Master Informatique - Université de Montpellier - Ingénierie Dirigée par les Modèles - 2018

DSL internes pour les *feature models*
Transformation de modèles

1 Contexte : les *feature models*

Les *feature models* sont une famille de langages de description visuels qui permettent de décrire un ensemble de caractéristiques ainsi que des relations entre ces caractéristiques. Ils sont utilisés pour représenter un ensemble de produits appartenant à une même famille, de manière compacte et compréhensible.

Un *feature model* organise de manière hiérarchique un ensemble de caractéristiques dans un arbre : la caractéristique racine est la plus générale (elle représente souvent le nom de la famille de produits modélisée) alors que les caractéristiques les plus basses dans la hiérarchie sont les plus spécialisées. Une arête de l'arbre représente donc une relation de raffinement entre une caractéristique mère et une (plusieurs) caractéristique(s) fille(s). Un produit correspond à un ensemble de caractéristiques sélectionnées dans cet arbre. Ces arêtes peuvent être décorées pour contraindre la sélection. Si on sélectionne une caractéristique qui possède une sous-caractéristique : une arête *optionnelle* définit que la sous-caractéristique n'est pas obligatoirement sélectionnée, et une arête *obligatoire* force la sous-caractéristique à être sélectionnée. On trouve aussi des contraintes sur les groupes de caractéristiques. Si on sélectionne une caractéristique qui possède un groupe de sous-caractéristiques : dans un groupe *or*, au moins une des sous-caractéristiques doit être sélectionnée, alors que dans un groupe *xor*, exactement une doit être sélectionnée. Les arêtes correspondant à ces 4 relations sont exposées dans la Figure 1 (gauche). Des contraintes qui ne peuvent être exprimées dans la hiérarchie (implications, exclusions) peuvent être rajoutées textuellement.

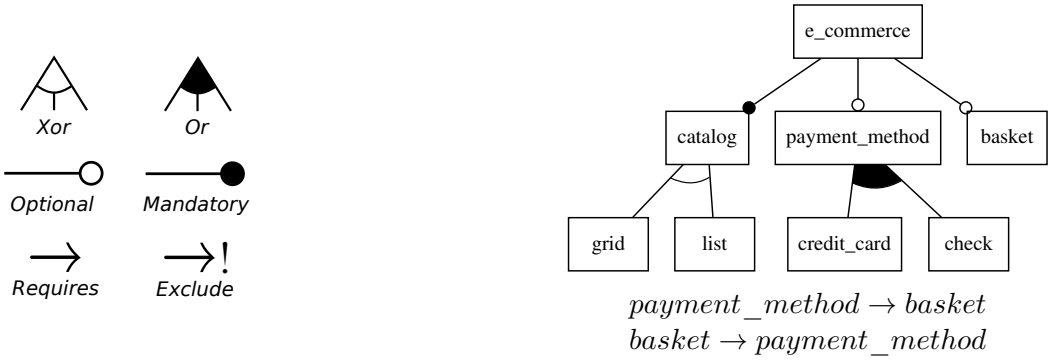

FIGURE 1 – Exemple de *feature model* (droite) et ses différentes relations (gauche)

La Figure 1 (droite) représente un exemple de *feature model* sur une famille d'applications de e-commerce. On peut lire que : une application de e-commerce possède obligatoirement un catalogue ; ce catalogue peut être affiché sous forme de grille ou de liste, mais pas les deux ; l'application peut optionnellement proposer des méthodes de paiement ; elle peut proposer un paiement par carte, par chèque, ou bien les deux ; une application de e-commerce peut éventuellement proposer une gestion de panier ; si la gestion de panier est sélectionnée, l'application doit posséder des méthodes de paiement, et inversement.

La configuration $\{e\_commerce, catalog, grid, basket, payment\_method, credit\_card\}$ respecte l'ensemble des contraintes du modèle, et représente donc une variante possible de la famille de produits. Cependant, la configuration $\{e\_commerce, catalog, grid, basket\}$ ne représente pas une variante valide, car la contrainte $basket \rightarrow payment\_method$ n'est pas respectée. La liste des 8 configurations valides du modèle de la Figure 1 sont listées dans la Table 1.

TABLE 1 – Liste des 8 configurations décrites par la *feature model* de la Fig. 1

	e_commerce	catalog	grid	list	paym_method	credit_card	check	basket
c1	x	x	x					
c2	x	x		x				
c3	x	x	x		x	x		x
c4	x	x	x		x		x	x
c5	x	x	x		x	x	x	x
c6	x	x		x	x	x		x
c7	x	x		x	x		x	x
c8	x	x		x	x	x	x	x

Les *feature models* sont le standard *de facto* pour modéliser et représenter les connaissances liées à la variabilité d'une famille de produits. Ils permettent une meilleure gestion des éléments communs d'un ensemble de variantes, et des éléments spécifiques de certaines variantes, favorisant ainsi la réutilisation : les coûts et temps de développement sont alors réduits, et la qualité des produits ainsi obtenus est meilleure.

2 Un méta-modèle pour les *feature models*

La Figure 2 représente un méta-modèle de *feature models*. Il se situe au niveau M2 de l'architecture d'IDM. Le niveau M0 (monde réel), représente l'ensemble des systèmes logiciels similaires appartenant à une même famille de produits logiciels. Au niveau M1, on retrouve les modèles capables de décrire l'ensemble des ces systèmes logiciels, comme par exemple les *feature models* : tout ensemble de systèmes logiciels similaires peut donc être représenté par un *feature model*. Au niveau supérieur (M2) se situe alors les méta-modèles de ces *feature models*, permettant de représenter tout *feature model*.

Question 1 : Ouvrez dans Eclipse un projet Java classique. Implémentez le méta-modèle de la Figure 2 : créez les classes, attributs et méthodes nécessaires à son utilisation.

Question 2 : Créez une nouvelle classe `TestFM` possédant une méthode `main`. Utilisez les classes créées précédemment pour instancier le *feature model* représenté dans la Figure 1.

3 Génération de DSL internes depuis un métamodèle

Nous allons à présent définir un *domain-specific language* (DSL) interne pour représenter les *feature models*. Un DSL s'oppose à un *general purpose language* (GPL) tels que C++, Java etc., dont le but est de fournir une solution générique pour résoudre n'importe quel problème. Le but d'un DSL est d'être plus léger, moins verbeux et donc plus compréhensible par un utilisateur. Il est aussi plus spécifique à la représentation des problèmes d'un domaine particulier. Des exemples de DSL que vous avez utilisé sont Latex et SQL.

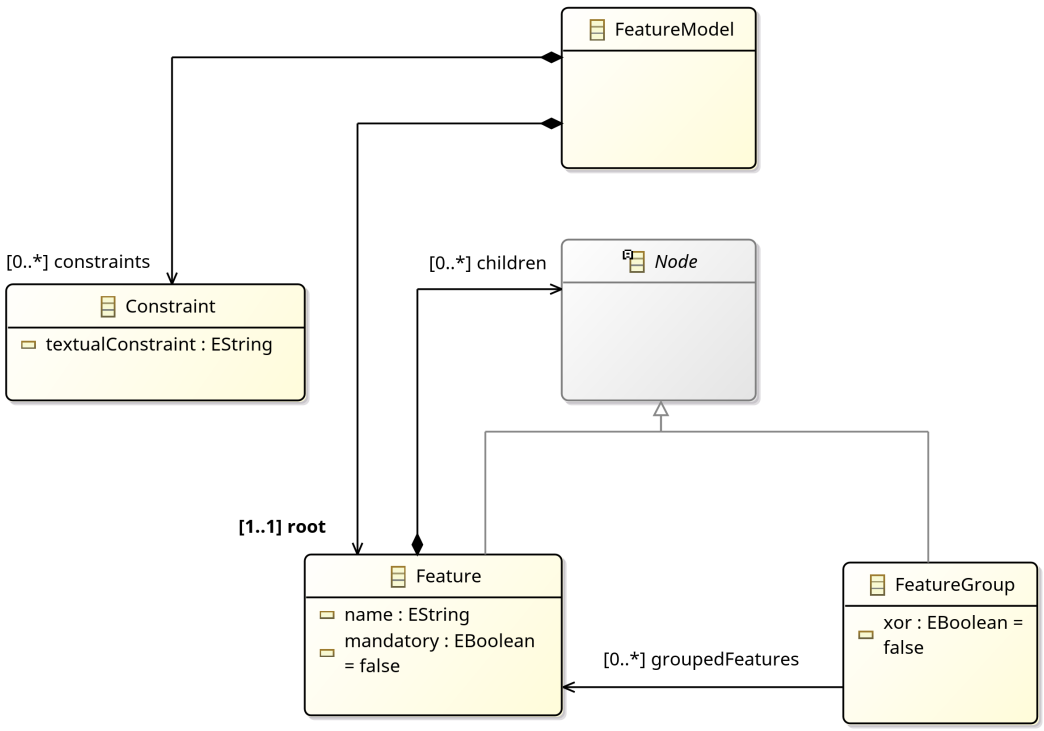

FIGURE 2 – Méta-modèle de *feature models*

On appelle *DSL externe* un langage définit de manière indépendante aux autres langages de programmation, qui est alors libre de toutes contraintes syntaxiques. On appelle *DSL interne* un langage spécifique basé sur la grammaire d'un langage existant. Typiquement, ces langages internes utilisent un sous ensemble des éléments de la syntaxe et des caractéristiques du langage hôte. Les programmes écrits avec un DSL interne sont considérés comme valides dans le langage hôte. Cela permet de restreindre les éléments du langage hôte qui vont être utilisés pour modéliser les problèmes du domaine : le langage est plus lisible, plus simple et il "cache" ainsi les détails linguistiques qui ne sont pas nécessaires à la résolution des problèmes. La réalisation d'un DSL interne permet de bénéficier des outils du langage hôte, et demande donc moins d'investissement pour sa conception et son utilisation.

En général, les mêmes approches et patterns sont utilisés pour implémenter des DSL internes. Par exemple, les DSL internes basés sur Java utilisent régulièrement les *design patterns* et *programming language idioms* suivant :

— method chaining
— static factory method
— builder pattern
— variadic function
— intermediate object

La construction de DSL internes se réalise souvent de manière manuelle ou semi-manuelle par un concepteur. Or, la réutilisation de patrons et de méthodes d'implémentation ouvre la voix vers l'automatisation de la création de DSL internes.

Dans ce qui suit, nous allons implémenter un DSL interne basé sur Java, à partir d'un diagramme de classes UML et en utilisant le patron *method chaining* (Wikipédia). Un exemple de modèle avec deux implémentations (la première sans chaînage de méthodes, comme nous l'avons fait dans les questions précédentes,

et la seconde avec chaînage de méthodes) est donné à la Figure 3.

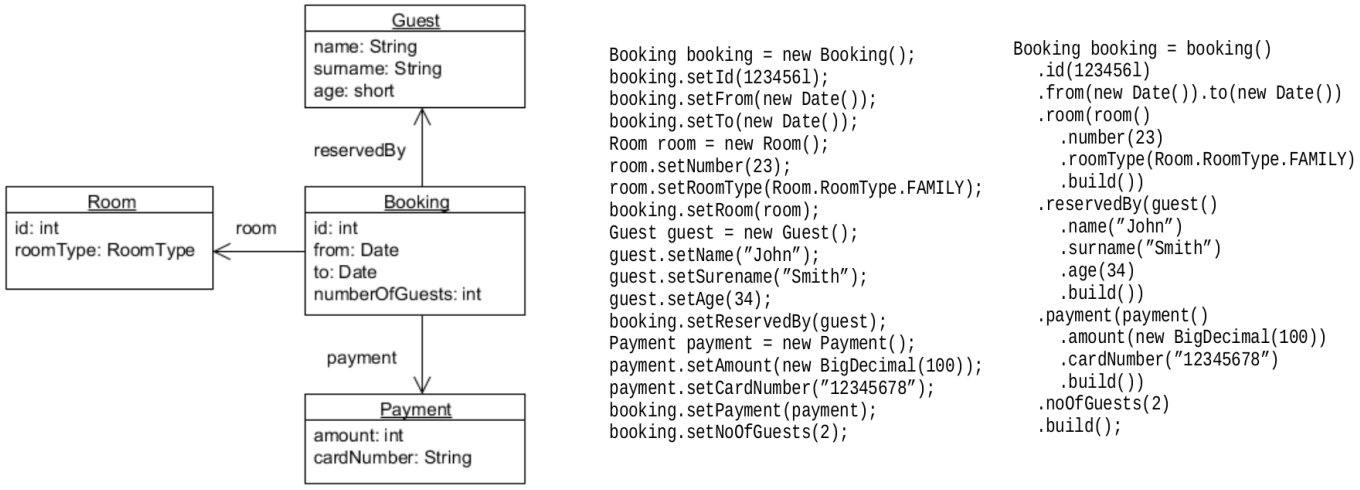

FIGURE 3 – Exemple d'implémentation d'un DSL interne en Java depuis un digramme de classes UML

Question 3 : Implémentez un DSL interne représentant les mêmes *feature models* que précédemment mais en utilisant le chaînage de méthodes. Instanciez le *feature model* de la Figure 1 avec ce DSL.

Question 4 : Identifiez les règles de transformation (`Model-to-text`) pour passer du méta-modèle de la Figure 2 à votre implémentation. Généralisez pour tout méta-modèle de ce type.

Question 5 : Implémentez ces règles de transformation. Pour cela, générez le diagramme de classe UML associé à la Figure 2. Puis, utilisez le `LoadUML.java` du TP précédent pour manipuler ce diagramme et définir les règles de transformation. Pour chaque diagramme UML, générez un nouveau dossier ayant le même nom que le modèle. Pour chaque classe du diagramme, créez un nouveau fichier java dans lequel sera généré automatiquement le code nécessaire à l'utilisation du DSL interne.

4 FAMILIAR, un autre méta-modèle

`FAMILIAR` (pour *FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) est un *domain-specific language* pour la manipulation de *feature models*. Il permet entre autre de créer des *feature models* conformes à ceux décrits en Section 1, de réaliser des opérations de modification (édition, fusion de modèles) ainsi que des opérations d'analyse telles que la vérification de la validité du modèle, la détection des caractéristiques inutilisées, la génération des configurations possibles, la comparaison de modèles, etc.

4.1 Téléchargement et installation

Le langage `FAMILIAR` s'accompagne d'un environnement disponible en version graphique, téléchargeable sous la forme d'un fichier JAR :

Version graphique :
`http://mathieuacher.com/pub/FAMILIAR/releases/FML-environment-1.1.jar`

Dans ce TP, nous utiliserons cet outils pour visualiser nos *feature models* instanciés.

Pour utiliser l'environnement graphique, lancez simplement la commande suivante dans un terminal :

```
java -jar FML-environment-1.1.jar
```

La documentation de `FAMILIAR` peut être trouvée ici :
`https://github.com/FAMILIAR-project/familiar-documentation/tree/master/manual`

4.2 Syntaxe

Pour instancier un nouveau *feature model* avec `FAMILIAR`, il faut utiliser le constructeur "`FM`" :

```
fm1 = FM (A : B C [D]; B: (E|F) ; D : (J|K)+ ; (!C | D) ; )
```

— A est la caractéristique **racine**
— B, C et D sont les caractéristiques filles de A : B et C sont **obligatoires**, D est **optionnelle**
— E et F forment un **groupe XOR**, et sont les caractéristiques filles de B
— J et K forment un **groupe OR**, et sont les caractéristiques filles de D
— (!C | D) est l'équivalent de la CTC **requires** : (C $\rightarrow$ D)
— Une contrainte d'exclusion (**excludes**) entre C et D s'écrirait donc (!C | !D)

Les détails de la notation textuelle interne sont disponibles au lien suivant :
`https://github.com/FAMILIAR-project/familiar-documentation/blob/master/manual/featuremodel.md`

4.3 Environnement graphique

Lancez l'environnement graphique sur un terminal. Vous devez voir apparaître une fenêtre dans laquelle un *feature model wiki* est déjà défini. Notez aussi la présence d'une console en bas de la fenêtre.

Question 6 : (Prise en main) Visualisez le *feature model* `fm1` représenté en langage `FAMILIAR` dans la sous-section précédente. Pour cela, copiez entièrement la ligne débutant par "`fm1 = FM(A ...`" et collez-la dans la console de l'environnement. Validez l'instanciation en appuyant sur la touche `entrée`. Ensuite, dans le menu en haut de la fenêtre, cliquez sur `Display » Display all ...` : vous devriez voir apparaître un nouvel onglet contenant une version graphique du *feature model* `fm1`.

5 Transformation `Model-to-Model`

Une transformation *Model-to-Model* consiste en la création automatique de modèles cibles depuis des modèles sources. C'est une transformation depuis une syntaxe abstraite vers une autre syntaxe abstraite.

Nous voulons pouvoir transformer n'importe quel *feature model* instancié depuis notre méta-modèle (i.e., avec notre DSL) en une instance équivalente, mais cette fois conforme au méta-modèle de `FAMILIAR`. Nous allons définir une méthode dans la classe `FeatureModel` qui va s'occuper de la transformation : cette méthode renverra une chaîne de caractères décrivant l'instance au format `FAMILIAR`.

Question 7 : Ajoutez des méthodes dans les classes de votre DSL pour extraire automatiquement une chaîne de caractères permettant de décrire ses instanciations au format `FAMILIAR`.

Question 8 : Testez votre transformation sur votre instanciation précédente (Question 3). Affichez la chaine de caractères dans Eclipse afin de la copier, puis coller-la dans la console de l'environnement graphique de `FAMILIAR`. Visualisez le *feature model*. Correspond-il bien à la Figure 1 ?

Questions 9 : À présent, instanciez le *feature model* de la Figure 4. Tranformez-le au format `FAMILIAR` et visualisez-le avec l'environnement graphique.

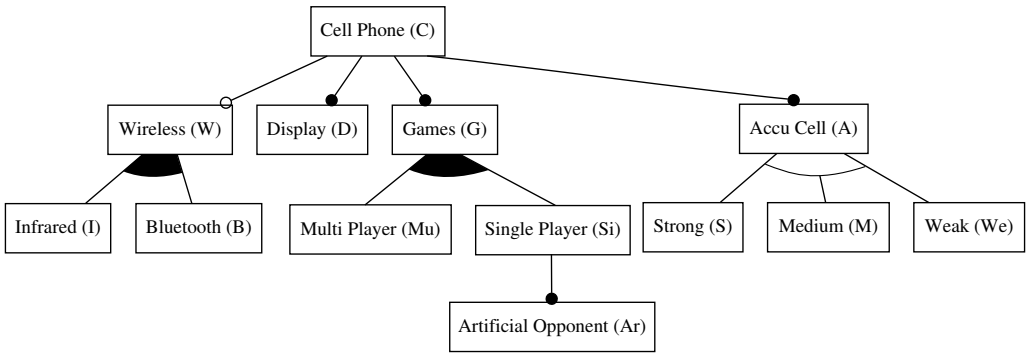

FIGURE 4 – *feature model* sur des téléphones portables