Transformations de modèles UML

1 Échauffement : réusinages

Une bonne pratique de développement est d'alterner les ajouts de fonctionnalités avec des étapes de réusinage (refactoring). Les réusinages se sont tout d'abord appliqués au code : un réusinage consiste à modifier le code pour le rendre plus lisible, plus facile à comprendre et à modifier, sans changer ses fonctionnalités. Au sein d'une approche IDM, on effectue également des réusinages, mais au niveau des modèles. Nous allons écrire ici quelques méthodes de réusinage de modèle UML.

Question 1. Ecrivez les réusinages suivants :

1. Écrire une méthode de réusinage qui permet de déplacer une classe d'un package à un autre.
2. Écrire une méthode de réusinage qui remplace un attribut public par un attribut privé et une paire d'accesseurs (en lecture et en écriture).
3. Écrire une méthode de réusinage qui fait "remonter" une méthode dans une super-classe. Si la superclasse donnée ne correspond pas à une super-classe de la classe origine, ou si la méthode donnée ne correspond pas à une méthode de la classe d'origine, ou encore si une méthode concrète du même nom que la méthode à faire remonter existe déjà dans la super-classe spécifiée, alors le réusinage échoue.

Côté technique

Placez vous dans un "empty EMF project". Vous utiliserez le fichier LoadUML.java présent sous Moodle. Ce fichier vous donne de quoi charger et sauver des modèles UML, et illustre l'appel de ces méthodes sur un petit modèle UML presque vide. Nous n'aurons pas besoin de générer le code pour le métamodèle UML, en effet, celui-ci est déjà présent dans un plug in eclipse. Les imports présents dans le fichier fourni ne devraient pas compiler directement. Au niveau des imports fautifs concernant xmi et UML, utilisez l'aide d'eclipse "fix project set up" et ensuite acceptez d'ajouter le plug in aux "required bundles".

Pour tester vos méthodes de réusinage, vous aurez besoin de modèles UML. Le plus simple est d'utiliser le modeleur Papyrus. Pour installer Papyrus, passez par l'installation de nouveaux composants de modélisation (install neww modelling components) du menu help. Le test de vos méthodes de réusinage consistera à :

- charger un modèle UML "d'entrée" stocké par exemple dans un fichier A.uml
- récupérer dans ce modèle les éléments sur lesquels vous allez appliquer votre réusinage (une classe, une méthode, un package, ...)
- appliquer votre réusinage aux éléments de modèle choisis
- sauvegarder le modèle réusiné dans un fichier par exemple Areusine.uml
- vérifier ("à l'oeil") que le réusinage a bien fonctionné comme prévu.

Créer un modèle Papyrus

- Dans le répertoire model de votre projet : New, other, Papyrus Model
- choisir UML comme langage
- choisir le nom du modèle
- choisir le diagramme de classes et cocher la case "a UML model with basic primitive types"

2 Application du patron de conception State

2.1 Métamodèle pour les machines à états UML

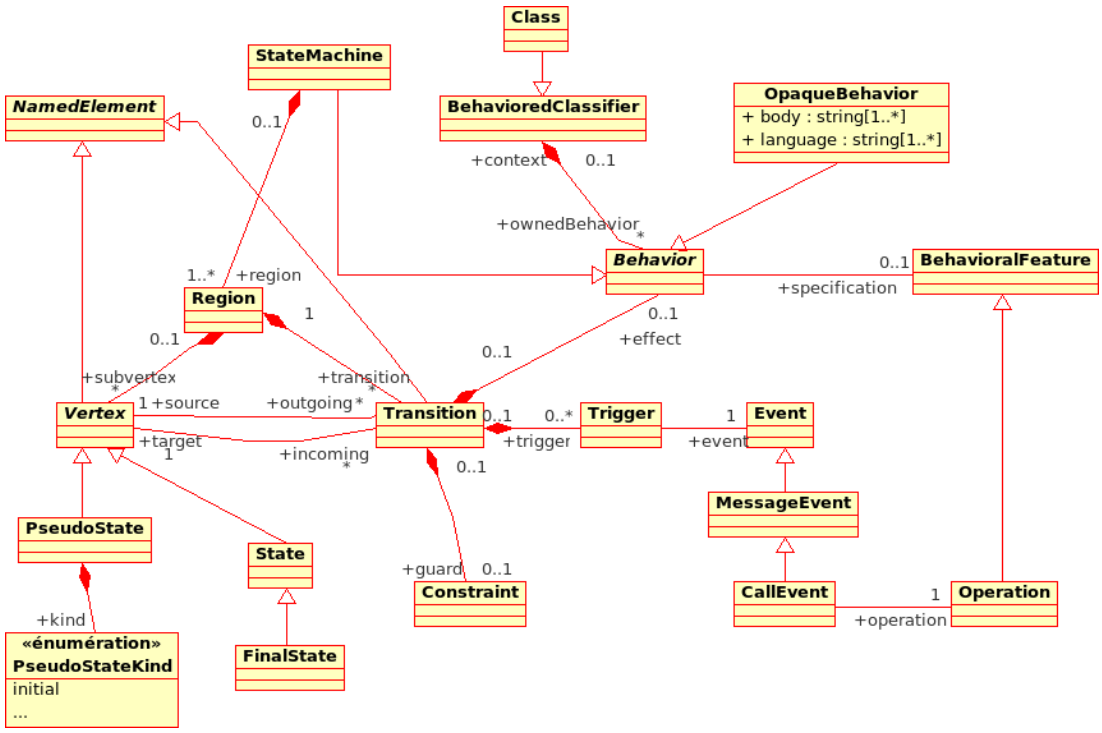

FIGURE 1 – Extrait du métamodèle UML pour les machines à états

Un extrait drastiquement simplifié du métamodèle des machines à états UML vous est donné à la figure 1. Vous y reconnaîtrez des éléments du métamodèle qui correspondent à la partie statique d'UML étudiée en cours et en TD, notamment les métaclasses `Class`, `NamedElement`, et `Operation`. Les liens qu'entretiennent ces métaclasses n'ont pas été explicités mais bien sûr ils existent néanmoins.

Une machine à états est un comportement. À ce titre, elle est reliée à un contexte (un classifieur à comportement) qui dans notre cas sera une classe (ce pourrait être aussi un cas d'utilisation par exemple). La machine à états peut accéder à toutes les propriétés et associations de son contexte. Une machine à états est composée de régions. Une région se compose de sommets (`Vertex`) et de transitions. `Vertex` est une classe abstraite factorisant les états et les pseudo-états. Un vertex est lié à ses transitions sortantes (outgoing) et entrantes (incoming).

Les pseudo-états peuvent être de plusieurs sortes (jonction, branchement, etc) mais la seule sorte qui nous intéresse ici est `initial` qui permet de représenter l'état initial de la machine à états (représenté graphiquement par un disque noir). La métaclasse `State` permet de représenter les états de la machine à états (graphiquement représentés par des rectangles à coins arrondis). `FinalState` est la métaclasse qui permet de représenter les états finaux d'une machine à états (graphiquement représentés par un disque noir entouré d'un cercle)

Chaque transition relie 2 Vertex (`Source` et `Target`). Une transition se compose d'une éventuelle garde (sous forme d'une contrainte) et d'un ensemble de déclencheurs ou gachettes (`trigger`). Chaque déclencheur est lié à l'événement permettant de le déclencher. On ne s'intéresse ici qu'aux événements `CallEvent`, qui représentent les événements liés aux appels de méthode. On transite d'un vertex V1 à un vertex V2 liés par une transition T quand l'un des événements des déclencheurs liés à la transition T est levé et quand la garde est vérifiée. S'il n'y a pas de déclencheurs, on dit que la transition est spontanée, et on transite dès que la garde est vraie. S'il n'y a pas de garde, cela est équivalent à une garde valant vrai. Une transition se compose également d'un éventuel effet, sous forme d'un comportement. Cela correspond au comportement à déclencher lorsque la transition est tirée (i.e. lorsque l'on transite effectivement par la transition). Ce comportement peut être spécifié par une `BehavioralFeature` comme par exemple une opération. Un comportement peut être un comportement opaque, qui est spécifié par 2 chaînes : `body` (la description du comportement) et `language` (le langage utilisé pour la description). On peut donner plusieurs descriptions dans plusieurs langages, d'où les tableaux de chaînes pour `body` et `language`.

2.2 Transformation de modèles pour State

Pour simplifier, dans cette partie, nous ne travaillerons qu'avec des machines à états à une seule région.

On souhaite écrire une transformation de modèle qui, à partir d'une classe disposant d'une machine à états, génère la structure statique engendrée par l'application du patron de conception State à la classe. Vous n'avez pas besoin de connaître le patron State pour réussir cet exercice. Lors de l'application du patron de conception à une classe A :

- Une classe abstraite nommée EtatA est créée, reliée par une association à la classe A : la classe A connaît son état. La classe EtatA possède toutes les signatures de méthodes dont le comportement dépend de l'état dans lequel se trouve une instance de A. Il s'agit ici de toutes les méthodes présentes comme déclencheurs dans les transitions de la machine à états. Ces méthodes sont abstraites dans EtatA.
- Pour chaque état de la machine à états, une sous-classe de la classe EtatA est créée, qui possède les mêmes méthodes que la classe mère. On ne s'intéressera pas ici au corps de ces méthodes (dans lequel le comportement adéquat doit être implémenté en fonction de l'effet de la transition, et qui doit permettre le changement d'état).

Question 2. Ecrivez une méthode prenant en paramètre une classe, et qui retourne les machines à état la décrivant.

Question 3. Ecrivez une méthode prenant en paramètre une machine à états et qui vérifie qu'elle est correctement formée pour notre exercice, c'est-à-dire qu'elle ne contient qu'une seule région.

Question 4. Ecrivez une méthode prenant en paramètre une machine à état bien formée pour notre exercice, et qui retourne la liste des états la composant.

Question 5. Ecrivez une méthode prenant en paramètre une machine à état bien formée pour notre exercice, et qui retourne la liste des opérations se trouvant comme trigger dans la machine à état.

Question 6. Ecrire une méthode qui appliquerait le patron State à une classe donnée.

Question 7. Ecrire une méthode prenant en paramètre le nom de fichier d'un modèle UML, et qui applique le patron State à toutes les classes (du modèle dont le nom est passé en paramètre) dotées d'une machine à états.

Côté technique

Pour tester vous aurez besoin de machines à états associées à des classes, quelques explications ici sur comment procéder avec Papyrus.

Pour créer une machine à états associée à une classe :

— Dans le model explorer, se placer sur la classe contexte
— clic droit , new diagram
— Choisir State Machine Diagram

Pour ajouter un trigger correspondant à un appel de méthode :

— Se placer sur les properties de la transition sur laquelle on veut ajouter le trigger
— Demander à créer un nouvel événement
— Choisir CallEvent
— Dans container : choisir l'élément qui doit contenir le nouvel événement, ici on pourra choisir RootElement par exemple, ou n'importe quel package
— puis choisir l'opération souhaitée.