C

Industrialisation des développements Spring dans Eclipse

L'objectif de cette annexe est de décrire comment mettre en œuvre une approche dirigée par les modèles afin d'industrialiser les développements fondés sur le framework Spring. Cette approche se fonde sur les standards en terme de modélisation afin d'améliorer la productivité des développements tout en garantissant une bonne maintenabilité et évolutivité des applications.

En effet, Spring offre d'intéressants mécanismes permettant d'améliorer la modularisation des traitements et l'architecture des applications. Il se positionne ainsi comme une excellente boite à outils pour mettre en œuvre des architectures d'entreprise. Néanmoins, les aspects structurels relatifs peuvent être facilement générés puisqu'ils suivent la même structure dans les applications.

Une approche de génération de code fondée sur les informations contenues dans des modèles UML offre ainsi un intéressant levier afin d'optimiser les développements. Après avoir identifié les différents types d'éléments, elle permet de déduire leurs traitements communs ainsi que les relations entre eux et leur configuration dans Spring.

Dans cette annexe, nous allons décrire les différentes étapes permettant de modéliser et d'industrialiser une architecture d'entreprise. À cet effet, le standard de modélisation UML ainsi que les outils Eclipse, Papyrus et Acceleo sont utilisés.

Concepts

Avant de montrer concrètement comment utiliser une approche dirigée par les modèles dans des développements utilisant Spring, rappelons brièvement les différents concepts mis en œuvre à savoir le standard UML et ses mécanismes d'extensions, l'approche MDA ainsi que la génération de code.

Langage UML

UML (Unified Modeling Language) décrit un formalisme correspondant à un langage de modélisation normalisé par l'OMG (Object Management Group) depuis novembre 1997. Il permet de définir et manipuler des modèles, un modèle étant un plan par analogie aux pratiques utilisées dans le secteur du bâtiment.

Le standard UML définit une palette de concepts permettant de modéliser les différentes entités et mécanismes des systèmes informatiques.

La version 2 d'UML introduit treize diagrammes afin de décrire tous les aspects d'un système en se fondant sur une palette de concepts, un diagramme correspondant à une vue graphique d'une partie d'un modèle. Les diagrammes proposés par le standard sont divisés en trois grandes familles. Les tableaux C-1 à C-3 listent ces différents diagrammes.

Tableau C-1 – Différents diagrammes structurels d'UML2

Type de diagramme	Description
Diagramme de classes	Permet de modéliser les classes de systèmes.
Diagramme d'objets	Permet de modéliser les objets, des instances de classes, de systèmes.
Diagramme de composants	Permet de modéliser les composants de systèmes.
Diagramme de déploiements	Permet de modéliser les infrastructures matérielles de systèmes.
Diagramme de paquetages	Permet d'organiser les éléments du modèle en différents paquetages.
Diagramme de structure composite	Permet de décrire la structure interne d'une classe.

Tableau C-2 – Différents diagrammes comportementaux d'UML2

Type de diagramme	Description
Diagramme de cas d'utilisation	Permet d'identifier les interactions entre les acteurs de systèmes et le système lui-même.
Diagramme d'état-transition	Permet de modéliser des machines à états pour décrire le comportement du système ou de tout élément du système
Diagramme d'activité	Permet de modéliser un flux d'activités pour décrire le comportement du système ou de tout élément du système

Tableau C-3 – Différents diagrammes d'interactions ou dynamiques d'UML2

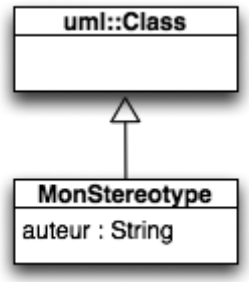
Type de diagramme	Description
Diagramme de séquence	Permet de décrire une séquence d'interactions entre différents éléments d'un système.
Diagramme de communication	Permet de décrire une séquence d'échange de messages entre différents éléments d'un système.
Diagramme global d'interaction	Permet de modéliser les enchaînements de plusieurs séquences d'inter actions afin de réaliser le lien entre différents diagrammes de séquences.
Diagramme de temps	Permet de décrire le comportement d'une donnée au cours du temps.

Comme nous l'avons vu précédemment, UML offre une large palette de concepts abstraits et génériques. Il est ainsi très souvent utile et nécessaire d'étendre le langage afin d'ajouter des informations sémantiques propres à un domaine métier. À cet effet, la notion de profil est introduite.

Un profil correspond à un paquetage permettant de regrouper des stéréotypes, un stéréotype étant un mécanisme d'extension des concepts UML permettant de les spécialiser pour un domaine. Pour ce faire, il suffit de l'appliquer sur une entité UML. L'usage d'un tel mécanisme permet ainsi de définir différents types d'éléments et même de leur ajouter des propriétés.

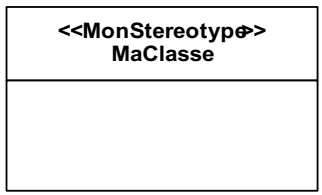
Il est par exemple possible de définir un stéréotype dénommé `MonStereotype` et possédant la propriété `uneProprietе` de type chaîne de caractères, comme l'illustre la figure C-1.

Figure C-1
Mise en œuvre d'un stéréotype



Une fois définis, les stéréotypes sont applicables sur tous les concepts d'UML. Ainsi, dans notre exemple, nous pouvons utiliser le stéréotype `MonStereotype` afin de spécialiser la classe `MaClasse`. Il apparaît visuellement sur la représentation graphique sous la forme `<<MonStereotype>>` et nous pouvons accéder à la propriété `auteur` afin de la valoriser. La figure C-2 illustre cet aspect.

Figure C-2
Application d'un stéréotype sur une classe



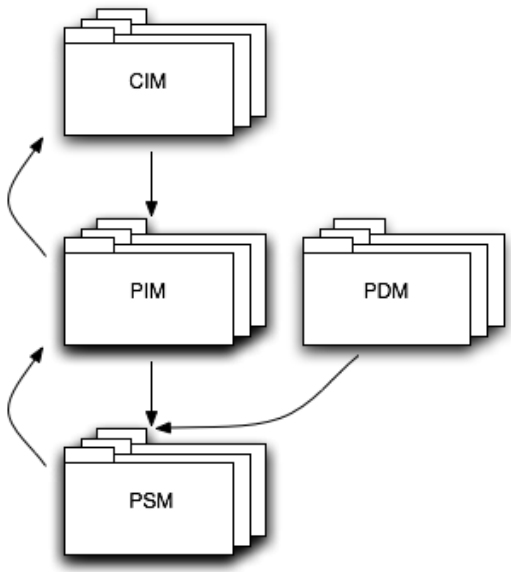
Ce mécanisme est utilisé afin de spécialiser les éléments modélisés afin de réaliser une génération différente suivant les stéréotypes appliqués.

Il est à noter qu'il existe également d'autres mécanismes d'extensions permettant notamment de définir des contraintes à l'aide du langage OCL (Object Constraint Language).

Approche MDA

L'approche MDA (Model Driven Architecture) est normalisée par l'OMG, un organisme définissant un ensemble de spécifications et de techniques afin de manipuler et transformer des modèles. Cette démarche propose un cadre méthodologique définissant différents niveaux de modélisation ainsi que les étapes permettant de passer d'un niveau à l'autre. La figure C-3 illustre ces différents niveaux ainsi que leurs relations.

Figure C-3
Différents niveaux mis en œuvre dans MDA



Pour ces niveaux, différents types de modèles sont définis et adressent un niveau d'abstraction particulier. Le tableau C-4 récapitule les différentes caractéristiques de ces derniers.

Tableau C-4 – Caractéristiques des principaux types de modèles de MDA

Type de modèle	Description
CIM (Computational Independent Model)	Correspond au modèle de description des exigences et besoins métiers.
PDM (Plateform Description Model)	Représente le modèle de description de la cible technologique.
PIM (Plateform Independent Model)	Représente le modèle d'analyse et de conception abstraite.
PSM (Plateform Specific Model)	Correspond au modèle de code et représente le code source de l'application pour une technologie.

Retenez néanmoins que l'approche MDA est analogue à celle utilisée dans le bâtiment ou l'industrie, approche se traduisant par la réalisation de plans ou de spécifications détaillées avant de produire. L'idée de l'approche est d'appliquer cette pratique au processus de construction du logiciel, les «plans» devenant dans ce contexte des «modèles».

Génération de code

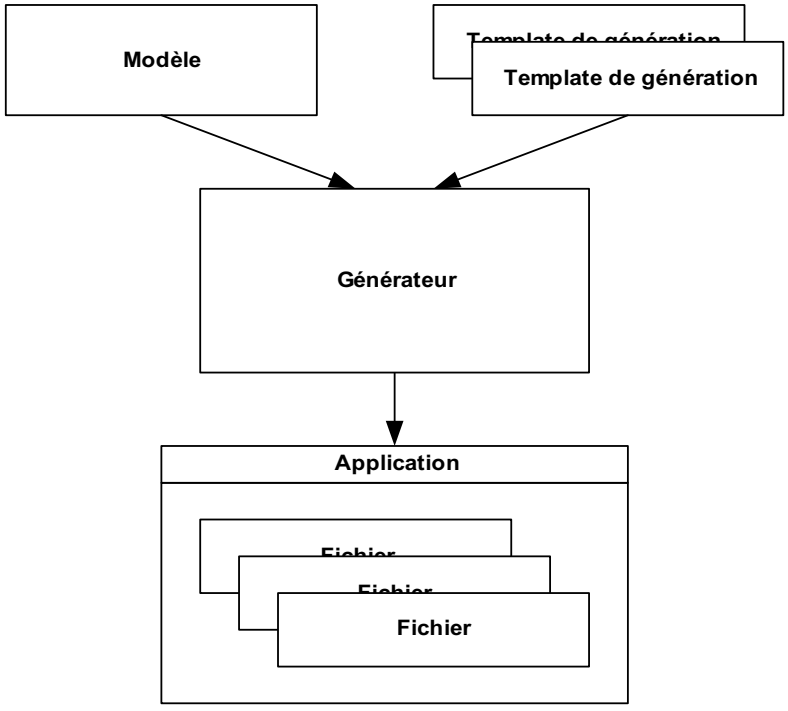
La génération de code consiste en l'utilisation des informations présentes dans un modèle afin de créer des fichiers de type texte contenant le code source correspondant et éventuellement d'autres informations utiles à la cible technologique visée.

Elle correspond donc à un cas particulier de l'approche MDA avec le passage d'un modèle d'analyse (PIM) vers un modèle de code (PSM).

À cet effet, les solutions de génération de code proposent de décrire les instructions de parcours du modèle ainsi que les blocs de code à générer à l'aide de fichiers template. Encore appelé script de génération, ces entités proposent des facilités afin d'accéder aux éléments du modèle. Les parties variables sont remplacées par les valeurs du modèle en entrée par le générateur lors l'exécution de la génération.

La figure C-4 suivante illustre la mise en œuvre de ces différents éléments afin de réaliser une génération de code.

Figure C-4
Chaîne de fonctionnement de la génération de code



Outillage

Afin de mettre en œuvre l'approche de génération de code, sont nécessaires un outil de modélisation, UML dans notre cas, afin de décrire dans des diagrammes de classes les différentes entités constitutives de notre application. Ce type d'outils permet d'éditer graphiquement des modèles.

À partir de ces modèles, un outil de génération doit être utilisé afin d'exploiter leurs informations et de créer les ressources correspondantes.

Papyrus

La fondation Eclipse héberge un projet de haut niveau dénommé Eclipse Modeling et regroupant les différentes briques fondamentales à la mise en œuvre d'une approche dirigée par les modèles dans cet environnement de développement. La figure C-5 décrit ces différentes briques tout en soulignant leurs dépendances.

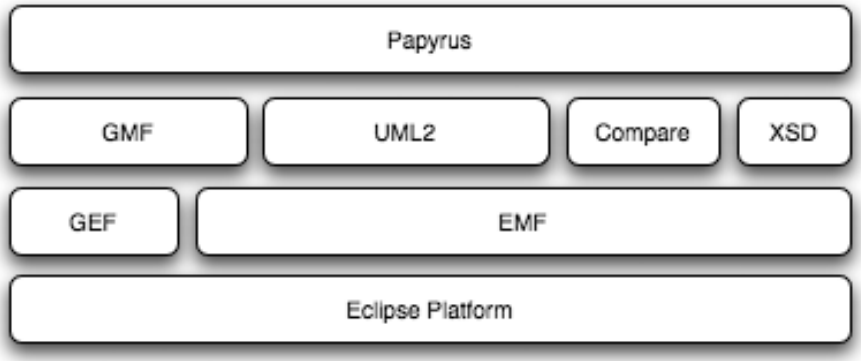


Figure C-5

Briques relatives à la modélisation du projet Eclipse Modeling

La brique de base EMF (Eclipse Modeling Framework) offre les mécanismes de définition et de manipulation de métamodèles et modèles et correspond à la pierre angulaire du projet Eclipse Modeling. Elle permet notamment de créer et naviguer dans un modèle tout en offrant le support des transactions, des mécanismes de requêtage et de validation.

Le support est mis à disposition par l'intermédiaire du greffon UML2 implémentant le métamodèle UML2 en se fondant sur EMF. Il permet ainsi de créer et manipuler par la programmation des modèles au format UML2 tout en offrant un éditeur graphique arborescent. Ce dernier permet aussi bien de parcourir les éléments du modèle que d'éditer ce dernier.

Pour finir, le greffon Papyrus fournit un modeleur UML haut niveau en s'appuyant sur Eclipse UML2 Ce modeleur offre la possibilité d'utiliser les différents diagrammes du standard afin de manipuler des modèles UML2. Modeleur UML officiel de l'environnement Eclipse, il vise à devenir l'implémentation de référence du standard UML.

L'installation des différents greffons d'Eclipse Modeling se réalise par l'intermédiaire de la fonctionnalité de mise à jour intégré à Eclipse. Cette dernière est disponible à l'aide du menu Aide/Software Updates, dans l'onglet « Available Software ». Dans cet onglet la section « Ganymede Update Site » propose une catégorie dénommée « Models and Model Development » contenant l'ensemble des greffons du projet Eclipse Modeling.

Afin d'installer les outils nécessaires, il convient de sélectionner les rubriques « UML2 End-User Features » et « Eclipse Papyrus » puis de lancer l'installation en cliquant sur le bouton « Install… », comme l'illustre la figure C-6.

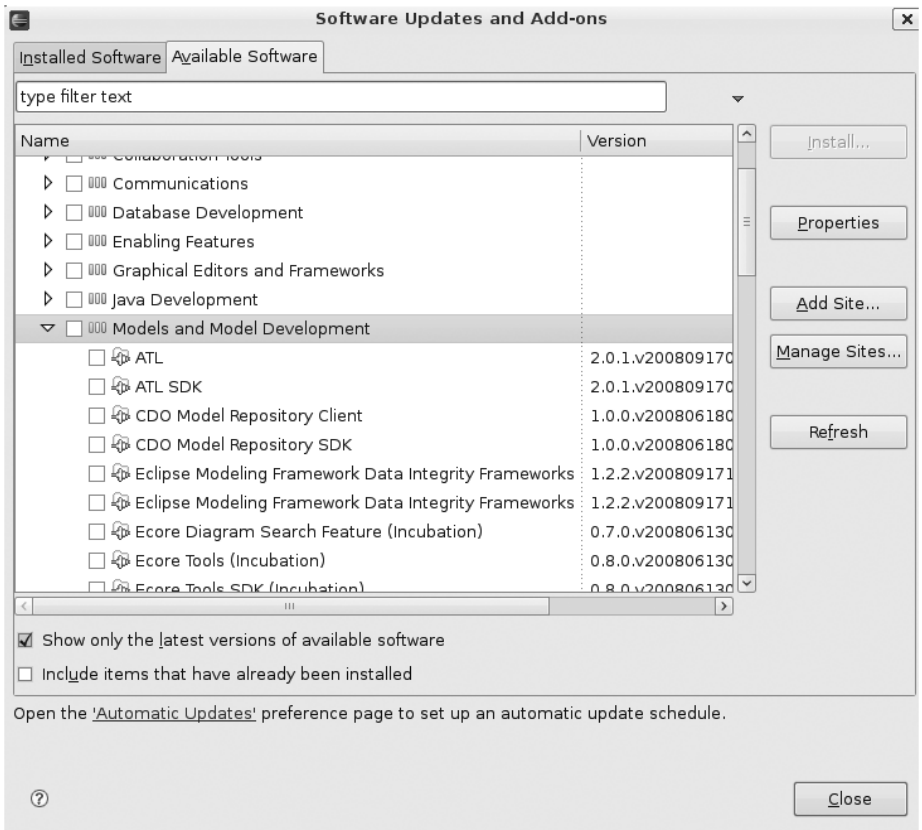


Figure C-6

Ecran de sélection et d'installation des greffons UML2 et Papyrus

Acceleo

Acceleo correspond à un projet Open-Source initié par la société Obeo et hébergé par le consortium OW2 (anciennement ObjectWeb). Il s'agit d'un générateur de code souple, modulaire très simple d'utilisation et parfaitement intégré à l'environnement de développement Eclipse.

Il offre une structuration et une syntaxe adaptées à la navigation et à l'exploitation des modèles fondés sur la technologie EMF. Cet aspect permet ainsi de définir des générateurs simplement en se fondant sur un langage de script conçu à cet effet. Il tire également parti de toutes facilités mises à disposition par Eclipse afin d'éditer, d'exécuter, et déboguer les générateurs de code développés.

Acceleo est disponible à l'adresse *http://www.acceleo.org* et dispose d'une large communauté d'utilisateurs et de contributeurs. Vous pouvez trouver à l'adresse *http://acceleo.org/pages/introduction/fr* de nombreuses ressources et documentations ainsi qu'un forum actif aussi bien en français qu'en anglais.

Le projet Acceleo intègre également un espace communautaire dénommé « la ferme de modules » et regroupant différents modules de génération adressant diverses problématiques et technologies. Le tableau C-5 liste les principaux de ces modules. Nous pouvons distinguer dans cette liste le module mettant en œuvre les technologies Spring, Hibernate et Struts.

Tableau C-5 – Principaux modules de génération de la ferme de modules Acceleo

Module	Description
C# - NHibernate	Générateur d'applications C# basées sur Nhibernate à partir de modèles UML 1.4.
JEE Spring/Hibernate/Struts	Générateur d'applications Java EE fondées sur Spring Framework, Hibernate et Struts à partir de modèles UML2 stéréotypés.
Leonardi	Générateur d'applications LEONARDI à partir de modèles de classes UML.
Topcased Java module et UML 2.1 to Java module	Générateurs de code Java à partir de modèles UML2.
UML 2.1 to C module	Génération de code C à partir de modèles UML2
UML2 to Dolibarr	Générateur Dolibarr à partir de modèles UML2.
Ecore to Python	Générateur Python à partir de modèles Ecore.
Lite JEE	Générateur Java EE basique à partir de modèles UML 2.1 stéréotypés.
PHP	Generateur PHP pear et smarty fondé sur une architecture standarde en trois couches.
WISSS/Webapp Is Simple, Stupid and Secure	Générateur PHP fondé sur le DSL WISSS et créant les couches persistance, métier et présentation en utilisant le Zend Framework.
Zope 3	Generateur d'applications Zope 3 à partir de modèles UML 2.1.
Mindmap	Module de génération de slides XHTML à partir de modèles de carte mentale.
Game DSL	Module de modélisation de jeux vidéo avec génération vers Python.

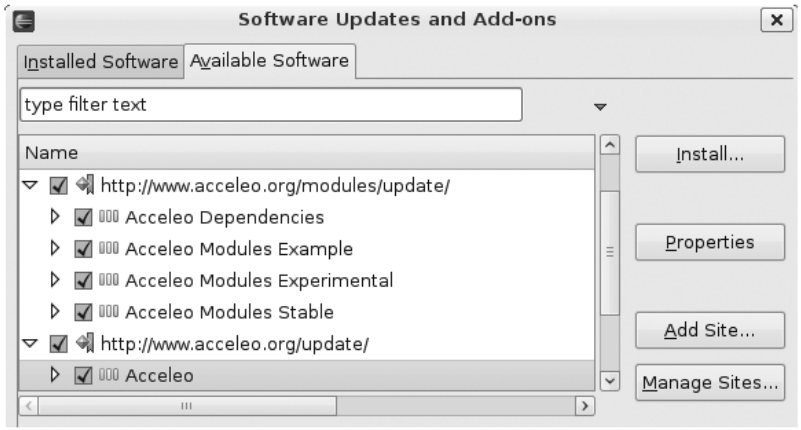
Afin d'installer les greffons Acceleo et Acceleo modules, nous allons utiliser la fonctionnalité intégrée de mise à jour d'Eclipse. En se fondant sur le menu Aide/Software Updates, et du bouton « Add Site » de l'onglet « Available Software », nous allons ajouter les deux sites suivants :

- *http://www.acceleo.org/update/* afin de spécifier le générateur Acceleo en lui-même ;
- *http://www.acceleo.org/modules/update/* afin de spécifier les modules de génération d'Acceleo.

Une fois ces sites spécifiés, nous pouvons sélectionner le moteur Acceleo ainsi que les modules proposés par la ferme de modules et lancer le processus d'installation à l'aide du bouton « Install ».

La figure C-7 décrit la sélection de ces éléments dans l'écran d'Eclipse relatif à la mise à jour.

Figure C-7
Écran de sélection et d'installation des greffons Acceleo



Mise en œuvre

Dans les précédentes sections, nous avons décrit les principaux concepts ainsi que les outils utilisés dans une approche dirigée par les modèles. Nous allons aborder maintenant la manière de les utiliser afin de mettre en œuvre la génération de code dirigée par les modèles dans les développements Java EE fondés sur Spring.

Création du profil

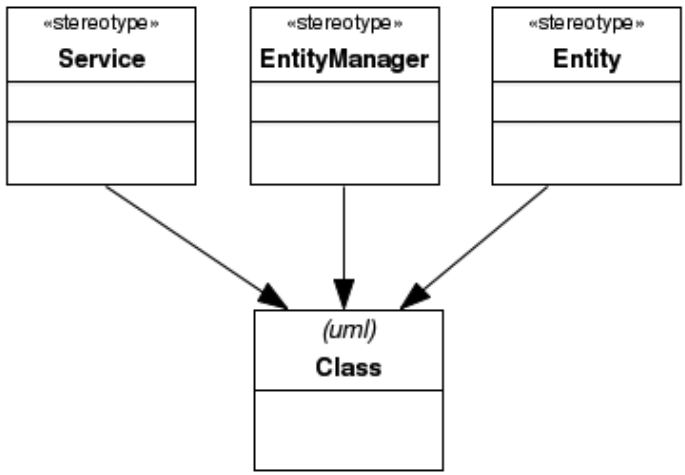
La première étape consiste en la mise en œuvre du profil, ce dernier décrivant les stéréotypes utilisables ainsi que les possibilités d'application sur les éléments des modèles. Ces stéréotypes nous offrent la possibilité de spécialiser des entités UML telles que des classes afin de leur spécifier un rôle plus précis.

Dans notre contexte, le profil définit les différents stéréotypes possédant les caractéristiques suivantes :

- `Service`, permettant d'identifier les classes UML jouant le rôle de services métier.
- `EntityManager`, permettant d'identifier les classes UML jouant le rôle de composants d'accès aux données.
- `Entity`, identifiant les classes du domaine. Il est utilisé afin de déterminer les classes devant être configurées au niveau de la couche d'accès aux données.

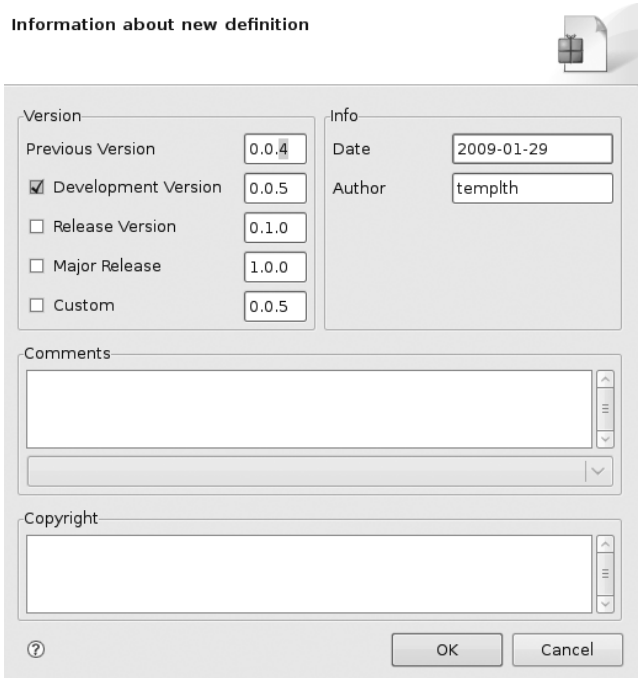
La figure C-8 illustre les différents stéréotypes présents dans le profil ainsi que leur application sur la métaclasse UML `Class` par l'intermédiaire d'une relation d'extension.

Figure C-8
Stéréotypes définis dans le profil Sring



Afin d'être utilisable dans les modèles, le profil doit être « défini ». Cette opération se réalise à sa sauvegarde dans Papyrus par l'intermédiaire de l'écran décrit dans la figure C-9. À ce niveau, une version du profil doit être spécifiée, cette version devant être incrémentée à chaque modification.

Figure C-9
Ecran de définition du profil



Il est à noter que les modules Acceleo mettent à disposition un profil avec des stéréotypes similaires, celui-ci pouvant être utilisé plutôt que d'en recréer un nouveau.

Création du modèle

Le modèle peut maintenant être défini par l'intermédiaire d'un diagramme de classes. Une fois le modèle et le diagramme créés, le profil précédemment décrit doit être appliqué afin de pouvoir utiliser les stéréotypes correspondant sur les classes du modèle.

Cette opération se réalise simplement dans Papyrus par l'intermédiaire des propriétés du modèle dans l'onglet « Profile » de la vue « Properties ». A ce niveau, une zone « Applied profiles » liste les profils appliqués au modèle. Des icônes permettent de gérer cette liste, comme le décrit la figure C-10.

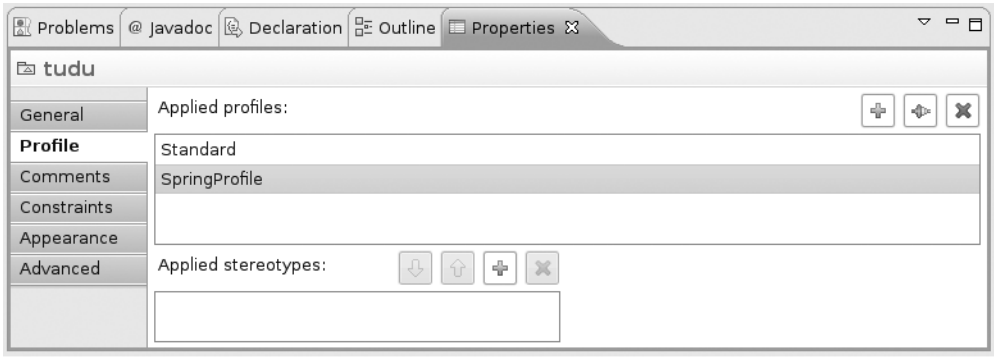
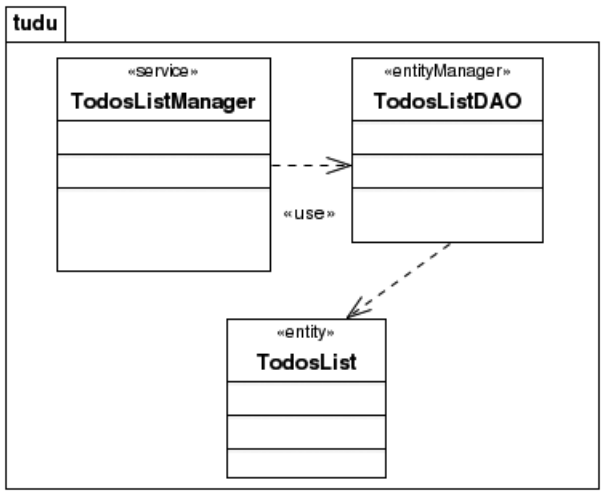


Figure C-10
Zone d'application de profils pour le modèle

Une fois le profil appliqué, il est alors possible de définir les composants de notre application en les spécialisant avec les différents stéréotypes décrits précédemment. Nous avons ici réalisé une modélisation simplifiée issue de l'application Tudu et relative aux `TodosList`.

La figure C-11 décrit le diagramme de classes stéréotypées correspondant.

Figure C-11
Modélisation des composants de l'application



Dans la figure C-11, trois classes ont été définies. La première, `TodosList`, correspond à un objet métier contenant les données de l'application. Ce comportement est spécifié par l'intermédiaire du stéréotype `Entity`. Bien qu'ils ne soient pas présents sur la figure, des associations peuvent être définies entre ces types de classes.

La seconde, `TodosListDAO`, intègre quand à elle les traitements relatifs à la persistance des données et possède donc le stéréotype `EntityManager`. Un lien de dépendance est défini entre ces deux classes afin de spécifier que la classe `TodosListDAO` a en charge la manipulation d'objets de type `TodosList`.

Pour finir, la classe `TodoListsManager` s'appuie sur la classe `TodosListDAO` afin d'offrir un service de manipulation de todolists. Ce lien est défini par une dépendance et ce comportement avec l'application du stéréotype `Service`.

À ce niveau, notre modèle contient toutes les informations nous permettant une génération de code pour les différents éléments de l'application.

Génération à partir du modèle

Maintenant que les mécanismes de mise en œuvre de modèles UML stéréotypés ont été décrits, voyons la manière de les exploiter afin de générer les constituants d'applications Java EE fondées sur le framework Spring.

Au niveau du principe de fonctionnement, sont générés les éléments suivants :

- Pour chaque classe stéréotypée avec `Entity`, une classe Java simple intégrant éventuellement les métadonnées relatives à la persistance.
- Pour chaque classe stéréotypée avec `EntityManager`, une interface avec les méthodes modélisées et une classe d'implémentation contenant les mécanismes d'accès aux données.
- Pour chaque classe stéréotypée avec `Service`, une interface avec les méthodes modélisées et une classe d'implémentation déléguant ses traitements aux DAO dont elle dépend.
- Un fichier Spring contenant les configurations des classes stéréotypées `Service` et `EntityManager` ainsi que leurs dépendances.

Générateur simple

Avec cette approche, un projet de génération Acceleo vide doit être créé à partir des facilités d'Eclipse. Celui-ci est accessible depuis le menu Fichier/Nouveau/Projet. Dans la rubrique Acceleo, il convient de choisir l'élément « Generator Project ».

L'outil Acceleo suit un modèle de programmation déclarative, ce qui signifie qu'un template de génération doit être créé par entité stéréotypée. Un template doit également être mis en œuvre au niveau du modèle lui-même pour la génération de la configuration.

Dans tous les cas, la première étape consiste en la création d'un template de génération en se fondant sur Fichier/Nouveau/Autres puis le choix de l'élément « Empty Generator ». Un

assistant facilite la création en permettant de choisir l'identifiant du metamodèle utilisé, `http://www.eclipse.org/uml2/2.1.0/UML` dans notre cas, et la localisation du fichier.

Une fois créé, le contenu du fichier possède la structuration décrite dans le code suivant :

```
<%
metamodel http://www.eclipse.org/uml2/2.1.0/UML→❶
%>
<%script type="uml.Class" name="service"
         file="<%name%>.java"%>→❷
```

L'entête (❶) définit le métamodèle utilisé et éventuellement l'importation d'entités de traitement complémentaires. Le template est ensuite divisé en script (❷) permettant de structurer les traitements de génération. Par convention, la balise `script` contenant l'attribut `file` correspond au point d'entrée du template. L'attribut `type` permet quant à lui de spécifier sur quel type d'élément du modèle, le template va être déclenché, une classe UML dans notre cas. Un nom peut éventuellement être précisé avec l'attribut `name`.

Afin d'activer un template en tenant compte du stéréotype présent, la méthode `getAppliedStereotypes` (❷) peut être utilisée, cette dernière retournant la liste des stéréotypes appliqués. Un nouveau bloc `script` (❶) doit être alors défini, ce bloc permettant de construire le nom du fichier à générer. Si le stéréotype n'est pas présent, le nom est alors vide et la génération n'est pas effectuée pour cet élément. Ce script est désormais appelé par le script principal par l'intermédiaire de l'attribut `file` (❸).

Le code suivant illustre la mise en œuvre de cette approche :

```
(...)

<%script type="uml.Class" name="nomFichierCible"%>→❶
<%if (getAppliedStereotypes()[name=="Entity"]){%>→❷
<%name%>.java
<%}%>

<%script type="uml.Class" name="service"
         file="<%nomFichierCible%>"%>→❸
```

Maintenant que nous avons initié un template, nous pouvons nous concentrer sur la génération du contenu des classes et interfaces.

Dans les templates, Acceleo permet d'avoir accès aux propriétés et méthodes de l'élément courant du modèle. Par exemple, pour un script, l'élément courant correspond à l'élément sur lequel il s'applique. Dans le cas d'une boucle, cet élément devient alors l'élément de l'itération.

Le code suivant illustre l'utilisation de ce principe avec la spécification du nom d'une classe (❶) et de tous ses attributs (❷) dans un template :

```
<%script type="uml.Class" name="service" file="<%name%>.java"%>
public class <%name%> {→❶
<%for (ownedAttribute){%>→❷
    <%visibility%> <%type.name%> <%name%>;→❷
<%}%>
}
```

Des scripts peuvent être également ajoutés afin de permettre la création des signatures des méthodes en se fondant sur les éléments présents dans le modèle. Ils permettent respectivement de déterminer le type de retour (❶), les différents paramètres (❷) et la signature globale (❸), comme l'illustre le code suivant :

```
<%script type="uml.Operation" name="genRetourMethode"%>→❶
<%if (ownedParameter[direction=="return"].nSize()>0){%>
<%ownedParameter[direction=="return"].type.name%>
<%}else{%>
void
<%}%>

<%script type="uml.Operation" name="genParamMethode"%>→❷
<%for (ownedParameter[direction=="return"]){%>
<%type.name%> <%name%>

<%script type="uml.Operation" name="genSignatureMethode"%>→❸
public <%genRetourMethode%> <%name%>(<%genParamMethode.sep(",")%>)
```

Un autre script peut également être défini afin de récupérer la liste des entités avec lesquelles une classe a des relations de dépendances. Le code suivant illustre un script de ce type dénommé `listeDependances`:

```
<%script type="uml.Class" name="listeDependances"%>
<%clientDependency.supplier[getApplicableStereotypes() √
                                    [name=="EntityManager"]]%>
```

Tous ces éléments peuvent être combinés afin de mettre en œuvre des générateurs construisant des composants Spring, des objets métier ainsi que leur configuration dans le framework.

Modularisation des traitements avec Acceleo

Un ensemble de scripts peut être regroupé dans un fichier avec l'extension mt. Ce fichier peut ensuite être inclu dans des templates afin de les utiliser et de partager ainsi des traitements.

L'inclusion se réalise par l'intermédiaire du mot-clé import dans l'entête des templates après la ligne contenant le mot-clé metamodel.

Le code suivant permet de générer une implémentation simplifiée d'un service métier contenant son nom et sa liaison avec son interface (❶) ainsi que les méthodes (❸) de la classe tout en prenant en compte ses liaisons avec les interfaces des DAO (❷) dont il dépend :

```
<%
metamodel http://www.eclipse.org/uml2/2.1.0/UML
%>

<%script type="uml.Class" name="nomFichierCible"%>
<%if (getAppliedStereotypes()[name=="Service"]){%>
<%name%>.java
<%}%>
```

```
<%script type="uml.Class" name="service"
         file="<%nomFichierCible%>"%>
public class <%name%>Impl implements <%name%> {→❶

<%for (listeDependancesVersEntityManager){%>→❷
    private <%name%> <%name.toL1Case()%>;
<%}%>

<%for (listeDependancesVersEntityManager){%>→❷
    public <%name%> get<%name%>() {
        return <%name.toL1Case()%>;
    }

    public void set<%name%>(<%name%> <%name.toL1Case()%>) {
        this.<%name.toL1Case()%> = <%name.toL1Case()%>;
    }
<%}%>

<%for (ownedOperation){%>→❸
    <%genSignatureMethode%> {
    }
<%}%>
}

(...)
<%script type="uml.Class"
         name="listeDependancesVersEntityManager"%>
<%clientDependency.supplier[getApplicableStereotypes() ➥
                                        [name=="EntityManager"]]%>
```

En considérant que les implémentations des DAO utilisent JPA et se fondent sur une instance d'`EntityManagerFactory`, le code suivant décrit un template permettant de construire la configuration Spring des services (❶), DAO (❸) ainsi que leurs relations (❷) :

```
<%
metamodel http://www.eclipse.org/uml2/2.1.0/UML
%>

<%script type="uml.Model" name="entity"
         file="applicationContext.xml"%>
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="(...)">

<!-- Services -->
<%for (listeServices){%>→❶
    <bean id="<%name.toL1Case()%>" class="<%name.toL1Case()%>Impl">
<%for (listeDependancesVersEntityManager){%>→❷
        <property name="<%name.toL1Case()%>"
                  ref="<%name.toL1Case()%>"/>
```

```
<%}%>
</bean>
<%}%>

<!-- EntityManagers -->
<%for (listeEntityManager){%>→❸
    <bean id="<%name.toL1Case()%>" class="<%name.toL1Case()%>Impl">
        <property name="entityManagerFactory"
                  ref="entityManagerFactory"/>
</bean>
<%}%>

</beans>

<%script type="uml.Model" name="listeServices"%>→❶
<%eAllContents("uml.Class")[getAppliedStereotypes()➥
                                    [name=="Service"]]%>

<%script type="uml.Class"→❷
         name="listeDependancesVersEntityManager"%>
<%clientDependency.supplier[getApplicableStereotypes()➥
                                    [name=="EntityManager"]]%>

<%script type="uml.Model" name="listeEntityManager"%>→❸
<%eAllContents("uml.Class")[getAppliedStereotypes()➥
                                    [name=="EntityManager"]]%>
```

Une fois tous ces templates mis en œuvre, une chaîne de lancement doit être créée afin de permettre l'exécution de la génération. À cet effet, il convient d'utiliser le menu Fichier/Nouveau/Autres et de choisir dans la rubrique Acceleo l'élément « Chain ». Un assistant nous aide alors à créer cette entité par la saisie des différentes informations nécessaires :

- L'identifiant du métamodèle utilisé. Il s'agit dans notre cas d'UML2 avec l'identifiant `http://www.eclipse.org/uml2/2.1.0/UML`.
- Le fichier du modèle utilisé, ce dernier possédant l'extension `uml` dans notre contexte.
- Les différents templates mis en œuvre dans la génération, ces derniers possédant l'extension `mt`.
- Le projet dans lequel créer la chaîne ainsi que son nom.

La figure C-12 décrit les différents constituants sous le nœud « Repository » ainsi que les différentes générations mises en œuvre sous le nœud « Action Set ».

Une fois la chaîne créée, la sélection de l'entrée « Launch » dans le menu contextuel correspondant permet de lancer la génération de code, comme l'illustre la figure C-13.

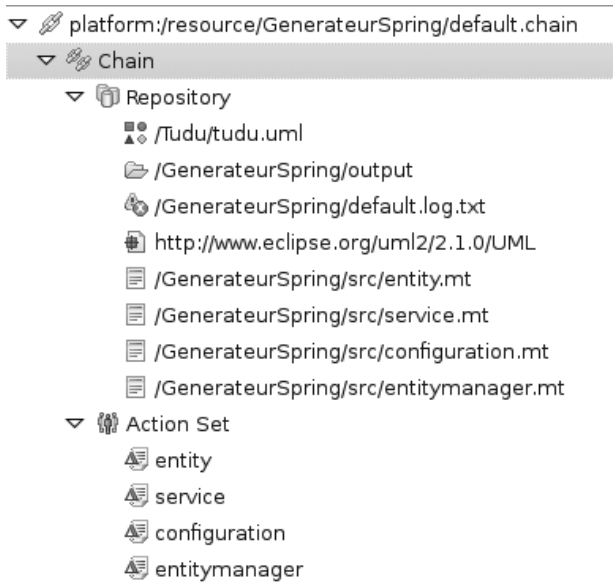


Figure C-12
Contenu de la chaîne d'exécution de la génération

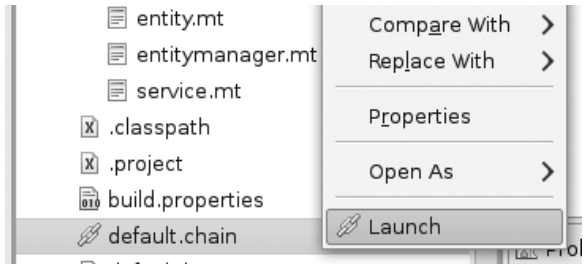


Figure C-13
Lancement de la génération de code avec Acceleo

Utilisation du module Spring d'Acceleo

Dans ce cas, les templates de génération n'ont plus à être créés. Ils sont en effet mis à disposition par l'intermédiaire du module Spring. Seul un générateur doit être créé en se fondant sur un module existant. La création d'un générateur de ce type se fonde sur menu Fichier/Nouveau/Autres et la sélection de l'élément « Module Launcher » dans la rubrique Acceleo. Un assistant permet de sélectionner le module à utiliser et de préciser le nom de la chaîne de lancement créée.

La figure C-14 décrit l'écran de sélection du module Acceleo de l'assistant précédemment cité. Nous avons choisi ici le module « Spring Business Layer Generator (Spring + Hibernate + Hessian WS ».

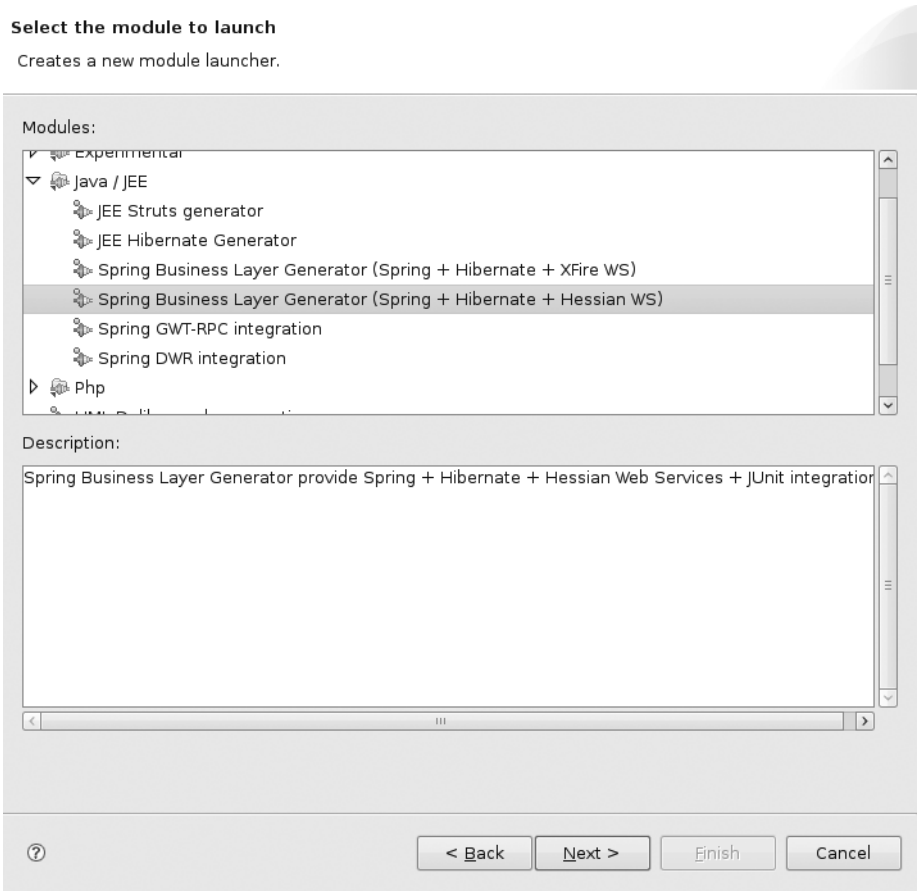


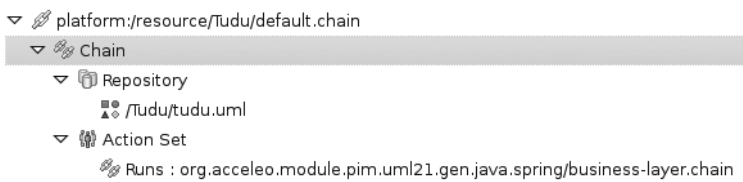
Figure C-14

Sélection du module de génération dans l'assistant « Module Launcher »

Dans ce cas, le contenu de la chaîne d'exécution créée est décrit dans la figure C-15.

Figure C-15

Contenu de la chaîne d'exécution de la génération à partir d'un module Acceleo



L'exécution de la génération se réalise de manière classique en se fondant sur la chaîne créée au terme de l'assistant.

Il est à noter que le module se fonde sur les mêmes noms de stéréotypes que précédemment, excepté pour les DAO avec les stéréotypes `Dao`. Ainsi, un fichier de propriétés doit être créé au même niveau que la chaîne de lancement afin de pouvoir utiliser notre propre profil dénommé `SpringProfile`, comme l'illustre le code suivant :

```
Service=SpringProfile::Service
Dao=SpringProfile::EntityManager
Entity=SpringProfile::Entity
```

La mise en œuvre du fichier de propriétés est requise uniquement lors de l'utilisation d'un profil spécifique afin de fournir une table de correspondance. Dans le cas de l'utilisation du profil mis à disposition par le module, ce fichier n'est pas nécessaire.

Les éléments générés correspondent aussi bien aux différents composants applicatifs (services métier et composants d'accès aux données), aux objets métier ainsi qu'aux différents fichiers de configuration de Spring. Les composants applicatifs utilisent les différents mécanismes et supports classiques du framework Spring et mettent en œuvre le framework Hibernate pour l'accès aux données. Au niveau de ces fichiers, des configurations de beans sont automatiquement ajoutées afin de rendre accessibles de manière distante les services en fonction des technologies choisies pour le générateur (Hessian, XFire, GWT ou DWR).