Génération de code

Objectifs

- Présenter les règles de génération de code Java à partir d'un modèle UML
- Présenter les problèmes du cycle reverse/génération
- Présenter les cycles de développement avec UML

D'UML à Java

Nous avons comparé au chapitre 3 la sémantique UML avec la sémantique Java. Nous avons alors proposé des règles de correspondance permettant de construire automatiquement une partie d'un modèle UML à partir d'une application Java. Soulignons que ces règles représentent une façon parmi d'autres de passer de Java vers UML. Elles constituent un des ponts sémantiques de Java vers UML.

Dans le présent chapitre, nous allons proposer un pont sémantique inverse, permettant de construire automatiquement une application Java à partir d'un modèle UML. Ce pont permettra de réaliser l'opération de génération de code Java à partir de modèles UML.

Pour établir ce pont, nous devons définir un ensemble de règles de correspondances des concepts UML vers les concepts Java.

Règles de correspondance UML vers Java

1. À toute classe UML doit correspondre une classe Java portant le même nom que la classe UML.

2. À toute interface UML doit correspondre une interface Java portant le même nom que l'interface UML.

3. À toute propriété d'une classe UML doit correspondre un attribut appartenant à la classe Java correspondant à la classe UML. Le nom de l'attribut doit être le même que le nom de la propriété. Le type de l'attribut doit être une correspondance Java du type de la propriété UML. Si le nombre maximal de valeurs pouvant être portées par la propriété est supérieur à 1, l'attribut Java est un tableau.

4. À toute opération d'une classe UML doit correspondre une opération appartenant à la classe Java correspondant à la classe UML. Les noms des opérations doivent être les mêmes. Étant donné que Java ne supporte que les directions `in` et `return`, si l'opération contient des paramètres de direction `out` ou `inout`, nous considérons qu'il n'est pas possible de générer du code Java. Sinon, pour chaque paramètre de l'opération UML dont la direction est `in` doit correspondre un paramètre de l'opération Java. Les noms des paramètres doivent être les mêmes. Les types des paramètres doivent être une correspondance Java des types des paramètres UML. Si l'opération UML contient un paramètre de direction `return`, l'opération Java doit définir un retour qui lui correspond. Si l'opération UML ne contient pas de paramètre de direction `return`, l'opération Java retourne `void`.

5. Si une classe UML A est associée à une classe UML B et que l'association soit navigable, il doit correspondre un attribut dans la classe Java correspondant à la classe UML A. Le nom de l'attribut doit correspondre au nom du rôle de l'association. Le type de l'attribut doit être une correspondance Java de la classe UML B associée. Si l'association spécifie que le nombre maximal d'objets pouvant être reliés est supérieur à 1, l'attribut Java est un tableau. Si l'association n'est pas navigable, nous considérons qu'il n'est pas possible de générer du code Java.

6. Si une classe UML hérite d'une autre classe UML, il doit correspondre une relation d'héritage (`extends` en Java) entre les classes Java correspondantes. Comme Java ne supporte pas l'héritage multiple, si une classe UML hérite de plusieurs autres classes UML, nous considérons qu'il n'est pas possible de générer du code Java.

7. Si une classe UML réalise une ou plusieurs interfaces UML, il doit correspondre une relation de réalisation entre la classe et les interfaces Java correspondantes.

8. Si une classe UML est contenue dans un package, la classe Java correspondante doit déclarer qu'elle appartient à un package Java. Le nom du package Java doit être le même que le nom du package UML.

9. Si un package UML importe un autre package UML, toutes les classes Java correspondant aux classes UML incluses dans le package UML doivent déclarer un import Java vers toutes les classes Java correspondant aux classes incluses dans le package UML importé.

Ces règles de correspondances ne prennent pas en compte les traitements associés aux opérations UML, car ceux-ci ne sont pas nativement définis dans les modèles. Le code

Java généré ne contient donc que des squelettes de code sans comportement réel associé aux méthodes. De ce fait, l'application générée ne pourra jamais être exécutée.

Lorsque nous avons défini notre opération de Reverse Engineering, nous avons précisé que le code des traitements associés aux opérations était intégré au modèle à l'aide de notes UML. Nous pouvons donc ajouter la règle suivante à nos règles de génération de code, qui ne sera exploitable que si le modèle contient des notes de code (ce qui est garanti si le modèle UML est obtenu à partir d'une opération de Reverse Engineering) :

10. Si des notes de code Java sont associées aux opérations des classes UML, ce code est recopié dans les opérations Java correspondantes.

Notons que nos règles de correspondances ne bénéficient pas assez de l'API Java. Il serait possible, par exemple, d'améliorer la règle n° 5, qui porte sur les associations entre classes, en remplaçant l'utilisation du tableau (qui est utilisé pour les associations permettant de relier plusieurs objets) par celle de la classe Java `ArrayList`, qui représente un tableau dynamique.

Ainsi la règle n° 5 deviendrait :

> Si une classe UML est associée à une autre classe UML et que l'association soit navigable, il doit se trouver un attribut dans la classe Java correspondant à la classe UML. Le nom de l'attribut doit correspondre au nom du rôle de l'association. Si l'association spécifie que le nombre maximal d'objets pouvant être reliés est supérieur à 1, le type de l'attribut Java est de type `ArrayList`. Sinon, le type de l'attribut doit être une correspondance Java de la classe UML associée. Si l'association n'est pas navigable, nous considérons qu'il n'est pas possible de générer du code Java.

Notons pour finir que ces règles de correspondances ne prennent pas en compte la sémantique de contenance des associations, Java ne supportant pas un tel concept. Nous considérons donc qu'il est impossible de générer du code Java si le modèle UML contient des associations d'agrégation ou de composition.

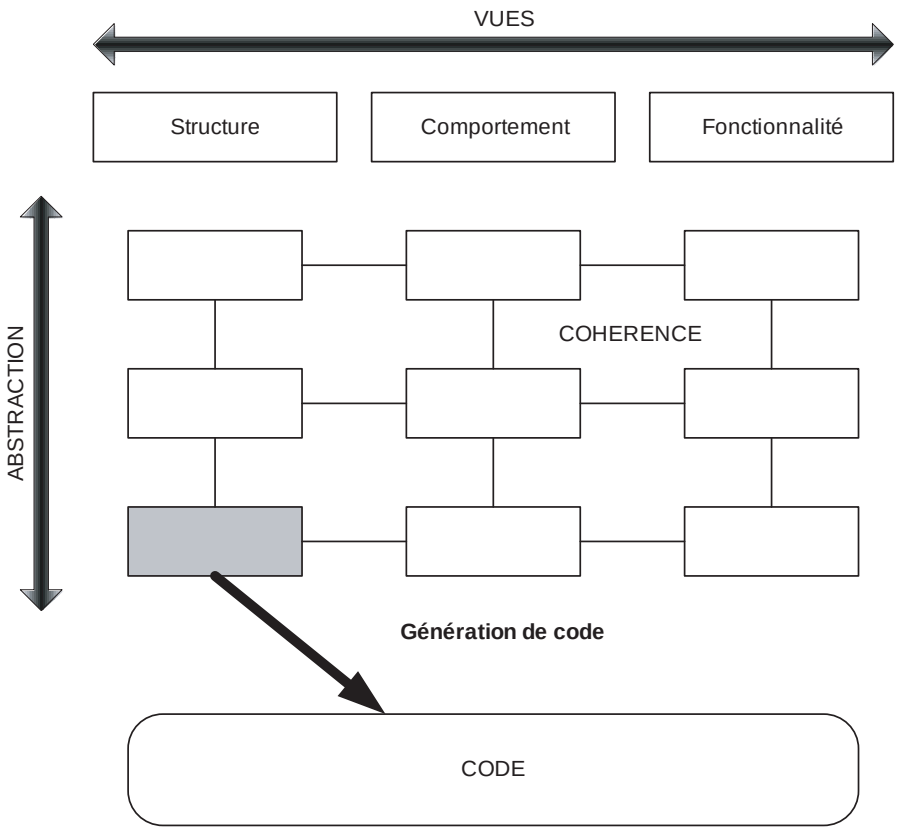
En résumé, les règles de correspondances que nous venons de présenter permettent de décrire brièvement le fonctionnement d'une opération de génération de code Java à partir de modèles UML. Contrairement aux règles de l'opération de Reverse Engineering, ces règles contiennent des contraintes sur la nature des modèles UML à partir desquels peut se faire la génération. Par exemple, il n'est pas possible de générer du code Java si des classes du modèle UML ont des héritages multiples ou si les opérations du modèle UML utilisent les directions `out` ou `inout`.

De plus, l'opération de génération de code Java n'est pleinement exploitable que si elle est exécutée sur un modèle UML qui contient des notes de code associées à ses opérations (un modèle obtenu à partir d'une opération de Reverse Engineering, par exemple). En effet, seule la règle n° 10 permet la génération de code Java exécutable. Lorsque l'opération de génération de code est exécutée sur un modèle qui n'a pas de notes de code associées à ses opérations, le code généré ne contient que des squelettes de code.

Soulignons pour finir que l'opération de génération de code s'applique sur la partie structurelle du modèle UML au plus bas niveau d'abstraction.

La figure 5.1 représente cette opération selon notre représentation du modèle UML.

Figure 5.1
Code, modèle et opération de génération de code



UML vers Java et Java vers UML

Nous venons de voir deux opérations qui permettent respectivement de passer de Java vers UML et de UML vers Java. Il est dès lors nécessaire de savoir si les effets de ces deux opérations s'annulent. En d'autres termes, obtenons-nous le même code après avoir exécuté une opération de Reverse Engineering suivie d'une opération de génération de code ? À l'inverse, obtenons-nous le même modèle après avoir exécuté une opération de génération de code suivie d'une opération de Reverse Engineering ?

En fait, telles que nous les avons définies, l'exécution d'une génération de code suivie d'un Reverse Engineering peuvent retourner un modèle différent du modèle d'entrée, alors que l'exécution d'un Reverse Engineering suivi d'une génération de code retournent toujours le même code.

La raison principale à cela est que l'opération de génération de code fait disparaître les associations entre les classes UML dans le code généré et que celles-ci ne réapparaissent

pas après une opération de génération de code. De plus, la génération de code utilise des classes particulières de l'API Java, qui apparaissent dans le modèle après l'opération de Reverse Engineering. Par contre, tous les concepts Java sont intégrés dans le modèle UML et se retrouvent dans le code après la génération de code.

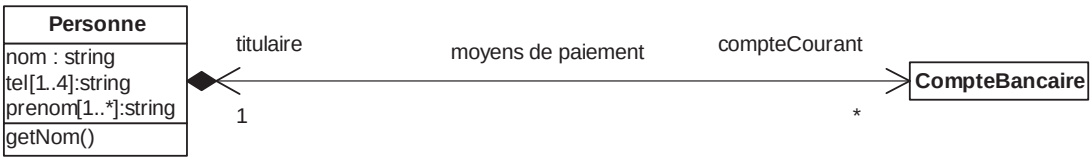


Figure 5.2
Modèle UML avant génération/reverse

Par exemple, en exécutant l'opération de génération de code à partir du modèle illustré à la figure 5.2, nous obtenons le code Java suivant :

```
public class Personne {
    ArrayList compteCourant ;
}
```

```
public class CompteBancaire {
    Personne titulaire ;
}
```

Après l'exécution d'un Reverse Engineering, ce code permet d'obtenir le modèle illustré à la figure 5.3, qui est complètement différent du modèle d'origine.

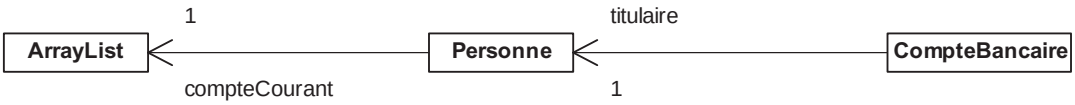


Figure 5.3
Modèle UML après génération/reverse

Les effets engendrés par les opérations de génération de code et de Reverse Engineering sont dictés par les règles de correspondances qui définissent ces opérations. Or, nous avons précisé que ces règles de correspondances n'étaient en aucun cas standards et que chaque outil proposait les siennes. De ce fait, il est impossible de prédire, sans connaître très précisément ces règles, quel sera le résultat obtenu après des exécutions successives d'opérations de génération de code et de Reverse Engineering.

De plus, à l'heure actuelle, aucun outil du marché ne précise pleinement ses règles de correspondances. Nous déconseillons donc, dans le cadre de ce cours, l'exécution successive d'opérations de Reverse Engineering et de génération de code, à moins de savoir exactement quels en seront les effets.

Approches UML et code

Nous venons de voir qu'il n'était actuellement pas raisonnable d'exécuter successivement les opérations de Reverse Engineering et de génération de code. Pour autant, ce sont ces opérations qui permettent une synchronisation entre le code et le modèle.

Rappelons que notre objectif depuis le début de ce cours est d'effectuer des opérations sur les modèles (générer de la documentation, casser les dépendances ou appliquer des patrons de conception) et d'effectuer des opérations sur le code (coder les traitements associés aux opérations, compiler et exécuter).

Il est donc absolument nécessaire de définir une approche permettant de réaliser, d'une part, des opérations sur le code et, d'autre part, des opérations sur le modèle, tout en gardant une synchronisation entre le code et le modèle.

Approches envisageables

Approche Code Driven

Le point de départ de cette approche est le code. L'objectif est de ne jamais utiliser l'opération de génération de code. La cohérence entre le modèle et le code est maintenue grâce à l'opération de Reverse Engineering. L'intérêt de cette approche est limité, car seules les opérations de lecture sur les modèles peuvent être utilisées. Par exemple, il est possible de générer la documentation de l'application, mais il n'est pas possible de casser les dépendances ou d'appliquer des patrons de conception sur les modèles.

Approche Model Driven

Le point de départ de cette approche est le modèle. L'objectif est de ne jamais utiliser l'opération de Reverse Engineering. La cohérence entre le modèle et le code est maintenue grâce à l'opération de génération de code. L'intérêt de cette approche est actuellement limité, car il n'est pas possible de modéliser en UML les traitements associés aux opérations. La génération de code exécutable n'est donc pas possible.

Soulignons cependant qu'il est possible de suivre une approche Model Driven en intégrant directement dans le modèle UML les notes de code Java afin de pouvoir générer un code exécutable. Même si cette approche consiste à intégrer du Java dans le modèle UML, elle reste une approche Model Driven puisque l'opération de Reverse Engineering n'est jamais utilisée. Son inconvénient est de devoir coder du Java dans le modèle UML (les outils UML ne supportent que faiblement cela actuellement).

Approche Round Trip

Le point de départ de cette approche peut être soit le code, soit le modèle. L'objectif est d'utiliser aussi bien les opérations de Reverse Engineering que de génération de code pour assurer la synchronisation entre modèle et code. Cependant, ces opérations doivent être bien préparées et ne doivent pas être utilisées n'importe quand afin de ne pas subir les conséquences des modifications qu'elles réalisent. Cette approche est actuellement la plus intéressante, car elle cumule les avantages offerts par UML et par Java. Cependant,

elle est aussi la plus délicate à mettre en œuvre, car elle nécessite une connaissance très fine des opérations de Reverse Engineering et de génération de code.

Dans le cadre de ce cours, nous utiliserons l'approche Round Trip avec les précautions suivantes :

- Les modèles UML ne doivent pas contenir d'héritages multiples entre classes.
- Les modèles UML ne doivent pas contenir d'associations non navigables.
- Les modèles UML ne doivent pas contenir d'associations d'agrégation ou de composition.
- Les modèles UML ne doivent pas contenir d'associations navigables spécifiant que le nombre maximal d'objets pouvant être reliés est supérieur à 1. À la place, nous ferons en sorte que les modèles UML utilisent la classe `ArrayList`.
- Les modèles UML doivent contenir une note de code attachée à chaque opération.

Ces précautions nous permettront de réaliser successivement les opérations de Reverse Engineering et de génération de code telles que nous les avons définies sans risquer de ne plus avoir de synchronisation entre le modèle et le code.

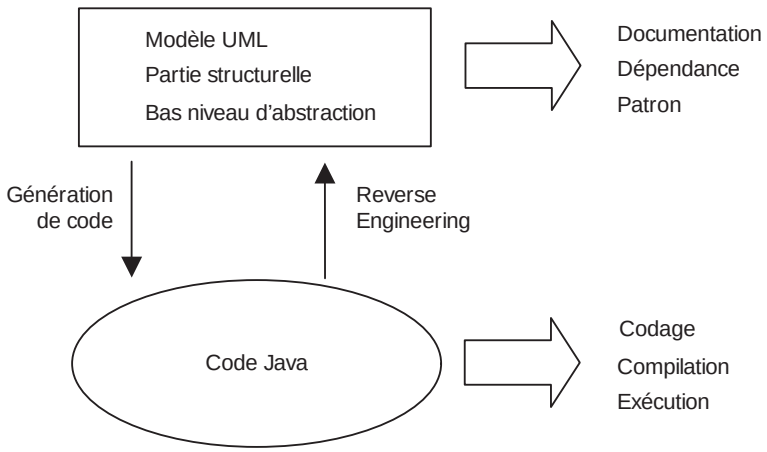
Cycle de développement UML

Grâce aux concepts que nous avons introduits jusqu'à présent dans ce cours, nous pouvons définir un cycle de développement UML, dont les caractéristiques sont les suivantes :

- Il suit une approche Round Trip, car il utilise les opérations de génération de code et de Reverse Engineering pour assurer la synchronisation entre le modèle et le code.
- Il préconise la création de différents diagrammes de classes afin de mieux présenter la structuration de l'application. Grâce aux diagrammes de classes, il est possible de générer automatiquement une documentation de la structuration de l'application à un bas niveau d'abstraction.
- Il préconise la suppression des cycles entre les packages grâce à l'identification des dépendances entre classes et au mécanisme permettant de casser les cycles de dépendances entre classes.
- Il préconise l'application de patrons de conception sur le modèle.
- Il contraint les modifications du modèle UML afin que les exécutions successives de génération de code et de Reverse Engineering ne mettent pas en péril la synchronisation entre modèle et code.
- Il préconise de spécifier les traitements associés aux opérations des classes à l'aide de code Java.
- Il préconise de réaliser la compilation et l'exécution de l'application à l'aide des outils Java classiques.
- Il n'utilise que la partie structurelle au plus bas niveau d'abstraction du modèle de l'application.

La figure 5.4 schématise ce cycle de développement UML.

Figure 5.4
Cycle de développement avec UML



Ce cycle de développement UML cumule donc les avantages de la modélisation et de la programmation, tout en assurant une cohérence globale du modèle et du code. Grâce à la modélisation, il facilite la génération de documentation, l'identification de dépendances, la correction de cycles de dépendances et l'application de patrons de conception. Grâce à la programmation, il facilite le codage des traitements des opérations, la compilation et l'exécution.

Synthèse

Ce chapitre a introduit l'opération de génération de code Java à partir d'un modèle UML. Cette opération est définie à l'aide de règles de correspondances entre les concepts UML et les concepts Java.

Nous avons ensuite souligné que les opérations de génération de code et de Reverse Engineering n'étaient pas symétriques. Les exécutions successives de ces deux opérations peuvent engendrer de fortes modifications dans le modèle et dans le code, qui ne vont pas sans conséquences néfastes sur la synchronisation entre modèle et code.

Nous avons indiqué les différentes approches qui permettent d'utiliser UML dans un cycle de développement. Nous avons expliqué l'approche Round Trip, préconisée dans le cadre de ce cours, qui utilise les opérations de Reverse Engineering et de génération de code sous réserve de respecter certaines contraintes de modélisation afin d'assurer une synchronisation entre modèle et code.

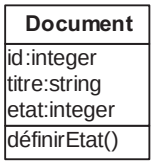
Pour finir, nous avons schématisé le socle fondateur de notre cycle de développement avec UML en précisant ses avantages et les différents points sur lesquels il apporte un gain.

Travaux dirigés

TD5. Génération de code

Question 42 *Écrivez le code généré à partir de la classe* `Document` *illustrée à la figure 5.5.*

Figure 5.5
Classe Document

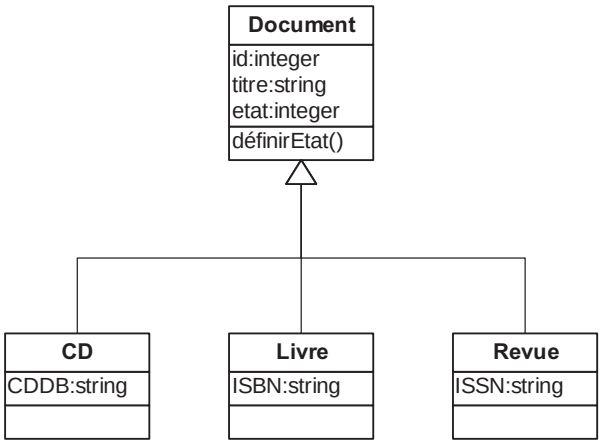


Question 43 *Écrivez le code généré à partir de la classe* `Bibliothèque` *illustrée à la figure 5.6.*

Figure 5.6
Classes Bibliothèque *et* Document

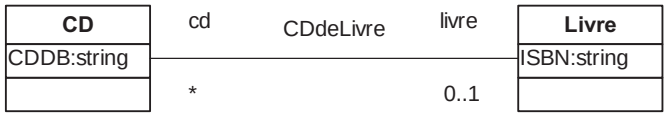
Question 44 *Écrivez le code généré à partir des classes* `Livre`, `CD`, `Revue` *(voir figure 5.7).*

Figure 5.7
Classes CD, Livre *et* Revue



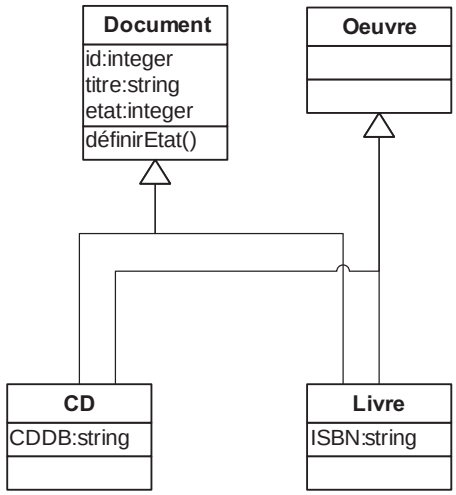
Question 45 *Écrivez le code généré à partir de l'association* `CDdeLivre` *représentée à la figure 5.8 après avoir défini les règles de génération de code que vous comptez utiliser.*

Figure 5.8
Association CDdeLivre



Question 46 *Écrivez le code généré à partir des classes représentées à la figure 5.9 après avoir défini les règles de génération de code que vous comptez utiliser.*

Figure 5.9
Héritage multiple



Un mécanisme update permet de faire remonter les modifications du code Java dans le modèle UML avec lequel il est déjà synchronisé. Par exemple, considérons que le modèle UML et le code Java de la classe `Bibliothèque` *sont synchronisés depuis la question 43 : si nous ajoutons dans le code l'attribut* `nom` *à la classe* `Bibliothèque`*, alors celui-ci apparaîtra dans le modèle UML après exécution de l'update.*

Nous considérons pour l'instant que le mécanisme d'update correspond à une opération de Reverse Engineering du code Java, si ce n'est que les éléments du code qui n'apparaissaient pas dans le modèle y sont directement ajoutés.

Question 47 *Construisez le modèle UML de la classe* `Bibliothèque` *(dont vous avez fourni le code à la question 43) obtenu par update après avoir ajouté dans le code Java les attributs* `nom`*,* `adresse` *et* `type` *dont les types sont des String.*

Question 48 *Nous voulons maintenant, toujours dans le code Java, changer l'attribut* `type` *en attribut* `domaine`*. Pensez-vous qu'il soit possible, après un update,*

que les deux attributs `type` *et* `domaine` *puissent être présents dans le modèle ? Si oui, à quoi est dû ce comportement bizarre ?*

Question 49 *Proposez un nouveau mécanisme d'update ne souffrant pas des défauts présentés à la question 48.*

Question 50 *Proposez le mécanisme inverse de l'update permettant de modifier un modèle UML déjà synchronisé avec du code et de mettre à jour automatiquement le code Java.*

Question 51 *Dans quelle approche de programmation par modélisation (Model Driven, Code Driven et Round Trip) ces mécanismes d'update sont-ils fondamentaux ?*

Ce TD aura atteint son objectif pédagogique si et seulement si :

- Vous comprenez le mécanisme de génération de code présenté.
- Vous avez pris conscience de la complexité et des limites d'une génération de code pour une application réelle.
- Vous avez compris l'importance du mécanisme d'update.