# Diagrammes de séquence de test

### Objectifs

- Présenter les concepts du test
- Sensibiliser à la difficulté de la construction d'une suite de tests
- Présenter l'intérêt des interactions UML pour la spécification des cas de test
- Présenter le cycle de développement avec UML intégrant les tests

## Les tests

Grâce notamment aux techniques dites XP (eXtreme Programming), les tests sont de plus en plus utilisés en développement. L'idée générale est de pouvoir tester le code que nous sommes en train de développer afin de nous assurer que celui-ci est correct, c'est-à-dire qu'il respecte le fameux besoin du client *(voir le chapitre 1)*.

Le concept de test n'est cependant pas si simple, et il est nécessaire de bien avoir en tête certaines définitions avant de voir comment intégrer les tests dans un cycle de développement avec UML.

Avant de présenter les définitions des concepts nécessaires aux tests, il est important de savoir répondre à la question suivante : « À quoi sert le test ? ». Pour l'IEEE, le but du test est de révéler les fautes.

Il s'ensuit les définitions suivantes :

- Une faute intervient quand l'exécution d'un logiciel fournit un résultat autre que celui attendu *(IEEE std 982 1044)*.
- Une faute est causée par une ou plusieurs défaillances dans une implémentation *(IEEE std 982 1044)*.
- Un cas de test est un test d'une propriété particulière d'une application. Il s'agit d'un scénario d'exécution de l'application exhibant une suite de stimuli et comparant les résultats obtenus après stimulation de l'application avec les résultats attendus.
- Une suite de tests est un ensemble de cas de test permettant de valider l'ensemble des propriétés d'une application.

En résumé, nous pouvons dire que les tests sont réalisés dans l'objectif de trouver les défaillances (bogues) d'une application. Pour ce faire, chaque test (cas de test) stimule l'application afin de provoquer une éventuelle faute. Lorsqu'une faute est révélée, cela signifie que l'application contient une ou plusieurs défaillances.

L'idée principale du test fonctionne donc sur l'implication suivante :

*Faute révélée* implique *défaillance(s) dans l'application.*

À partir de cette implication, nous comprenons mieux pourquoi le test a pour objectif de révéler des fautes : c'est en révélant des fautes que nous pouvons affirmer que l'application contient des défaillances.

Soulignons que cette implication n'est pas une équivalence. Ce n'est pas parce qu'aucune faute n'est révélée que nous pouvons affirmer que l'application ne contient pas de défaillances.

Soulignons de surcroît que les tests n'offrent aucun mécanisme pour trouver la raison des défaillances dans le code de l'application et, par voie de conséquence, aucun mécanisme pour les corriger.

Associés à ces concepts relativement théoriques, les concepts suivants ont été définis afin de mettre en pratique l'exécution du test :

- Un cas de test abstrait est un cas de test construit à partir de la spécification de l'application.
- Un cas de test exécutable est un cas de test exécutable sur une architecture d'implémentation cible. Un test exécutable est construit à partir d'un test abstrait.
- Un testeur est une application qui contrôle l'exécution de l'application à tester en lui fournissant les entrées et en comparant les résultats retournés par l'application aux résultats prévus (c'est-à-dire les résultats spécifiés).

Ainsi, afin de réaliser et d'exécuter une suite de tests sur une application, il est nécessaire de :

1. Construire l'ensemble des cas de test abstraits composant la suite de tests. Ces cas de test sont basés sur la spécification de l'application.

2. Construire l'ensemble des cas de test exécutables composant la suite de tests. Ces cas de test sont basés sur les cas de test abstraits et doivent pouvoir s'exécuter sur l'application.
3. Construire un testeur capable d'exécuter la suite de tests sur l'application afin de rendre le verdict (comparaison entre les résultats obtenus et les résultats attendus). Nous verrons dans la suite de ce chapitre que certains environnements Open Source proposent des testeurs.

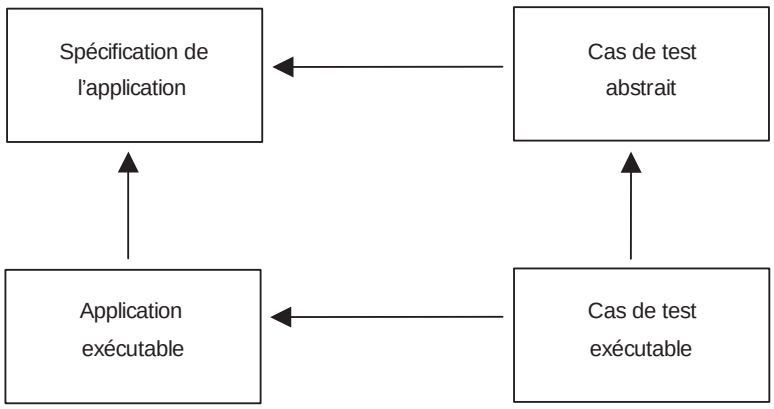
## Comment utiliser les tests ?

Les définitions que nous venons de rappeler impliquent qu'il faut disposer d'une spécification de l'application pour pouvoir construire les cas de test abstraits. Dans notre contexte, nous pouvons considérer que la spécification de l'application est incluse dans le modèle de l'application.

Nous avons aussi montré qu'il fallait disposer d'une application exécutable pour construire des cas de test exécutables. Dans notre contexte, nous pouvons réduire l'application exécutable au code de l'application, car seul le code est nécessaire pour pouvoir exécuter l'application.

Cette distinction entre « spécification de l'application » et « application exécutable » ainsi que les relations qui existent avec les cas de test abstraits et exécutables sont schématisées à la figure 7.1 (les flèches représentent les dépendances entre les éléments).

**Figure 7.1**
*Dépendances entre tests et application*




Cette présentation des relations entre, d'une part, les cas de test abstraits et exécutables et, d'autre part, la spécification de l'application et l'application exécutable met bien en évidence l'importance des cas de test abstraits.

En effet, c'est uniquement de leur qualité que dépend la qualité des résultats obtenus (en terme de fautes révélées, par exemple). Si les cas de test abstraits sont mal construits et stimulent mal l'application, aucune faute ne peut être révélée. Les cas de test exécutables

ne sont qu'un reflet des cas de test abstraits afin de permettre leur exécution sur l'application.

De ce fait, la difficulté la plus importante dans la réalisation d'une suite de tests réside dans la construction des cas de test abstraits. Comment construire de « bons » cas de test abstraits, autrement dit comment construire des cas de test qui permettent de révéler les fautes d'une application ? Ces questions relèvent encore malheureusement du domaine de la recherche.

Nous pourrions ajouter la question suivante : comment faire un ensemble de cas de test abstraits complet, autrement dit comment faire un ensemble de cas de test suffisamment exhaustif pour assurer une certaine qualité d'une application lorsque aucune faute n'est révélée ? Ces questions, tout aussi importantes que les précédentes, sont aussi des questions de recherche.

Nous ne détaillons pas les réponses actuelles à ces questions, qui sortent du contexte de ce cours. Nous pouvons néanmoins prendre conscience de leur complexité grâce à un exemple simple tel que celui d'une application réalisant un tri alphabétique sur les cases d'un tableau de chaînes de caractères.

Une faute possible de cette application serait de mal trier les cases du tableau. Construire un cas de test visant à révéler cette faute est cependant extrêmement complexe. Sur quelles chaînes de caractères faut-il tester l'application ? Sur quelles tailles de tableau faut-il tester l'application ? Quel serait un ensemble de cas de test abstraits suffisamment complet pour pouvoir assurer dans une certaine mesure que l'application réalise correctement le tri de n'importe quel tableau ?

## Écriture de cas de test à partir d'un modèle UML

Le test apporte un gain significatif dans le développement d'applications informatiques. Il est donc absolument nécessaire de l'intégrer à notre cycle de développement avec UML. Nous présentons dans cette section une façon de spécifier les cas de test avec UML.

### *Cas de test abstrait et interaction*

Nous avons vu au chapitre précédent qu'une interaction représentait une succession d'échanges de messages entre plusieurs objets qui peuvent survenir pendant l'exécution d'une application.

Il est possible de vérifier qu'une interaction est réalisée par l'exécution d'une application. Cela signifie que la succession d'échanges de messages spécifiée par l'interaction a été réalisée par l'application lors de son exécution.

Dans le contexte du test, nous pouvons très facilement faire un rapprochement entre les interactions et les cas de test. En effet, il est possible de considérer que la partie du cas de test qui concerne les stimuli envoyés par le testeur vers l'application est un échange de

messages entre le testeur et des objets de l'application. Cette partie du cas de test peut donc être spécifiée grâce à une interaction.

Pour pouvoir spécifier intégralement un cas de test, il faut dès lors être capable d'ajouter aux interactions la spécification des résultats attendus, ainsi qu'une information précisant que l'interaction spécifie un scénario initié par un objet externe à l'application (le testeur).

Nous proposons donc de modifier les classiques interactions UML afin de pouvoir spécifier des cas de test. Nous appellerons *interaction de test* une interaction respectant les contraintes suivantes :

- L'interaction doit obligatoirement contenir un objet représentant le testeur. Cet objet doit être identifié `Testeur` et ne pas avoir de type. L'objet `Testeur` ne doit pas être créé ni supprimé par un objet de l'interaction. L'objet `Testeur` ne doit pas non plus recevoir de message d'appel d'opération.
- L'interaction doit obligatoirement contenir d'autres objets. Tous les autres objets doivent être identifiés et typés. Tous ces objets doivent être créés par l'objet `Testeur`.
- L'interaction peut contenir des messages d'appel d'opération synchrone ou asynchrone, mais seul l'objet `Testeur` peut être l'objet qui envoie ces messages.
- L'interaction doit contenir une note contenant le résultat attendu.

Nous appellerons diagramme de séquence de test le diagramme de séquence représentant graphiquement une interaction de test. Ce diagramme doit respecter les contraintes suivantes :

- L'objet `Testeur` doit être l'objet le plus à gauche du diagramme.
- La note contenant le résultat attendu doit apparaître sur le diagramme, de préférence en bas, après le dernier message.

La figure 7.2 présente un diagramme de séquence de test représentant un cas de test abstrait sur l'algorithme de tri que nous avons présenté précédemment. Ce diagramme respecte toutes les contraintes que nous avons définies. Soulignons que ce cas de test abstrait ne permet que de s'assurer que l'algorithme ne retourne pas d'erreur de tri si nous lui demandons de trier le tableau « b », « a », « c ». Ce cas de test ne donne donc aucune garantie quant au résultat du tri sur tout autre jeu de données.

Soulignons que les interactions de test permettent de spécifier les cas de test abstraits et non les cas de test concrets. En effet les interactions sont cohérentes avec les classes UML qui ne sont pas exécutables car elles ne contiennent pas les traitements associés à leurs opérations autrement que sous forme de note de code écrite dans des langages de programmation tels que Java.

### Cas de test exécutables et interactions

Nous avons déjà précisé qu'un cas de test exécutable était le reflet d'un cas de test abstrait afin de permettre son exécution sur l'application. Dit autrement, un cas de test exécutable est la traduction d'un cas de test abstrait dans un langage de programmation particulier.

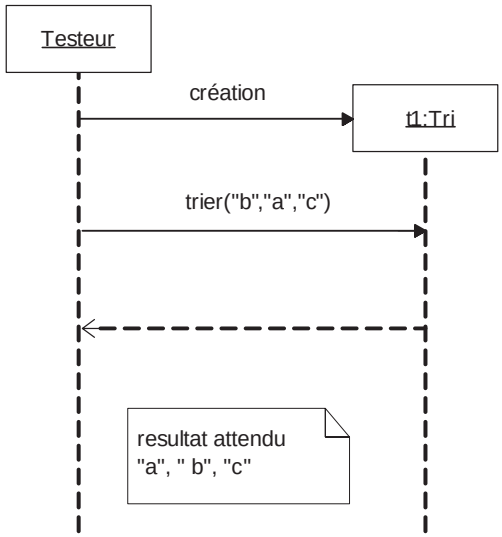



**Figure 7.2**
*Cas de test abstrait en UML*

Étant donné, d'une part, que nous spécifions les cas de test abstraits à l'aide d'interactions cohérentes avec la partie structurelle du modèle et que, d'autre part, nous générons le code de l'application grâce à l'opération de génération de code à partir de cette partie structurelle, il semble naturel de vouloir générer les cas de test exécutables à l'aide d'une opération de génération de code de test similaire à l'opération de génération de code que nous avons présentée au chapitre 5.

Nous proposons donc de définir une opération de génération de code de test à partir d'une interaction. Ce code de test devant être exécuté par un testeur, nous faisons le choix de ne pas développer nous-même ce testeur mais de réutiliser le framework de test JUnit *(http://www.junit.org/index.htm)*. Celui-ci propose une API Java permettant de coder des cas de test et de les exécuter sur une application Java.

Notre opération de génération de code de test à partir d'une interaction s'appuie donc sur le framework JUnit et est spécifiée avec les règles de correspondance suivantes :

1. À toute interaction doit correspondre un cas de test JUnit. Cela signifie qu'une nouvelle classe Java doit être construite. Cette classe doit hériter de la classe JUnit `TestCase`. La classe doit contenir une méthode correspondant au test. Cette méthode aura pour nom `testExecutable`.
2. Dans la méthode `testExecutable` de la classe correspondant au cas de test, il doit correspondre un appel de méthode Java pour chaque message de l'interaction partant de l'objet `Testeur`.
   – Si le message est une création, l'appel de méthode Java doit être une création d'objet (`new`).
   – Si le message est une suppression d'objet, nous considérons que la génération s'arrête en soulevant une erreur, car Java ne supporte pas les suppressions d'objets.

– Si le message est un appel synchrone d'opération, l'appel de méthode Java doit être un appel vers la méthode Java correspondant à l'opération.
– Si le message est un appel asynchrone d'opération, nous considérons que la génération s'arrête en soulevant une erreur, car Java ne supporte pas nativement les appels asynchrones.

3. Dans la méthode `testExecutable` de la classe correspondant au cas de test, il doit correspondre une assertion JUnit correspondant au résultat attendu spécifié dans l'interaction. Pour pouvoir automatiser l'écriture de cette assertion JUnit, il faut proposer un formalisme de spécification du résultat attendu dans l'interaction UML (notre exemple ne fait que spécifier le résultat attendu en langage naturel). UML ne standardise pas un tel formalisme, mais la plupart des outils UML proposent chacun leur propre formalisme de spécification.

En appliquant ces règles de génération de code de test sur le cas de test abstrait que nous avons spécifié à la section précédente à l'aide d'une interaction, nous obtenons le cas de test exécutable suivant :

```
public class TriInteraction extends TestCase {
    public void testExecutable() {
        t1 = new Tri() ;
        String[] resultatObtenu = t1.trier("b" , "a", "c") ;
        assertEquals(resultatObtenu[0] , "a") ;
assertEquals(resultatObtenu[1] , "b") ;
assertEquals(resultatObtenu[2] , "c") ;
    }
}
```

Grâce au framework JUnit, ce cas de test exécutable peut être exécuté sur l'application qui aura été obtenue à l'aide d'une génération de code.

## Synthèse

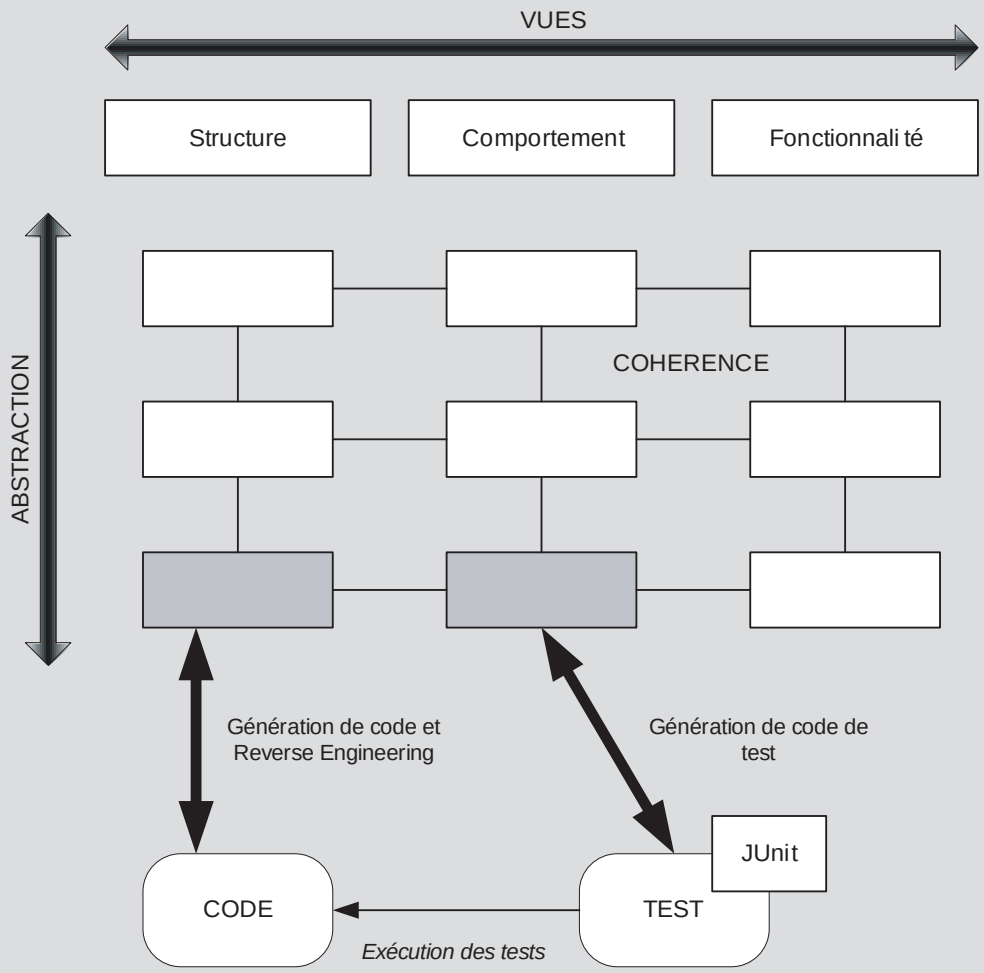



**Figure 7.3**

*Le test dans notre cycle de développement avec UML*

Nous avons introduit dans ce chapitre les concepts de base du test. Nous avons en particulier insisté sur le principe fondamental du test, qui est de révéler les fautes de l'application afin de permettre l'identification des défaillances. Il est important de souligner le fait que, si aucune faute n'est révélée, cela ne donne aucune garantie sur la non-existence de défaillances. Nous avons simplement identifié un ensemble de « données » ne provoquant pas de défaillance et n'avons aucune information sur ce qui se passe avec d'autres données.

Nous avons ensuite présenté les concepts permettant la réalisation du test. Nous avons notamment détaillé les relations entre la spécification de l'application, le code de l'application, les cas de test abstraits et les cas de test exécutables.

Nous avons en outre montré en quoi les interactions UML et les diagrammes de séquence pouvaient être utilisés pour spécifier les cas de test abstraits. Pour ce faire, nous avons proposé une façon d'utiliser les interactions afin de permettre une spécification complète des cas de test.

Pour finir, nous avons présenté une opération de génération de code de test à partir des interactions de test.

Nous pouvons dès lors intégrer tous ces mécanismes dans notre cycle de développement avec UML.

La figure 7.3 schématise cette intégration en mettant bien en évidence le fait que les parties structurelles et comportementales de bas niveau d'abstraction sont exploitées et offrent des gains de productivité pour le développement de l'application. En effet, si des fautes sont révélées après l'exécution des tests sur l'application, l'identification et la correction des défaillances peuvent se faire soit sur le code, soit sur le modèle.

Travaux dirigés

# TD7. Diagrammes de séquence de test

La classe `Partie` de l'application de gestion de championnat d'échecs présentée au TD6 représente une partie d'échecs. Elle permet aux joueurs de jouer leur partie en appelant l'opération `jouerCoup()`. Chaque fois qu'un coup est joué, l'opération `vérifierMat()` est appelée afin de vérifier que la position n'est pas mat. Si tel est le cas, la partie est finie. Aucun coup ne peut alors être joué (voir TD6 pour la modélisation de classe `Partie` ainsi qu'un diagramme de séquence spécifiant un cas nominal de déroulement d'une partie entre deux joueurs).

**Question 60.** *Identifiez une faute qui pourrait intervenir lors du déroulement d'une partie.*

**Question 61.** *Définissez un cas de test abstrait visant à révéler cette faute.*

**Question 62.** *Construisez un diagramme de séquence de test modélisant le cas de test abstrait de la question précédente.*

**Question 63.** *Écrivez le pseudo-code Java du cas de test exécutable correspondant au cas de test abstrait de la question précédente.*

**Question 64.** *Si ce cas de test ne révèle pas de faute, est-ce que cela signifie que l'application ne contient pas de défaillance ?*

**Question 65.** *Combien de cas de test faudrait-il élaborer pour améliorer la qualité de l'application ?*

*L'application permettant la gestion de championnat d'échecs contient aussi la classe* `ChampionnatDEchecs`*, qui est associée à la classe* `Partie` *et qui permet de gérer l'inscription des joueurs et la création des parties (voir TD6).*

**Question 66.** *Identifiez une faute qui pourrait intervenir lors de la création des parties d'un championnat. Définissez un cas de test abstrait visant à révéler cette faute, et construisez un diagramme de séquence de test modélisant ce cas de test abstrait.*

**Question 67.** *Est-il possible de lier les deux cas de test abstrait que vous avez définis (un à la question 61, l'autre à la question 66) ?*

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

- Vous savez identifier des fautes possibles.
- Vous savez définir les cas de test permettant de révéler les fautes.
- Vous avez conscience de la complexité de définir un jeu de tests complet.

# 8

# Plates-formes d'exécution

**Objectifs**

- Définir la notion de plate-forme d'exécution
- Présenter la façon dont UML prend en charge les plates-formes
- Préciser comment et pourquoi s'abstraire des plates-formes d'exécution

## Java dans UML

Depuis le début de ce cours, tous les modèles UML que nous avons présentés étaient plus ou moins liés à Java. Il est cependant essentiel de différencier, dans un modèle UML, les éléments qui dépendent de Java et les autres.

Nous introduisons dans cette section les concepts de modèle UML pour Java et de modèle 100 % UML afin d'expliciter cette différence.

### *Modèles 100 % UML et modèles UML pour Java*

Lorsque nous avons présenté l'opération de Reverse Engineering, nous avons expliqué, d'une part, que les classes de l'API Java étaient introduites dans le modèle et, d'autre part, que le code Java était intégré au modèle dans des notes attachées aux opérations des classes.

Lorsque nous avons présenté l'opération de génération de code, nous avons précisé des contraintes sur les modèles. Ces contraintes ont été définies afin d'assurer la génération

du code Java. Elles dépendent donc de Java. Par exemple, nous avons précisé qu'il ne fallait pas que le modèle UML utilise l'héritage multiple, car celui-ci n'était pas supporté dans Java.

Lorsque nous avons présenté les interactions, nous avons précisé des règles assurant leur cohérence avec la partie structurelle du modèle. De ce fait, elles dépendent aussi de Java. Il est notamment envisageable que des objets participant aux interactions soient des objets instances des classes de l'API Java.

Pour finir, lorsque nous avons présenté la génération du code de test, nous avons utilisé la plate-forme JUnit, qui est une plate-forme Java. Les interactions de tests sont donc elles aussi fortement dépendantes de Java.

Pour résumer, les modèles que nous avons réalisés depuis le début de ce cours sont des modèles UML qui dépendent de Java. Nous les appelons « modèles UML pour Java ». À l'inverse, nous appelons « modèles 100 % UML » les modèles indépendants de tout langage de programmation.

### *UML productif ou pérenne*

S'il existe deux sortes de modèles UML (modèles 100 % UML et modèles UML pour Java), il est important de bien comprendre ce qui les différencie et d'identifier les gains que nous pouvons obtenir de chacun d'entre eux.

Nous connaissons très bien les modèles UML pour Java, car ce sont les modèles que nous avons utilisés depuis le début de ce cours. Comme indiqué à la section précédente, la particularité d'un modèle UML pour Java est de dépendre du langage de programmation Java. Cette particularité est autant un avantage qu'un inconvénient.

L'avantage est que, grâce à cette dépendance, les opérations de génération de code et de Reverse Engineering peuvent être réalisées conjointement, garantissant ainsi une synchronisation entre le modèle et le code d'une application. Grâce à cette synchronisation, il est possible d'obtenir à la fois les avantages des opérations réalisables sur les modèles (recherche et modification des dépendances, génération de documentation, spécification et génération des cas de test en cohérence avec l'application) et les avantages des opérations réalisables sur le code (codage, compilation et exécution).

Les modèles UML pour Java offrent donc des gains de productivité vers le langage Java.

Cette caractéristique des modèles UML pour Java est aussi un inconvénient, en ce qu'elle restreint les gains potentiels offerts par UML uniquement à des gains de productivité vers le langage Java. Pour autant, UML a été défini historiquement afin de faciliter la compréhension et la conception d'applications orientées objet. La majorité des concepts UML ont été définis afin de mieux appréhender la complexité de la construction de ces applications. Le concept d'association, par exemple, est très intéressant pour spécifier les liens structurels existant entre les classes. Ce concept permet typiquement de gérer la complexité des applications mais n'offre pas de gains de productivité pour générer le code de l'application.

Les modèles UML pour Java n'offrent donc que peu de gains pour gérer la complexité des applications orientées objet.

Comme leur nom l'indique, les modèles 100 % UML sont quant à eux complètement indépendants des plates-formes d'exécution. Leurs avantages et inconvénients sont donc essentiellement inverses à ceux des modèles UML pour Java.

**Plate-forme d'exécution**

La notion de « plate-forme d'exécution » englobe à la fois les langages de programmation (Java, C++, C#, etc.) et les serveurs d'applications (J2EE, PHP, EJB, .Net, etc.).

Le fait que les modèles soient indépendants des plates-formes d'exécution fait que les informations qu'ils contiennent sont, par définition, indépendantes des changements internes des plates-formes.

La plate-forme Java, par exemple, a déjà changé cinq fois de version en dix ans, avec des modifications assez importantes de l'API, qui nécessitent une modification des modèles UML pour Java déjà réalisés si l'application modélisée doit pourvoir s'exécuter sur la nouvelle plate-forme. Faire un modèle 100 % UML permet de s'affranchir de ces modifications et donc de rendre l'information beaucoup plus pérenne.

Faire un modèle 100 % UML permet en outre, et surtout, de faire des choix de conception indépendamment des plates-formes d'exécution. Il est ainsi possible de commencer un développement sans avoir, au préalable, choisi la plate-forme d'exécution.

Pour finir, soulignons que la pérennité de l'information contenue dans un modèle 100 % UML est plus facile à atteindre grâce à l'emploi de tous les concepts UML construits spécialement pour cela (association, objet non typé, etc.). Rappelons que l'objectif premier d'UML était d'être un langage de modélisation, et non un langage de programmation.

L'inconvénient des modèles 100 % UML est toutefois de n'avoir aucun lien avec les plates-formes d'exécution. De ce fait, il est très difficile de générer du code exécutable à partir d'un modèle 100 % UML. Si nous prenons l'exemple de la plate-forme Java, que nous connaissons déjà, il est très difficile de générer du code à partir d'un modèle 100 % UML utilisant l'héritage multiple.

Pour résumer, les modèles UML pour Java et les modèles 100 % UML sont complémentaires. Les modèles UML pour Java offrent des gains de productivité, tandis que les modèles 100 % UML offrent des gains de pérennité et facilitent la gestion de la complexité de la construction des applications orientées objet. Il est donc intéressant de ne pas les considérer comme des modèles indépendants, mais plutôt comme des modèles complémentaires.

## Niveaux conceptuel et physique

Nous venons de voir que les modèles 100 % UML et les modèles UML pour Java étaient complémentaires. En fait, les modèles 100 % UML sont des abstractions des modèles UML pour Java, car ils masquent toutes les informations relatives à la plate-forme Java. Nous pourrions dire en outre que les modèles UML pour Java précisent les modèles 100 % UML en expliquant comment utiliser la plate-forme Java pour mettre en œuvre la conception exprimée dans le modèle 100 % UML.

### *Abstraction de la plate-forme*

Comme les modèles 100 % UML sont des abstractions des modèles UML pour Java, il est possible de définir une opération permettant de construire un modèle 100 % UML à partir d'un modèle UML pour Java. Il suffit pour cela de supprimer toutes les informations relatives à la plate-forme Java.

Les règles à appliquer pour construire un modèle 100 % UML à partir d'un modèle UML pour Java sont les suivantes :

1. Supprimer toutes les associations entre les classes qui n'appartiennent pas à l'API Java et les classes qui appartiennent à l'API Java.
2. Supprimer toutes les classes de l'API Java.
3. Si une propriété d'une classe qui n'appartient pas à l'API Java a un type Java, remplacer ce type par un type UML correspondant (il peut être nécessaire de définir la classe représentant ce type s'il ne s'agit pas d'un type UML de base).
4. Si une opération d'une classe qui n'appartient pas à l'API Java a un paramètre qui a un type Java, remplacer ce type par un type UML correspondant (même remarque que pour les types des propriétés).
5. Dans les interactions, supprimer les objets instances des classes de l'API Java.

Ces règles draconiennes permettent d'obtenir un modèle 100 % UML à partir d'un modèle UML pour Java. Cependant, il est important de noter que le modèle 100 % UML obtenu n'est pas réellement exploitable, car il ne contient que des informations partielles et incomplètes. Il faut donc le compléter en ajoutant des associations entres les classes ou en complétant les interactions afin de bien spécifier les informations principales de l'application, aussi appelées informations métier.

Les informations relatives aux parties graphiques de l'application, qui sont obligatoirement dépendantes de la plate-forme d'exécution, sont, par exemple, entièrement retirées du modèle 100 % UML. Le modèle est dès lors indépendant des changements internes de la plate-forme Java et peut être réutilisé pour d'autres réalisations (vers la plate-forme .Net, par exemple).

À l'inverse, il est très délicat de construire automatiquement un modèle UML pour Java à partir d'un modèle 100 % UML. En effet, si le modèle 100 % UML utilise des constructions non supportées par la plate-forme Java, il faut traduire ces constructions afin de

les représenter dans le modèle UML pour Java. Par exemple, si le modèle 100 % UML utilise l'héritage multiple, il faut transformer tout héritage multiple en un ensemble d'héritages simple, ce qui est possible mais reste une opération très délicate et très complexe.

De plus, construire automatiquement un modèle UML pour Java à partir d'un modèle 100 % UML ne peut se faire que si nous connaissons la signification des classes du modèle 100 % UML afin de bien identifier la partie de l'API Java à utiliser. Par exemple, si nous savons que le modèle 100 % UML définit une classe responsable de la sauvegarde sur disque, nous pouvons utiliser l'API Java dédiée aux entrées-sorties **(java.io.File).** Malheureusement, il n'est pas possible de préciser ce niveau de détail des classes en UML.

Pour toutes ces raisons, plutôt que de générer automatiquement des modèles UML pour Java à partir de modèles 100 % UML ou l'inverse, nous préconisons plutôt de spécifier la logique métier de l'application à l'aide d'un modèle 100 % UML et de spécifier la réalisation de cette logique métier sur une plate-forme d'exécution particulière (pour nous Java) à l'aide d'un modèle UML pour Java.

Définir la logique métier d'une application consiste, d'une part, à définir les informations principales manipulées par l'application (objets métier) et, d'autre part, à définir les fonctions principales de l'application (fonctions métier) en précisant leurs impacts en terme de modification sur les données.

L'objectif d'un modèle 100 % UML est donc de représenter la logique métier de l'application et non d'expliquer comment cela fonctionne dans Java. Tous les objets et fonctions d'un modèle 100 % UML ne se retrouvent donc pas obligatoirement tels quels dans le modèle UML pour Java correspondant.

## *Niveaux d'abstraction*

Nous venons de voir que les modèles 100 % UML et UML pour Java étaient complémentaires, que les uns étaient des abstractions des autres, et qu'il n'était pas envisageable de passer des uns aux autres à l'aide d'opérations automatiques.

De ce fait, il parait naturel de ne pas traiter ces modèles comme des modèles indépendants, mais plutôt comme des parties d'un même modèle situées à différents niveaux d'abstraction.

Nous considérons les deux niveaux d'abstraction suivants :

- **Niveau conceptuel.** Correspond au niveau 100 % UML. Ce niveau contient la logique métier de l'application, celle-ci étant spécifiée d'une façon indépendante de toute plate-forme d'exécution.
- **Niveau physique.** Correspond au niveau UML pour Java. Ce niveau contient la réalisation de la logique métier sur une plate-forme d'exécution particulière, les informations contenues à ce niveau étant dépendantes de la plate-forme d'exécution (Java dans notre cas).

Ces deux niveaux sont liés, car le niveau conceptuel est une abstraction du niveau physique. Plus précisément, tous les éléments du niveau conceptuel (classes, associations, interactions) doivent avoir au moins un élément correspondant dans le niveau physique. Par contre, tous les éléments du niveau physique ne sont pas forcément des concrétisations d'éléments du niveau conceptuel, certains ayant pu être ajoutés pour obtenir un modèle à partir duquel il est possible de produire du code Java.

Les liens entre ces niveaux ne sont pas obtenus par une opération de génération automatique. Ils doivent être précisés par le concepteur du modèle lors de la conception du modèle. Cette tâche peut être ardue. Elle n'en est pas moins indispensable, car ces liens garantissent la cohérence des informations situées aux deux niveaux d'abstraction.

## Cycle de développement UML

Notre cycle de développement avec UML intègre maintenant deux niveaux d'abstraction. Le niveau physique est en cohérence avec le code grâce aux opérations de génération de code et de Reverse Engineering. Le niveau conceptuel est en cohérence avec le niveau physique grâce aux relations d'abstraction spécifiées par le concepteur du modèle. Dans cette section nous expliquons comment intégrer ces deux niveaux conceptuels dans notre cycle de développement UML.

### *Intégration des deux niveaux dans le cycle*

Nous venons de voir que les modèles 100 % UML et les modèles UML pour Java étaient deux parties d'un même modèle, situées à deux niveaux d'abstraction différents.

Ces niveaux d'abstraction correspondent aux deux niveaux d'abstraction les plus bas de notre vision schématique du modèle UML d'une application. La figure 8.1 illustre ces deux niveaux.

Nous avons volontairement fait apparaître un lien de cohérence entre les parties structurelles des niveaux conceptuel et physique. Ce lien peut être spécifié en UML à l'aide d'une relation d'abstraction entre les classes situées dans ces deux niveaux.

Cette relation d'abstraction peut apparaître graphiquement sur un diagramme de classes à l'aide d'une flèche pointillée. Par contre, nous n'avons pas fait apparaître de lien de cohérence entre les parties comportementales des deux niveaux. Cela s'explique par le fait qu'UML ne propose pas de concept permettant de représenter graphiquement une relation d'abstraction entre deux interactions.

### *Approches possibles*

Soulignons que les objectifs des niveaux physique et conceptuel sont complémentaires. L'intérêt du niveau conceptuel par rapport au niveau physique est de pérenniser les informations les plus importantes du modèle (les informations métier). De plus, en faisant abstraction des spécificités des plates-formes d'exécution, la construction du niveau conceptuel permet de mieux appréhender la complexité de la construction des applica-

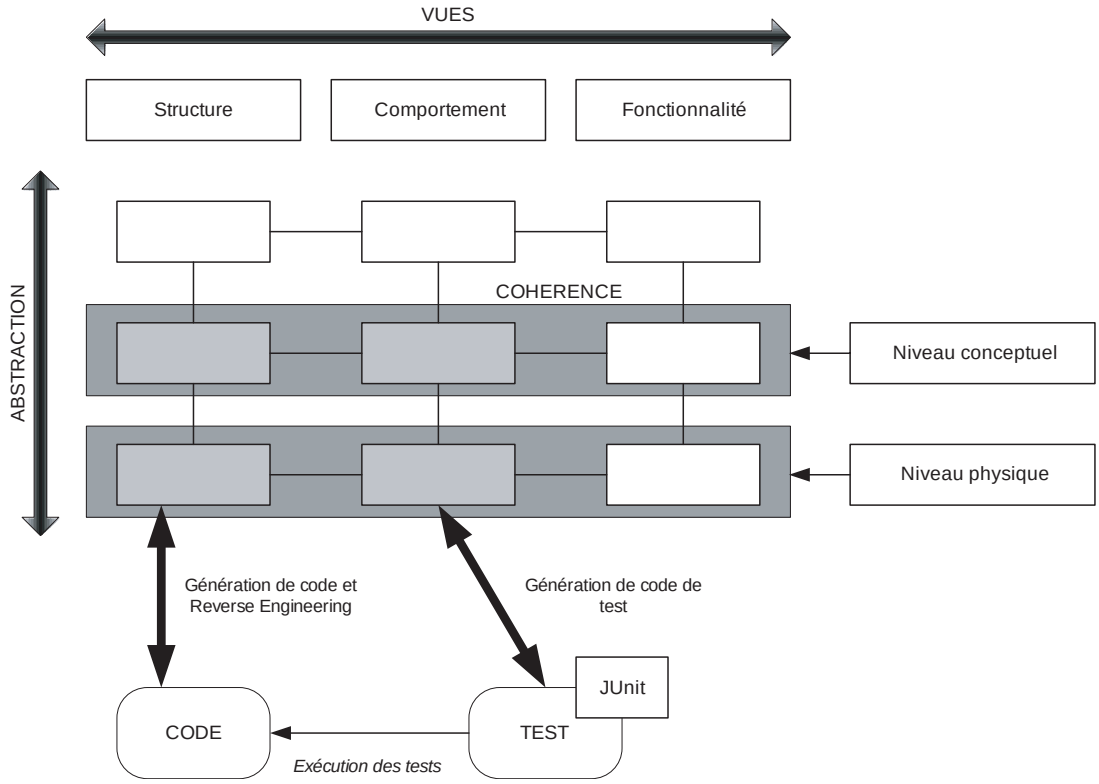



**Figure 8.1**

*Niveaux d'abstraction « physique » et « conceptuel » dans le modèle UML*

tions orientées objet. Contrairement au niveau physique, le niveau conceptuel n'offre donc pas de gain de productivité réellement quantifiable.

C'est pourquoi nous ne préconisons l'élaboration du niveau conceptuel que s'il y a un réel intérêt soit à pérenniser les informations métier (si un changement de plate-forme d'exécution est envisagé), soit à gérer la complexité de l'application en faisant abstraction des détails techniques.

Il est vrai que ces intérêts se retrouvent principalement dans le contexte de la construction d'applications relativement complexes, dont la taille dépasse dix mille lignes de code. Pour les autres applications, dont la taille est inférieure à dix mille lignes de code, l'intérêt de disposer d'un niveau abstrait disparaît devant la difficulté à mettre en œuvre les relations de cohérence avec le niveau physique.

Les deux approches envisageables pour suivre un cycle de développement avec UML sont donc soit de mettre en œuvre le niveau conceptuel, soit de s'en passer. Il serait contre-productif de vouloir mettre absolument en œuvre le niveau conceptuel si aucun bénéfice ne pouvait en être retiré.

## Synthèse

Nous avons vu dans ce chapitre que les modèles que nous avons présentés jusqu'ici étaient fortement dépendants de la plate-forme Java. Forts de ce constat, nous avons introduit la notion de modèle pour Java et de modèle 100 % UML.

Nous avons montré que les modèles pour Java et les modèles 100 % UML étaient complémentaires et qu'ils offraient des gains différents. Les modèles pour Java apportent des gains de productivité, tandis que les modèles 100 % UML apportent des gains de pérennité et de gestion de la complexité.

Nous avons ensuite précisé la relation d'abstraction qui existait entre ces deux niveaux. Les modèles 100 % UML sont des abstractions des modèles UML pour Java, car ils masquent les détails techniques de la plate-forme Java. Ainsi, les modèles 100 % UML spécifient la logique métier de l'application, alors que les modèles UML pour Java spécifient la façon dont est utilisée la plate-forme Java pour réaliser l'application.

Nous avons en outre expliqué pourquoi il était plus naturel de considérer ces deux types de modèles non pas comme des modèles indépendants, mais comme des parties différentes d'un même modèle, situées à différents niveaux d'abstraction. Nous avons alors précisé comment ces deux niveaux d'abstraction (conceptuel et physique) étaient intégrés à notre vision schématique des modèles UML d'applications.

Pour finir, nous avons introduit les deux façons envisageables de suivre un cycle de développement avec UML, en précisant qu'il n'était pas obligatoire de mettre en œuvre le niveau conceptuel; tout dépendant des bénéfices que nous souhaitions obtenir du modèle UML.

Travaux dirigés

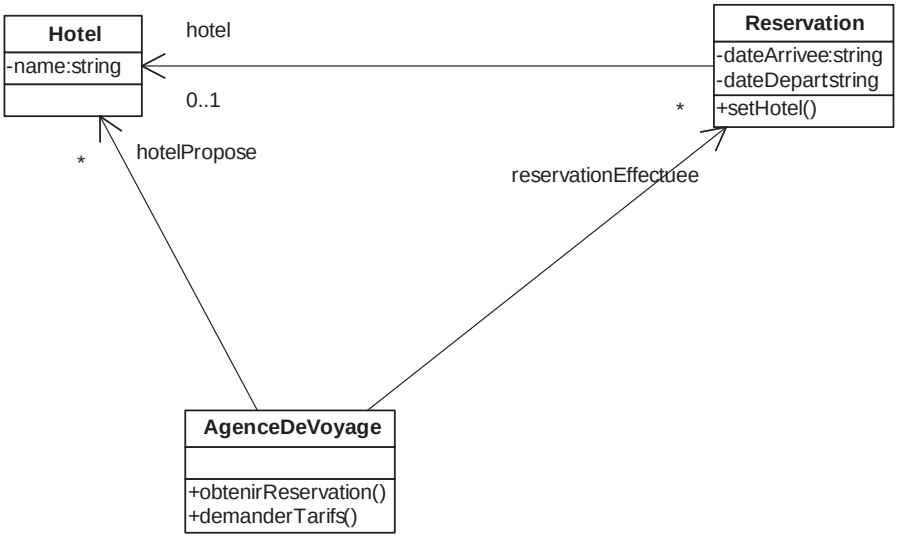
# TD8. Plates-formes d'exécution

**Question 68.** *Le diagramme de l'agence de voyage représenté à la figure 8.2 correspond-il à un modèle conceptuel ou à un modèle physique ?*

**Question 69.** *Pensez-vous qu'il soit intéressant d'appliquer des patrons de conception sur les modèles conceptuels ?*

**Question 70.** *Le diagramme de séquence représenté à la figure 8.3 est-il conceptuel ou physique ? Vous noterez qu'il fait intervenir une opération qui n'apparaît pas dans le diagramme de classes initial. Quelle classe doit posséder cette opération ?*

**Figure 8.2**
*Diagramme de l'agence de voyage*



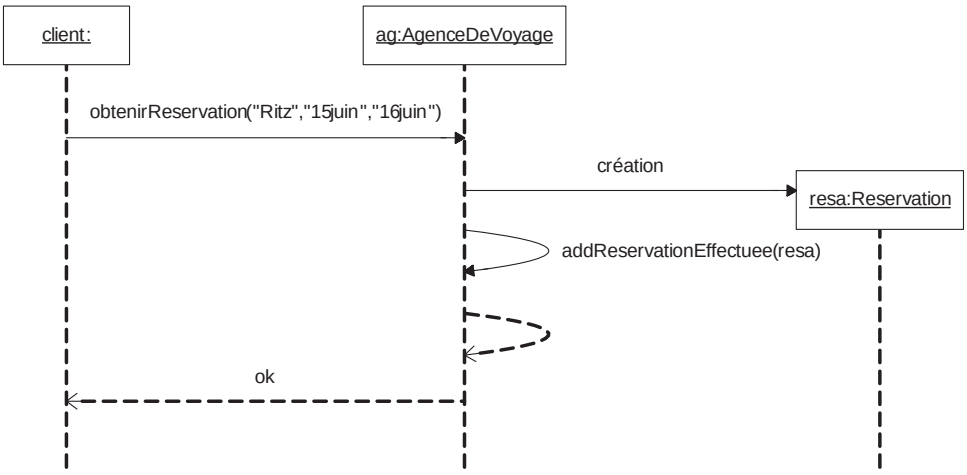




**Figure 8.3**
*Interaction représentant une réservation*

**Question 71.** *Serait-il possible de spécifier en « 100 % UML » le comportement de l'agence de voyage ?*

**Question 72.** *Serait-il possible de spécifier en « 100 % UML » des tests pour l'agence de voyage ? Justifiez l'intérêt de ces tests.*

**Question 73.** *Le diagramme représenté à la figure 8.4 est une concrétisation du diagramme conceptuel de l'agence de voyage. Exprimez les relations d'abstraction entre les éléments des deux diagrammes.*

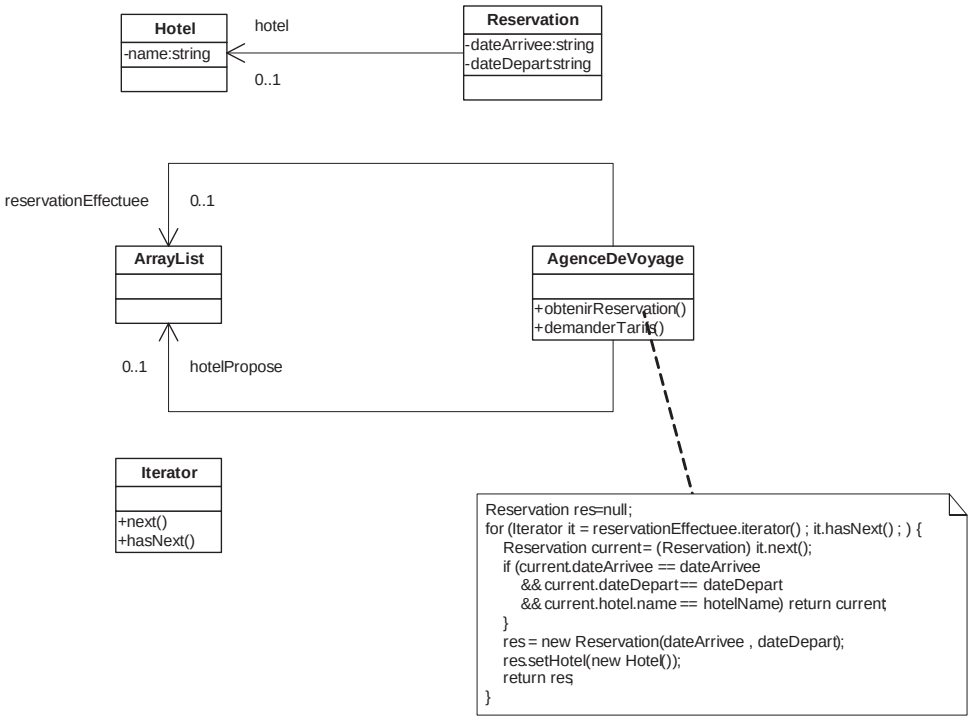



**Figure 8.4**

*Classes du niveau physique de l'agence de voyage*

**Question 74.** *Quel est l'intérêt d'avoir fait apparaître les classes* `ArrayList` *et* `Iterator` *dans le modèle concret (considérez en particulier la génération de code et le Reverse Engineering) ?*

**Question 75.** *Construisez le diagramme de séquence concrétisant le diagramme de séquence présenté à la question 69.*

**Question 76.** *Exprimez les relations d'abstraction entre les diagrammes de séquence.*

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

- Vous savez différencier des modèles conceptuels et des modèles physiques.
- Vous savez établir des relations d'abstraction entre modèles de niveau différent.
- Vous avez conscience que plus le modèle physique est proche de la plate-forme d'exécution, plus il est loin du modèle conceptuel (et inversement).