Oussalah & S.Thibaudeau

N.Boukra
L.Duringer
A.Marguerite
M.Ouairy
N.Sebari

Université de Nantes
2 rue de la Houssinière, BP92208, F-44322 Nantes cedex 03, FRANCE

Table des matières

Remerciements

Nous tenons tout d'abord à remercier notre responsable pédagogique de l'Université de Nantes, M.Oussalah et responsable du Master 2 *ALMA* pour son encadrement. Nous tenons également à remercier l'entreprise *Obeo* pour nous avoir donné la possibilté de bénéficier du savoir faire de ses salariés en nous proposant ce stage. Nous tenons à remercier tout particulièrement S.Thibaudeau pour sa disponibité, et son soutien tout au long de ces trois mois.

Introduction

C'est dans le cadre du module SC Project que nous avons été amenés à effectuer le projet *"Mise en place de générateurs pour des applications web"*. Nous avons porté notre choix sur ce sujet pour sa contribution dans l'approche MDA [1] dans un contexte industriel. Ce projet s'articule autour de la problématique suivante :

"Comment pouvons-nous mettre en place des générateurs de code pour des applications web ?"

Pour vous décrire le déroulement du projet et répondre à cette problématique, notre rapport est scindé en cinq grandes parties. Nous allons, dans un premier temps, présenter l'entreprise *Obeo* pour laquelle nous avons réalisé ce projet, pour dans, un second temps, annoncer les objectifs que nous nous sommes fixés au préalable et la démarche mise en oeuvre pour les réaliser. Nous parlons ensuite de la méthodologie qui nous a été proposée par S.Thibaudeau puis des différents travaux que nous avons mis en place. Nous concluons ce rapport, par un bilan sur les travaux réalisés mais aussi les problèmes persistants.

1. Model Driven Architecture.

1 L'entreprise Obeo

1.1 Présentation de l'entreprise

Obeo est une société de service et un éditeur de logiciels [**?**]. Elle est fondée en 2005 dans la région nantaise par l'initiative de S.Lacrampe (directeur général), E.Juliot (directeur commercial) et J.Musset (ancien directeur technique). Elle est à l'initiative du projet Acceleo (*cf.* section [**?**]), un générateur de code basé sur le framework EMF. Son expertise dans le domaine de l'ingénierie des modèles (démarche MDA) lui permet de proposer des solutions allant de la création à la refonte d'applications informatiques.

Figure 1.1 – L'entreprise Obeo

Ces solutions proposées permettent notamment de diminuer les délais de projets et de diminuer les risques d'erreurs. Les amélioration des performances d'adaptation et d'agilité font aussi parties des objectifs des outils et méthode de la société *Obeo*.

La société *Obeo* est aussi un membre actif dans le domaine Open Source et est membre de la fondation Eclipse.

1.2 Démarche M2T

Dans le cadre du développement à base de modèle, l'Object Management Group [**?**] a créer la spécification Meta-Object Facility Model to Text Transformation Language (ou MOF2Text). La société *Obeo* emploie cette démarche qui est au coeur de son projet Acceleo (*cf.* 4.1).

Figure 1.2 – Object Management Group

Le principe du Model To Text repose sur au moins trois notions essentielles :

- Le Méta-Modèle (ou M2) : Le M2 sert à définir les différents concepts pouvant entrer en jeu, ainsi que les relations que ces concepts entretienent entre-eux. Le M2 sert donc de base à la modélisation d'un système.
- Le Modèle (ou M1) : Il repose sur le M2 pour décrire un système (ou une facette d'un système) en organisant les différents éléments entre eux de manière concrête, et en leur assignant un certain nombre de propriétés. Ainsi, le Modèle est utilisé pour fournir une description non-technique du réel et/ou de l'application attendue.
- Le générateur de code : Il est utilisé pour « traduire » un Modèle en code exécutable. Utilisant - la plupart du temps - un système basé sur les templates, il est configuré pour générer certaines portions de code en fonction des éléments rencontrés en parcourant le Modèle. Le générateur de code est donc utilisé pour passer d'un Modèle non-technique à une application exécutable sur la plateforme et/ou l'environnement cible. L'objectif étant qu'un même générateur de code soit capable de produire des applications différentes à partir de Modèles différents (reposant sur le même M2).

Le Modèle permet donc de représenter une application et son contexte en faisant abstraction de la technologie ou de la plateforme cible, à l'inverse du Générateur qui doit savoir produire une solution technique en faisant abstraction du contexte concrêt, qui est fourni par le Modèle.

2 Objectifs et démarches

2.1 Objectifs

Dans le cadre du module *SC Project*, nous avons eu l'opportunité de travailler sur la conception d'outils de modélisation dédiés aux applications web. En partant de trois méta modèles mis à disposition par Obeo (*cf.* section 4) l'objectif était de créer des modèles et les générateurs de code pour produire le code d'une application. L'utilisateur final aura ainsi la possibilité de créer le modèle de son application web (à partir d'une vue en arbre par exemple) puis d'exécuter le générateur mis à disposition pour produire le code de l'application désirée.

2.2 Démarche

Les premières semaines du projet furent consacrées à la prise en main des outils et des technologies. S.Thibaudeau nous a mis à disposition des exemples de générateurs et les références vers les différentes initiations indispensables pour ce projet (Tutoriel Acceleo, ObeoDesigner ...). En parallèle nous avons aussi étudié le framework *Play!* et vérifié la compatibilité avec l'objectif de ce projet.

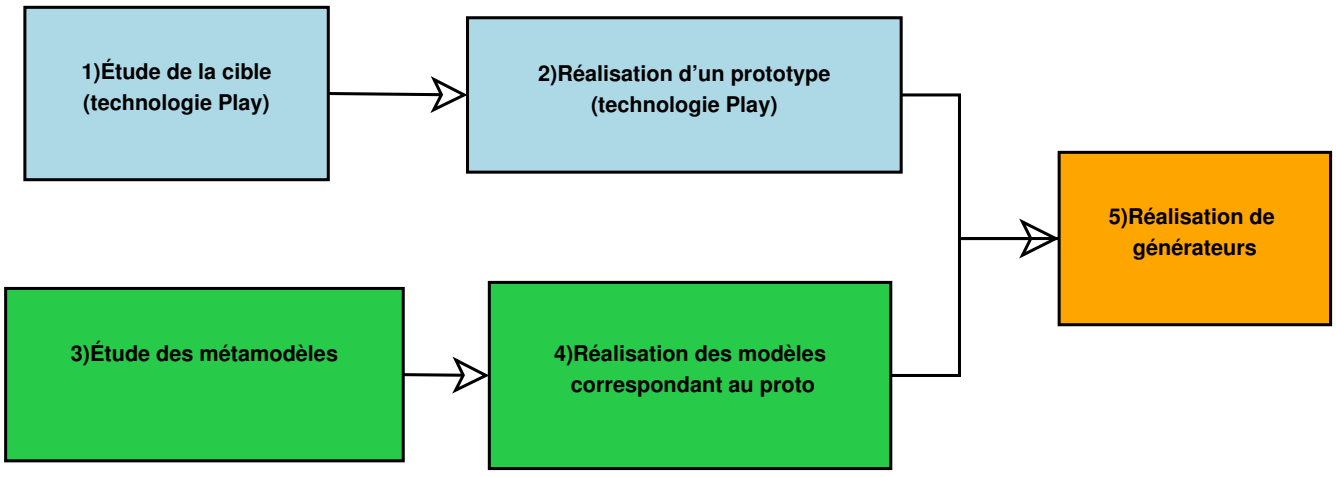

Figure 2.1 – Démarche employée

Dans un second temps, nous avons entamé la conception d'un prototype d'une application web avec *Play!*. Enfin, nous avons étudié les différents méta-modèles mis à disposition puis entamé la création des différents générateurs de code associés. Dans ce rapport nous détaillerons ces

différents étapes du projet. Ainsi le chapitre 3 résume l'étude menée sur le *Play!* et le prototype créé. Dans le chapitre 4 nous reviendrons sur les différents méta-modèles ainsi que les générateurs de code associés.

Outil de gestion de version Comme la plupart des projets à l'initiative de l'entreprise *Obeo*, ce projet est *Open Source*. Afin de partager et collaborer, S.THIBAUDEAU nous a proposé d'utiliser *Github* (*cf.* [**?**]). Nous avons créer un compte https ://github.com/alma2012 dédié à ce projet.

FIGURE 2.2 – L'outil Github

3 Technologie cible et outils

Le premier objectif proposé a été d'étudier le framework *Play!* et de s'assurer de la compatibilité de sa philosophie avec un projet de génération de code. Dans la section 3.1 de ce chapitre, nous détaillerons les qualités de ce framework qui ont motivées la sélection de *Play!* pour ce projet. Dans un second temps (*cf.* section 3.2), nous aborderons le prototype que nous avons élaboré avec *Play!*.

3.1 Le Framework Play

Play! est un framework (kit de composants logiciel) web basé sur les langages JAVA et Scala permettant de la création d'applications web. *Play!* est similaire à d'autres framework tels que Django [**?**] ou Ruby on rails [**?**]. Il utilise le langage JAVA sans utiliser les contraintes java EE, ainsi le développement est simplifié par rapport à d'autres plateformes JAVA. De plus en plus de développeurs choisissent cet outil qui présente de nombreux avantages. L'aspect « prêt à l'emploi » (*Plug'n Play*) permis par ses fonctionnalités par défaut le rendent efficace et rapide à mettre en place et à configurer. Le développement également est facilité, d'une part par par des mécanismes de compilation à la volée (*shadow-build*) lors du chargement des pages, mais aussi par la mise à disposition de système de tests intégrés (JUnit, Selenium). Sa gestion des requêtes Web peut être bloquante ou non-bloquante (synchronisme). La génération des pages Web renvoyées peut être dynamisée grâce à un mécanismes de templating basé sur Scala. Enfin son architecture modulaire, composée de plugins et du *design pattern* MVC (Modèle-Vue-Contrôleur), permettent une répartition claire du code source ce qui est propice à la démarche MDA.

FIGURE 3.1 – Framework Play !

Le Framework *Play!* propose une implémentation d'un site web en trois composantes principales :

3.1.1 Les pages web

Pour la gestion de ses pages web, *Play!* propose une approche basée sur les templates. Les pages peuvent donc être écrites avec du HTML classique, et être enrichies avec du code *Scala* pour générer du contenu dynamiquement. Ainsi, il est possible d'insérer des clauses conditionnelles (if/else) ou des boucles (for/while) à l'intérieur du code HTML. Afin de conserver une architecture

cohérente, il est possible de passer un certain nombre d'arguments/objets en paramètre de ces templates, afin que les pages puissent en afficher le contenu de manière formatée.

3.1.2 Les Données

La gestion des données dans *Play!* est laissée au choix de l'utilisateur - le développeur. Cependant, *Play!* embarque nativement l'ORM *Ebean*. Le principe d'un ORM (Object-Relational Mapping) est de fournir une couche d'abstraction au dessus d'une Base de Données relationnelle. Cette couche d'abstraction doit être suffisante pour que les différents éléments de la base soient manipulables directement en tant qu'Objets. Cela permet aux développeurs de s'affranchir des contraintes techniques que peuvent présenter les Bases de Données relationnelles.

Dans *Play!*, avec *Ebean*, une classe Java pourra être gérée comme étant une entité/table, et ses attributs et références seront traités comme des colonnes de cette table.

3.1.3 Les Contrôleurs et les Routes

Dans *Play!* le/les contrôleur(s) joue(nt) un rôle central au sein l'application. Ce sont les contrôleurs qui se chargent de récupérer les requêtes des visiteurs (de type HTTP), d'effectuer les traitements (ajout d'un cookie, traitement dans la Base de Données, ..., et d'effectuer le rendu des pages web retournées au visiteur.

Dans le monde du Web, les requêtes se présentent la plupart du temps sous forme d'adresse dite URL (Uniform Resource Locator) appelée par le visiteur. Comme beaucoup d'autres framework web, *Play!* propose un fichier de Routes dans sa configuration. Le rôle d'un fichier de Routes est de convertir/résoudre les adresses/URL afin de les faire correspondre à une méthode du Contrôleur de l'application.

Ainsi, lorsqu'une requête arrive à l'application, *Play!* regarde d'abord dans son fichier de Routes, puis appelle la méthode correspondante, qui retournera une réponse/page vers l'émetteur de la requête. Ce mécanisme est illustré sur la figure 3.2.

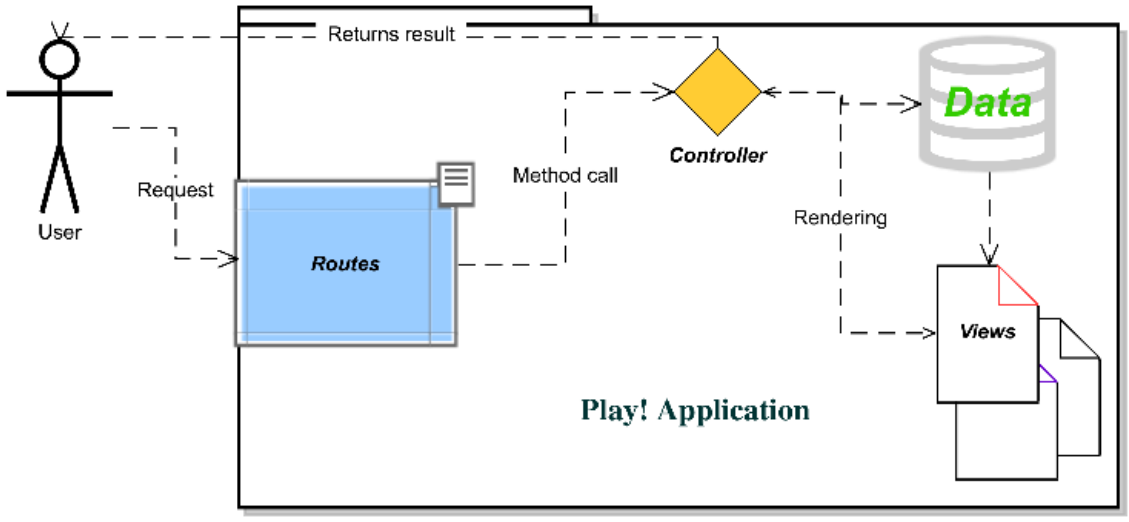

FIGURE 3.2 – Fonctionnement de Play !

3.2 Conception d'une maquette d'exemple

Nous nous sommes proposés de bâtir un mini-site Web basé sur *Play!*.
L'ojectif est double :

1. Nous familiariser avec le framework *Play!* et comprendre son fonctionnement.
2. Obtenir une maquette qui sera utilisée comme objectif de code à générer.

Le prototype que nous avons proposé est une implémentation simpliée d'un site marchand. Ce type de site web est en effet très courant, et il implique différents aspects et fonctionnalités :

- Stockage persistant de données : Produits vendus, profils clients.
- Utilisation de formulaires : Inscription des clients.
- Gestion de sessions : Authentification des clients sur le site, gestion d'un « Panier » temporaire, et achats.
- Mise en place de services administrateurs : Gestion des produits mis en vente.
- Dans la même optique, mise en place de services de type REST, pour une gestion distante du magasin.
- Étoffage graphique du site web : Un site marchand doit être agréable à parcourir.

FIGURE 3.3 – Prototype Play_Shop

4 Contributions

4.1 Présentation de l'outil Acceleo

Acceleo est un générateur de code Open Source développé par *Obeo* et la fondation Eclipse. Acceleo permet de générer du code, à partir de modèles basés sur le framework EMF (*cf.* [**?**]), en mettant en œuvre l'approche Model Driven Architecture (MDA). Le générateur Acceleo est une implémentation de la norme de l'OMG pour les transformations de modèle vers texte (Model to Text : M2T *cf.* section 1.2.

4.1.1 Fonctionnement

Comme la plupart des générateurs de code, Acceleo propose un système basé sur les templates : Les fichiers à générer peuvent être écrits avec des éléments statiques mêlés à des éléments dynamiques.

Les Modules

Dans Acceleo, un Module est une unité de génération amenée a être utilisée par le générateur. Un générateur est donc une agrégation de Modules.
Un Module est paramétré par un ou plusieurs DSL (ndla : définir précédemment, a vérifier) afin de pouvoir proposer les fonctionnalités associées, qui permettent notamment de parcourir chaque élément du Modèle.
Chaque Module est composée d'un ou plusieurs templates. Un template a pour rôle de générer du code d'après un ou plusieurs paramètres (éléments du Modèle). L'utilisation des templates est rendue très intuitive grâce au langage conçu pour Acceleo : Par défaut, le texte écrit dans un Template est recopié tel-quel dans le fichier qui sera généré en sortie. Afin d'insérer du contenu dynamique (base de la génération) dans le texte du Template, Acceleo utilise les balises « [» (ouvrantes) et « /] » (fermantes). Ces balisent permettent d'insérer un certain nombre d'instructions, comme la récupération d'un attribut d'un élément du Modèle, mais aussi des opérations conditionnelles ou des boucles de traitement.
Par convention, chaque Module concerne un ou plusieurs aspects de la génération. Un Module peut également être dédié à la génération d'un type de fichier en particulier.

```
[module utils('http://www.obeonetwork.org/dsl/soa/2.0.0')]

[template public gen_operation_call( anOperation : Operation )]
[anOperation.name/]([for( p : Parameter | anOperation.input)][name/][/for])
[/template]
```

FIGURE 4.1 – Exemple de fichier Module dans Acceleo

Les Queries

Les Modules permettent de décomposer la génération du code en plusieurs sous-parties réutilisables. Ils sont donc utiles pour générer un même type de contenu texte à partir de différents éléments du même type.
Cependant, dans certains cas, il est nécessaire d'exécuter une opération sur un élément du Modèle, comme la modification d'une chaine, ou l'accès à l'élément parent d'un élément, par exemple. Les Queries interviennent alors comme un moyen de déporter ces opérations non-triviales dans une base commune. Les spécificité des Queries par rapport aux templates est que ces dernières mettent en cache leur résultat. Ainsi, si une Query est appelée deux fois avec les mêmes paramètres, celle-ci ne sera pas re-calculée et se contentera de retourner le résultat obtenu précédement. Cette fonctionnalité est donc très utile pour l'appel répétitif d'opérations (même basiques), ou le stockage de variables globales.
Les Queries permettent plus généralement de structure le générateur. Par convention, les générations de texte sont traitées dans des templates, tandis que les opérations plus complexes (modification de chaine, parcours d'éléments) ou répétitive sont déportées dans des Queries.

```
[query public get_implementation_classname( aService : Service ) : String =
aService.name + 'Impl'
/]

[query public get_interface_classname( aService : Service ) : String =
aService.name + 'Iface'
/]
```

FIGURE 4.2 – Exemple de Queries dans Acceleo

Les Services

Les Services peuvent être vus comme une extension des Queries. Là où les opérations exécutables des Query se limitent à celles d'Ocl (TODO : Aborder Ocl). Le principe des Services consiste à appeler du code Java depuis une Query via des invocations de méthodes. Cela permet d'effectuer des opérations complexes ou bien de stocker un certain nombre d'informations via du code Java. Ces informations restent récupérables depuis les templates pendant toute la durée de la génération de code.

Les balises « Code Utilisateur »

Si un Générateur de Code permet aisément de transcrire la structure et la logique de fonctionnement d'un système, il est en revanche impossible d'en saisir automatiquement tous les aspects ou particularités. En d'autres termes, un code généré sera rarement complet. *Acceleo* fournit donc un système très simple permettant de définir des zones de « Code Utilisateur » au sein d'un Template. Ainsi, l'utilisateur peut apporter des modifications ou ajouts dans les fichiers générés. Si ces modifications ont été effectuées à l'intérieur d'une zone « Code Utilisateur », ces dernières ne seront pas affectées si les fichiers sont générés à nouveau. De telles zones peuvent être insérée au sein d'un Template en utilisant de simples balises « ***[protected]*** »

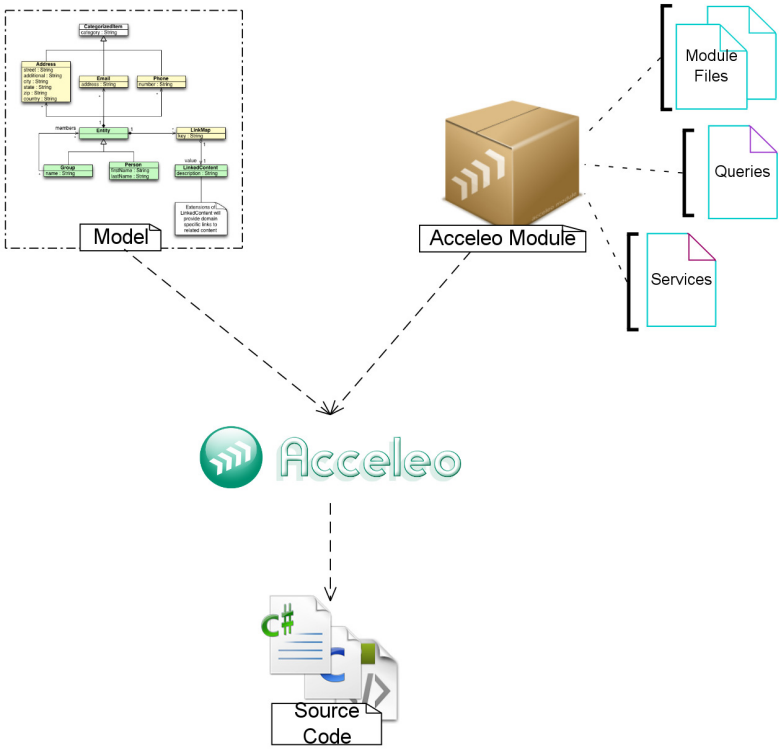

FIGURE 4.3 – Fonctionnement d'Acceleo

4.2 Le Méta-Modèle Entity

Les Entity permettent de simplifier la gestion des données au niveau d'une application, mais aussi de faciliter la sauvegarde en base de données. Plus concrètement, ces Entity nous permettent de prendre en charge la persistance des données de notre application dans une ou plusieurs sources de données, tout en gardant les relations entre celles-ci. Ces composants établissent donc la relation entre notre application et notre bases de données. Un méta-modèle Entity a été établie par Obeo afin de représenter la structure des Entity qui vont définir la couche métier de notre application. La figure 4.4 montre le méta-modèle Entity simplifié.

Tout modèle conforme au métamodèle "Entity" peut être constitué de plusieurs de plusieurs Bloques (blocks). Chaque bloque se compose de plusieurs Entity. Celle-ci possède un ou plusieurs attributs, et peut référencer d'autres Entity. Un attribut peut avoir une métadonnée qui peut être constituée d'une ou plusieurs annotations.

4.2.1 Gestion des entités avec *Play!*

Dans la plate-forme Play, tout le code métier est porté par les objets du modèle. Celui-ci contient les données persistantes, ce qui s'accordent parfaitement avec le concept d'*Entity*. On peut par exemple écrire le code suivant pour manipuler une entité "Personne" :

```
// Si on voulait récupérer toutes les personnes
List<Personne> personnes = Personne.find("byName","Takfarinas").fetch();
Personne p1 = Personne.findById(1); // Récupérer la personne ayant l'id 1
```

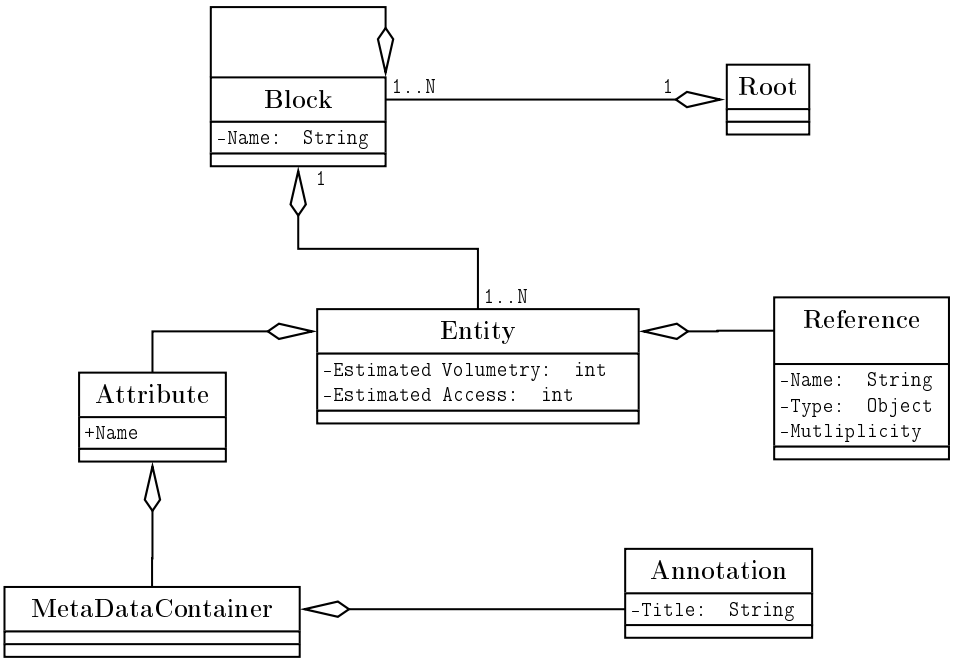

FIGURE 4.4 – Méta-modèle Entity

p1.firstName = "paul" ; // Modification de l'entité
p1.save() ; // Mise à jour dans la base de données

4.2.2 Conception du modèle

Nous avons employé la simple démarche suivante : pour chaque modèle *Play!* on associe une entité. À ce stade, on en déduit facilement les attributs de chaque instance d'`Entity` et les instances `Reference` correspondant aux associations avec d'autres entités. *Play!* utilise des mécanismes d'annotation au sein de ses modèles. Ces annotations permettent notamment de donner des contraintes à des attributs. Nous avons facilement pris en compte ce concept en utilisant la classe `Annotation` mise à disposition dans les méta datas d'un `Attribut`.
Nous avons élaboré un modèle d'entity de notre application conformément au méta-modèle que nous avons décris en dessus. Ce modèle est illustré par la figure 4.5 ci-dessous.

FIGURE 4.5 – modèle Entity de l'application

Nous avons représenté un seul bloque dans notre modèle d'Entity qui va regrouper les Entity suivantes :

- **Costumer** : Représente le client qui va faire l'achat sur la plate-forme
- **ProductSale** : représente les produits mises en vente.
- **ProductPurchased** : représente les produits achetés par un client.
- **ProductInfo** : contient les informations d'un produit.
- **cart** : représente une carte de paiement reliant un client aux produits qu'il a acheté.

Génération de code Entity

4.3 Le Méta-Modèle *SOA*

SOA (Service Oriented Architecture) - ou Architecture Orientée Services - modélise le concept de Services et d'Opérations au travers d'un système basé sur les Composants. Concrêtement, un modèle basé sur *SOA* est constitué de Composants. Chaque Composant peut possèder des Services dont des Interfaces permettent de communiquer avec l'extérieur. Ces Interfaces définissent une ou plusieurs Opérations, chaque opération étant caractérisée par un ensemble de Paramètres d'entrée et de sortie. Ces paramètres peuvent être des types relativement basiques (Integer, String, ...)mais également des « Entity » complexes, comme des informations complètes sur les utilisateurs, ou sur des produits.

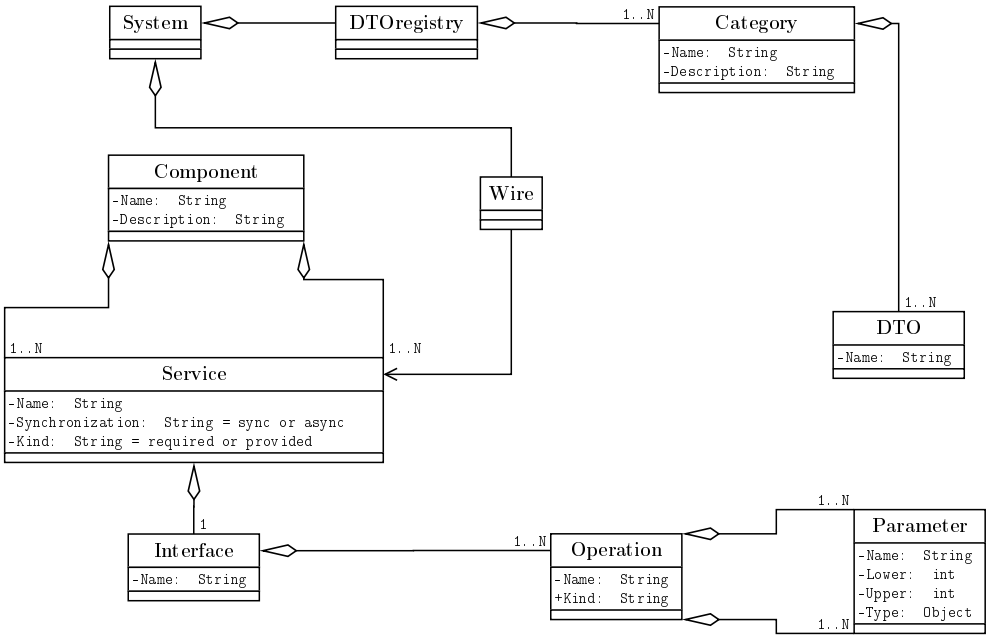

FIGURE 4.6 – Métamodèle SOA

4.3.1 Le concept de *SOA* dans *Play!*

Play!, de part son achitecture en MVC, ne propose pas d'implémentation concrête de la notion de Services. Nous avons donc fait le choix de déporter ces derniers dans des classes externes de type Singleton, ce qui permet de faire appel aux différents Services depuis n'importe quel Controleur de *Play!*. (*Services.getAKindOfService().do_service()*). Cela permet de séparer les différents concepts sans pour autant remettre en cause l'architecture originelle de *Play!*.

4.3.2 Conception du modèle

La conception du modèle SOA pour le contexte de notre magasin en ligne a été assez simple à convenir : Pour chaque cas d'utilisation ou action, nous avons modélisé un certain nombre d'opérations basiques. Ces opérations ont été réunies dans différents Composants et Services, en fonction de leur contexte, mais également en fonction du type d'utilisateur susceptible de faire appel à ces opérations.

Les types de Services que nous avons représenté dans notre Modèle SOA sont regroupés dans les Composants suivants :

- **UserManager :** Contient un Service de gestion des utilisateurs/clients du site, accessible depuis l'intérieur de l'application uniquement. Ce Service sera utilisé pour contrôler l'inscription et l'authentification des utilisateurs du site.
- **ProductsManager :** Contient des Services de gestion des produits vendus sur le site. Ces Services pourront - pour la plupart - être appelables depuis l'extérieur. On distinguera deux Services différents pour distinguer le traitement des informations relatives aux produits (label, description), et les ventes en elles-mêmes (prix, stock). Cela permet également d'assigner des propriétés/paramètres différents pour ces deux Services, ce qui faciliterait une gestion des droits d'accès aux Services.
- **ShopManager :** Tout comme l'**UserManager**, ce Composant contient des Services qui seront - à priori - destinés à une utilisation interne à l'application. Il sera utilisé lors de l'ajout des produits au « Panier » d'un client (appelé « Cart », en Anglais), et pour valider les achats des clients. C'est également ce Composant qui pourra faire le lien avec les Services de paiements externes (non-implémentés dans notre prototype).

4.3.3 Génération du code des services

Séparation interfaces/implémentations

Afin d'assurer une souplesse de l'application, la génération des différents Services systématiquement découpée en deux parties :

- Des interfaces, contenant les signatures des différentes opérations.
- Des classes d'implémentation, liées à ces interfaces.

Les classes d'implémentation sont générées avec le code des différentes opérations, lorsque celles-ci sont basiques et/ou facilement interprêtables (récupérer/éditer/détruire une entité). Dans tous les cas, des balises « user code » (à définir) sont également insérées afin de laisser libre choix à l'utilisateur pour les détails de l'implémentation.
Cette séparation interfaces/implémentations permet également d'avoir plusieurs types d'implémentation de Services pour une même interface.
Nous avons implémenté les différentes classes de Services comme étant des Singleton. Il est ainsi possible d'appeler un Service depuis n'importe quel endroit de l'application avec une syntaxe du type *Services.getMonPremierService().faireMonTravail()*.

Plus loin avec SOA : Les webservices

Ils nous a été proposé d'ajouter des couches supplémentaires par dessus les simples implémentations de Services SOA, notamment au niveau des moyens d'accéder à ces Services.
Nous avons donc mis en place des solutions permettant d'appeler certains Services depuis l'extérieur, sans passer par l'interface du site Web. Pour ce faire, nous avons procédé à la mise en place d'un service de type REST (REpresentational State Transfer).

Le principe est de recevoir des requêtes HTTP de type GET/POST/PUT/DELETE provenant d'autres programme que des simples navigateurs.
Par exemple, il devrait être possible d'appeler certains services depuis un logiciel de gestion, pour la gestion des ventes des produits, des factures, ou des comptes utilisateurs, ou pour collecter des informations.

Nous avons donc implémenté un tel système en configurant, pour chaque opération accessible depuis l'extérieur, des Routes spécifiques déclenchant l'exécution des opérations - c'est à dire des appels aux services générés précédement -, et retournant un résultat au format *JSON*.

4.4 Le Méta-Modèle Cinématique

Le métamodèle cinématique est organisé autour de trois principaux packages :

- Toolkit : représente les concepts liés à la définition des widgets[1] IHM.
  Le package Toolkit est construit de la manière suivante :

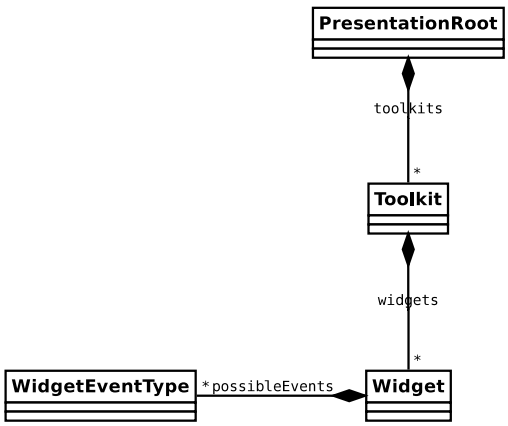

FIGURE 4.7 – Métamodèle toolkit

- View : représente les concepts liés à la définition des écrans IHM.
  Le package View est construit de la manière suivante :

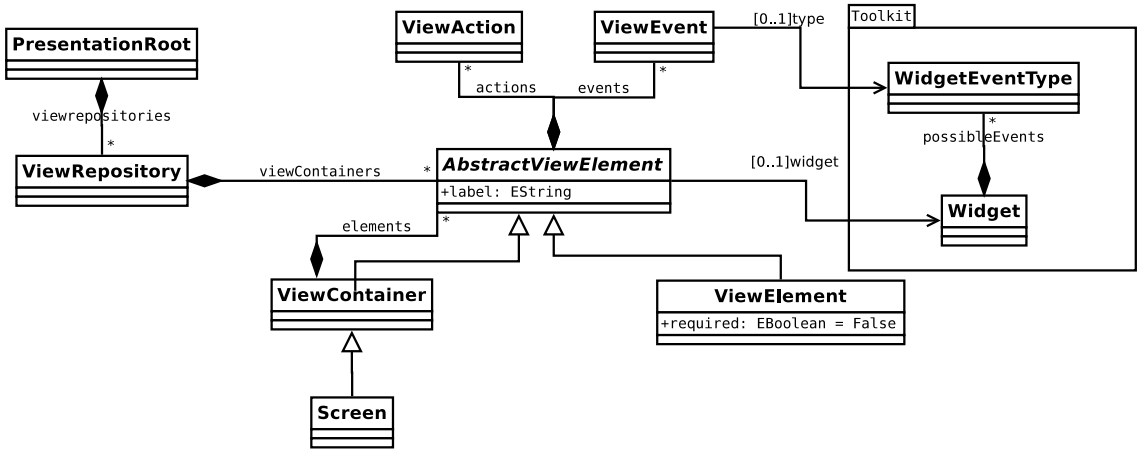

FIGURE 4.8 – Métamodèle view

- Flow : permet d'identifier le comportement dynamique des écrans IHM. Le flow peut être appréhendé comme une sorte de diagramme d'activités.
  Le package Flow est construit de la manière suivante :

1. Elément visuel d'une interface graphique (bouton, ascenseur,liste déroulante, etc.)

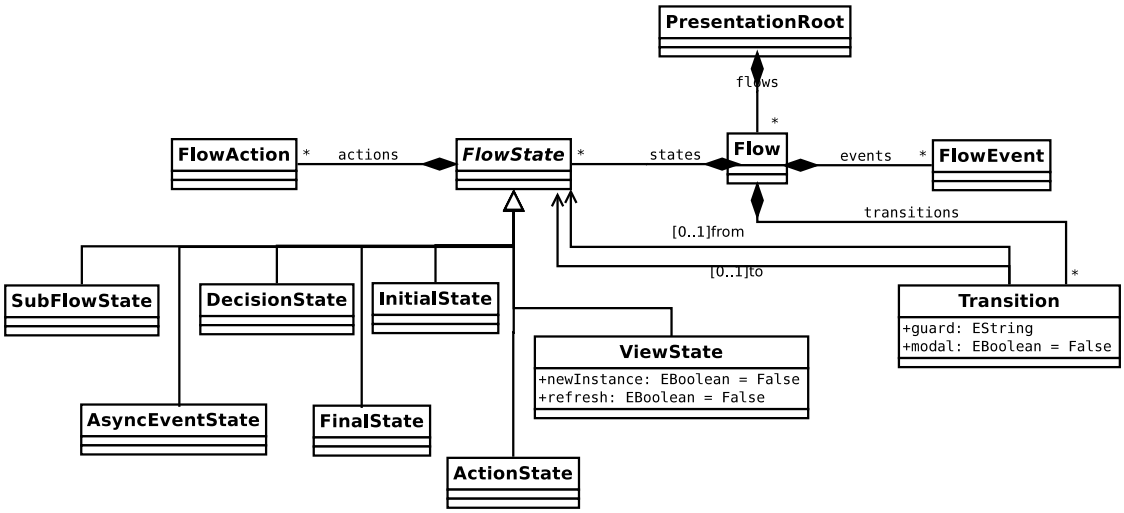

FIGURE 4.9 – Métamodèle flow

4.4.1 Le concept cinématique dans *Play!*

Nous retrouvons dans *Play!* un repertoire dans lequel sont regroupés les différents fichiers Web (HTML, XML, ...) liés aux vues (principe MVC). Les fichiers relatifs à la création des IHM sont donc indépendants du reste de l'application.

4.4.2 Conception du modèle

La modélisation IHM de l'application prototype `Play` a été mise en place à l'aide de l'outil acceleo de `ObeoDesigner`. Le modèle est construit autour des trois principes du métamodèle défini précédemment :

1. Toolkit : Représente une palette contenant des widgets. Nous avons utilisé le modèle toolkit (par défaut) car il couvre tous les éléments de type "widget web" qui puissent exister (bouton, liste déroulante, champs de saisie, tableau, etc.). Chaque élément "widget" peut lever des événements de type *WidgetEventType*.
2. View : Permet de représenter tous les éléments graphiques d'une interface utilisateur. Au plus haut niveau, nous définissons les éléments de type *View Container*.
   Un *View Container* peut être de type *Page*, *Panel* ou *Table*.
   Nous choisissons le type *Page* pour représenter un écran IHM. Ensuite, pour chaque View-Container, nous définissons les éléments qu'il contient, ça peut être un formulaire ou un tableau tout simplement. Nous construisons donc nos écrans, de cette manière, au fur et à mesure jusqu'à obtenir toutes les fenêtres IHM qui composent l'application.

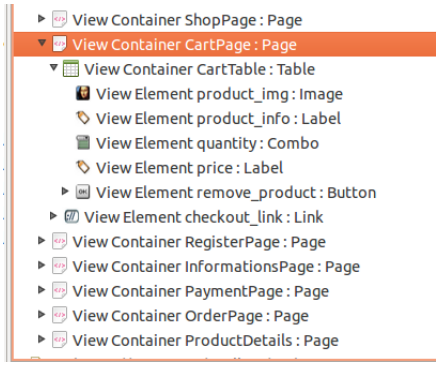

FIGURE 4.10 – Construction du modèle (partie Views)

3. Flow : Représente la manière dont les écrans de l'application peuvent s'enchaîner. Il décrit le comportement dynamique de l'IHM sous la forme d'enchaînements entre des états. Nous définissons ici, un état initial *Initial State Start*, des états *View State*, des états d'action *Action State* et des transitions :
   - Les états *View State* correspondent aux écrans IHM et contiennent les *View Container* décrit précédemment.
   - Les états d'action *Action State* définissent les actions à réaliser sur chaque écran de l'application. Ils sont en liés aux fonctions offertes par la partie *SOA*.
     Exemple : Pour le chargement de la liste des produits disponibles pour la page d'acceuil (vitrine), nous définissons un *Action State* nommé "Load Public Product List" que nous lions à l'opération getProductList() offerte par *SOA*.
   - Les transitions "*Transition*" font passer l'IHM d'un état à un autre (d'un écran à un autre).
     Exemple : Une transition pour l'affichage de l'écran d'acceuil "Display Main Page" va de l'état "Action State Load Public Product List" à l'état "View State Display MainPage". Les transitions peuvent également avoir des conditions de garde.

4.4.3 Génération du code

4.4.4 Résultats obtenus

4.5 Déploiement

Pendant le développement des différents Générateurs de Code, nous avons pu utiliser les outils proposés par *Eclipse* afin de configurer statiquement le répertoire cible des fichiers à générer lors de l'exécution desdits générateurs. Cette méthodologie nous a permis de définir un Projet *Play!* dédié comme cible de la génération, afin d'effectuer des tests rapidement, sans qu'il ne soit requis de reconfigurer la génération après chaque modification. Le système en place nous a donc permis de tester chaque générateur (pour Entity, SOA, et Cinematic) indépendamment.

Il n'était cependant pas concevable d'utiliser la même méthodologie pour une « mise en production » du générateur *Play!*. En effet, il n'est pas concevable d'obliger un utilisateur à configurer et lancer manuellement chaque « sous-générateur », car ceux-ci sont complémentaires (à l'exception peut-être de la génération des WebServices). De plus, il faut dispenser l'utilisateur d'une configuration fastidieuse, et trouver un système plus ergonomique pour que ce dernier puisse lancer rapidement une génération de code.

4.5.1 Regroupement des différents générateurs de code

La première étape du déploiement a été de réunir les différents « sous-générateurs » en un seul programme principal. Pour cela, nous avons choisi de créer un Méta-Modèle abstrait défini pour pouvoir contenir tout type de Modèle. Ainsi, nous avons pu encapsuler chaque Modèle utilisé - en l'occurence, Entity, SOA et Cinematic - en un seul Modèle « Application ».
Nous avons ensuite modifié la manière dont les générations sont lancées afin que le Modèle unique soit automatiquement parcouru afin d'y retrouver chaque Modèle et ainsi lancer les générateurs correspondant. La figure 4.11 illustre la structure de notre Application.

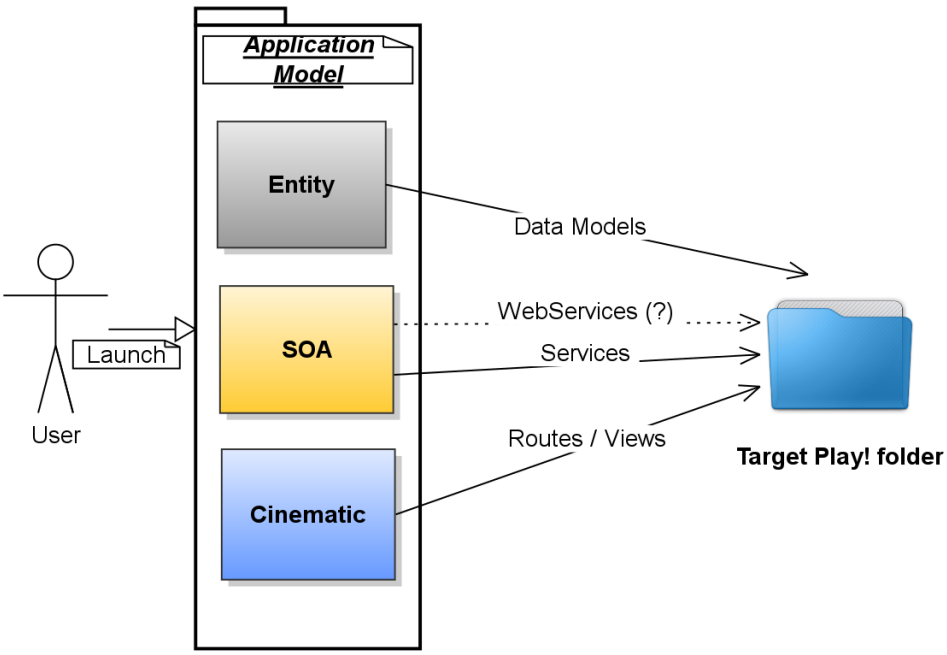

FIGURE 4.11 – Structure de notre Modèle « Application »

4.5.2 Interface Utilisateur

Acceleo embarque une grande quantité d'outils pour faciliter le déploiement des générateurs de code. Nous avons donc pu générer automatiquement une interface utilisateur. Cette interface se présente sous forme d'un Plugin *Eclipse* qui permet d'ajouter automatiquement un menu conextuel « Générer application Play ! » lorsqu'un clic-droit est effectué sur un fichier Modèle. Afin de rendre la génération plus adaptée aux besoins des utilisateurs, nous avons légèrement modifié le code de ce Plugin afin que l'utilisateur puisse sélectionner rapidement le dossier de destination du code à générer. Nous en avons égamelement profité pour ajouter une *Checkbox* (case à cocher) demandant à l'utilisateur s'il souhaite - ou non - générer le code relatif aux WebServices du site *Play!* (Figure 4.12).

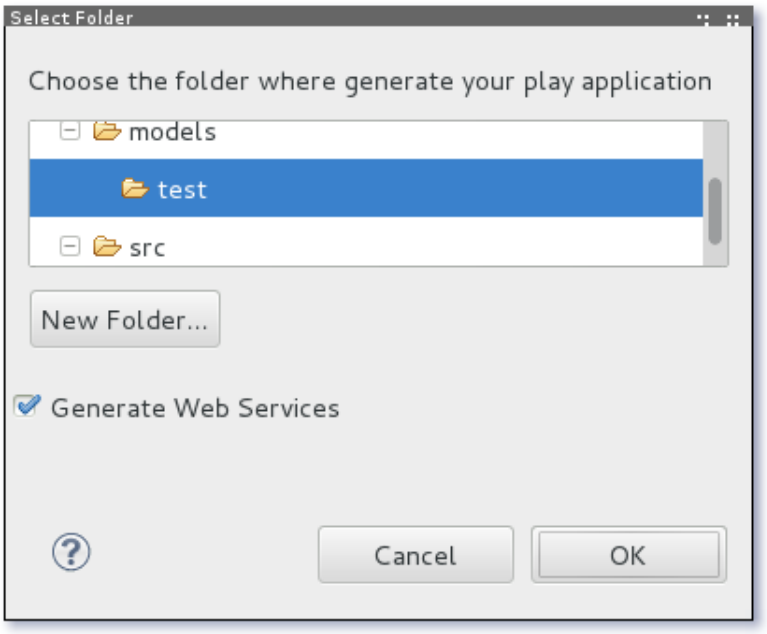

FIGURE 4.12 – Interface Utilisateur pour le lancement du générateur

5 Conclusion

5.1 Travaux effectués (terme pour « travaux terminés » ?)

Au terme de ce projet, nous proposons un générateur de code déployé sous forme de Plugin Eclipse. À partir des trois modèles (`Shop.entity`, `Shop.soa`, `Shop.cinematique`) l'utilisateur peut, par le biais d'une simple IHM (menu contextuel), lancer la génération d'une application *Play!* prête à l'emploi. Il sera toutefois nécessaire d'ajuster manuellement certaines portions de code, afin d'effectuer le « branchement » des Services entre-eux, ou d'ajouter des niveaux de détail aux différentes opérations.

5.2 Problèmes persistants

Étant donné le court délai induit par notre cursus (trois mois, à raison d'une journée par semaine pour ce travail) et le temps d'apprentissage pour *Acceleo* et le Framework *Play!*, tous les aspects n'ont pas pu être abordés, et les générateurs devront être complétés pour faire face à de nouveaux besoins. Nous avons néanmoins structuré les générateurs de façon à garantir leur maintenabilité, en structurant correctement les Templates et Queries et en établissant des conventions de nommage.
Il est également à noter que nous avons fait l'impasse sur les tests unitaires, qui peuvent pourtant être considérés comme primordiaux lorsque l'utilisateur/développeur final souhaite modifier le code tout en s'assurant que l'application fonctionne toujours.

5.3 Bilan

Ce Projet de fin d'études nous a permis de découvrir en profondeur les méthodologies du MDA au travers de la solution puissante que représente *Acceleo*. Nous avons également pu expérimenter le Framework *Play!* qui consitue une solution encore jeune mais efficace pour la conception de sites et services Web en Java. La solution que nous avons développée gagnerait à être enrichie et améliorée, mais elle nous a permis de passer par toutes les étapes d'un développement MDA, de la conception des Modèles au système de déploiement de l'application, en passant par la création des générateurs et l'élaboration d'un prototype.

Acceleo - et plus généralement les logiciels MDE - constitue une technologie d'avenir, qui permet à l'utilisateur (ou développeur) final de s'approprier son application et de l'adapter en fonction de ses nouveaux besoins. En effet, une simple modification du Modèle de l'application lui permet de modifier le comportement de son application, et ce sans nécessiter une expertise ou un nouveau cycle de développement, souvent coûteux en temps et en argent.

Table des figures