Diagrammes de séquence

Objectifs

- Présenter les concepts UML relatifs à la vue comportementale (diagramme de séquence)
- Présenter la notation graphique du diagramme de séquence UML
- Expliquer la sémantique des séquences UML en précisant le lien avec les classes UML

Vue comportementale du modèle UML

Depuis le début de ce cours, nous n'avons présenté que les concepts relatifs à la vue structurelle des applications. Dans le paradigme orienté objet, la structure d'une application est entièrement définie par ses classes et leurs relations. La vue structurelle est donc complètement couverte par les concepts UML relatifs aux classes (classe, opération, propriété, association, etc.). Rappelons que le diagramme de classes est la représentation graphique de cette vue.

L'aspect comportemental d'une application orientée objet est défini par la façon dont interagissent les objets qui composent l'application. À l'exécution, l'objet est l'entité de base d'une application. Les objets qui composent une application pendant son exécution et leurs échanges de messages permettent à l'application de réaliser les traitements pour lesquelles elle a été développée.

UML propose plusieurs vues permettant de définir les interactions entre objets. Une de ces vues permet de présenter des exemples d'interaction entre plusieurs objets. Grâce à

ces exemples d'interactions, il est possible de mieux comprendre le comportement de l'application ou de vérifier que l'exemple d'interaction se déroule convenablement.

Cette vue, dont la représentation graphique est le diagramme de séquence, définit deux concepts principaux : celui d'objet et celui de message échangé entre deux objets. Une interaction permet d'identifier plusieurs objets et de représenter les messages qu'ils s'échangent.

Concepts élémentaires

Cette section présente les concepts élémentaires de la vue comportementale d'un modèle UML. Dans notre contexte, ces concepts sont suffisants pour exprimer des exemples d'exécution d'une application.

Objet

Sémantique

Dans une application, chaque objet peut envoyer et recevoir des messages des autres objets qui composent l'application. En UML, les objets qui participent à une interaction s'échangent des messages entre eux.

Nous considérerons que, dans une interaction, il n'existe pas d'objet qui n'échange pas de message avec d'autres objets.

Dans une application, tout objet est au moins instance d'une classe concrète. Cette classe est celle qui a permis de construire l'objet. En UML, les objets qui participent à une interaction peuvent ne pas avoir de classe dont ils sont instances. Nous appellerons ces objets des objets non typés. Les objets non typés sont utilisés dans les interactions pour spécifier des objets qui ne font que demander la réalisation d'opérations et dont on ne se soucie pas de connaître le type.

Dans une application, tout objet a un identifiant. En UML, les objets qui participent à une interaction peuvent ne pas avoir d'identifiant. Nous appellerons ces objets des objets anonymes. Les objets anonymes sont utilisés dans les interactions pour spécifier des objets qui ne sont utilisés qu'une seule fois. Il ne sert alors à rien de bien les identifier.

Graphique

Les objets qui participent à une interaction sont représentés graphiquement dans un diagramme de séquence par un carré contenant l'identifiant de l'objet (si l'objet n'est pas anonyme), suivi du nom de la classe dont l'objet est instance (si l'objet est typé). Attaché à ce carré, une ligne verticale représente la vie de l'objet dans le temps (l'axe du temps étant dirigé vers le bas du diagramme).

La figure 6.1 représente quatre objets, dont un objet anonyme non typé, un objet anonyme typé, un objet identifié non typé et un objet identifié typé.

Dans le cadre de ce cours, nous préconisons d'identifier et de typer quasiment tous les objets de toutes les interactions. Cela permet un meilleur suivi des cohérences entre les

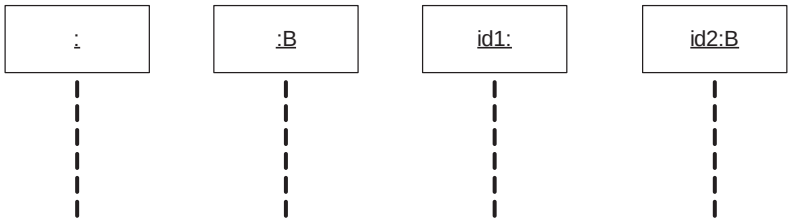


Figure 6.1
Représentation graphique des objets dans les interactions

différentes parties du modèle. En fait, nous utilisons les objets non typés pour spécifier les objets externes au système (comme les utilisateurs), dont nous ne connaissons pas le type.

Message

Sémantique

Dans une application orientée objet, les objets communiquent par échanges de messages. Le message le plus important est le message de demande de réalisation d'opération, par lequel un objet demande à un autre objet (ou à lui-même) de réaliser une des opérations dont il est responsable. En théorie, avec ce message seul, il est possible de décrire complètement le comportement d'une application.

UML intègre donc le message d'appel d'opération dans les interactions entre objets. Plus précisément, UML propose deux messages d'appel d'opération : un message pour les appels synchrones (l'appelant attend de recevoir le résultat de l'opération avant de continuer son activité) et un message pour les appels asynchrones (l'appelant n'attend pas de recevoir le résultat de l'opération et continue son activité après avoir envoyé son message).

UML propose aussi des messages de création et de suppression d'objets afin de gérer le cycle de vie des objets participant à une interaction. Dans une interaction UML, les objets peuvent soit exister au début de l'interaction, soit être créés par d'autres objets pendant l'interaction. Il est aussi possible de spécifier des suppressions d'objets. Celles-ci sont initiées par des objets participant à l'interaction. Un objet détruit ne peut plus recevoir de message.

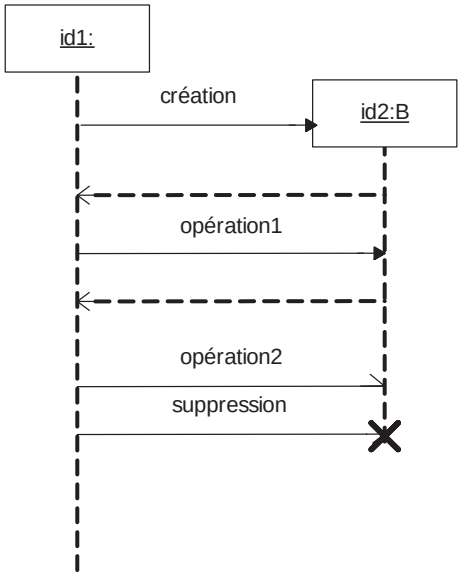
Graphique

La figure 6.2 représente graphiquement les quatre messages suivants supportés dans les interactions UML :

- Le premier message est un message de création échangé entre l'objet identifié `id1` et l'objet identifié `id2`. Ce message signifie que l'objet `id1` crée l'objet `id2`.
- Le deuxième message est un message d'appel synchrone d'opération. L'objet `id1` demande à l'objet `id2` de réaliser l'opération nommée `opération1`. L'objet `id1` attend que l'objet `id2` finisse de réaliser cette opération avant de continuer son activité. Le message de fin de traitement est représenté par une flèche pointillée.

- Le troisième message est un message d'appel asynchrone d'opération. Ce message signifie que l'objet `id1` demande à l'objet `id2` de réaliser l'opération nommée `opération2`. Une même opération sur un même objet peut être appelée de manière synchrone et asynchrone dans une même interaction. Il aurait donc été possible d'appeler à nouveau l'opération `opération1` mais de manière asynchrone. L'appel étant asynchrone, l'objet `id1` n'a pas besoin d'attendre la fin du traitement de l'opération pour continuer son activité.
- Le dernier message est un message de suppression. L'objet `id1` supprime l'objet `id2`.

Figure 6.2
Représentation graphique des messages dans une interaction



Le temps dans les diagrammes de séquence

Une interaction spécifie une succession d'échanges de messages entre les objets participant à l'interaction. Le temps est donc très important puisqu'il précise l'ordre d'exécution entre tous les messages d'une même interaction.

Dans une interaction UML, le temps est gouverné par deux règles principales *(voir ci-après)*. Ces règles considèrent les messages d'appel synchrone d'opération et de création comme deux messages, un appel et un retour. Il y a donc une flèche de l'objet émetteur vers l'objet récepteur (appel) et une autre de l'objet récepteur vers l'objet émetteur (retour).

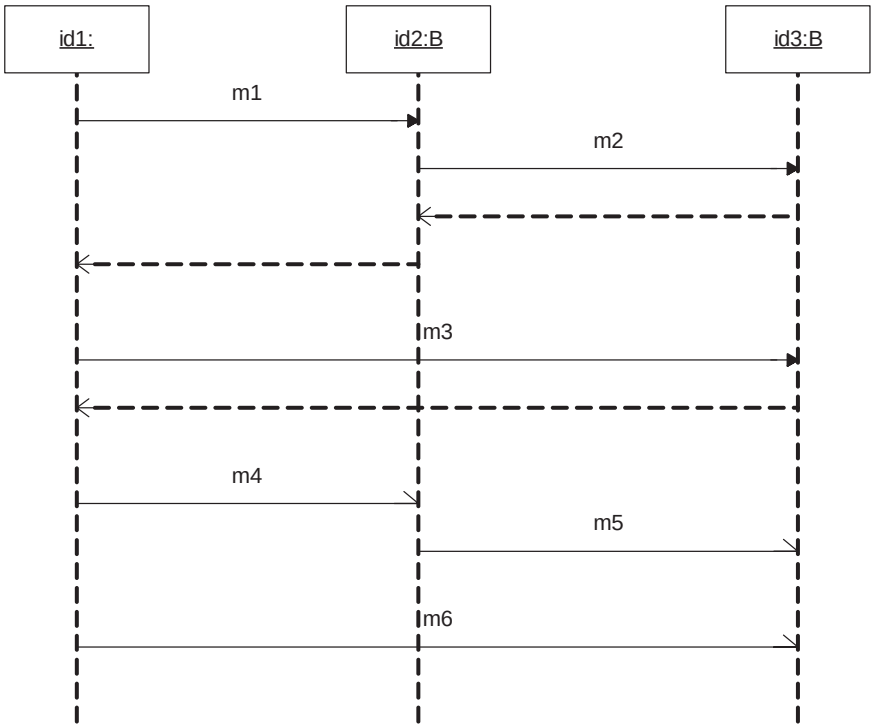
Chaque message est ensuite décomposé en deux événements, un événement d'envoi et un événement correspondant à la réception. L'envoi est matérialisé par l'extrémité de départ de la flèche correspondant au message et la réception par l'extrémité d'arrivée de la flèche. Ainsi, le temps est régulé par les règles suivantes :

1. Sur l'axe d'un objet, tous les événements sont ordonnés du haut vers le bas. Cela entraîne qu'un événement arrive avant un autre événement s'il est positionné plus haut sur l'axe d'un même objet.
2. Pour un même message, l'envoi se déroule toujours avant la réception.

Grâce à ces deux règles, il est possible de définir un ordre partiel entre tous les événements d'une interaction UML.

La figure 6.3 présente une interaction entre trois objets qui s'échangent des messages d'appels synchrones et asynchrones d'opérations.

Figure 6.3
Diagramme de séquence



Grâce à nos deux règles, nous savons que ces échanges obéissent à l'ordre suivant :

Grâce à la règle 1 :

1. Sur l'axe de l'objet `id1` : `m1` (appel/envoi) avant `m1` (retour/réception) avant `m3` (appel/envoi) avant `m3` (retour/réception) avant `m4` (appel/envoi) avant `m6` (appel/envoi).
2. Sur l'axe de l'objet `id2` : `m1` (appel/réception) avant `m2` (appel/envoi) avant `m2` (retour/réception) avant `m1` (retour/envoi) avant `m4` (appel/réception) avant `m5` (appel/envoi).
3. Sur l'axe de l'objet `id3` : `m2` (appel/réception) avant `m2` (retour/envoi) avant `m3` (appel/réception) avant `m3` (retour/envoi) avant `m5` (appel/réception) avant `m6` (appel/réception).

Grâce à la règle 2 :

4. Pour tous les messages `mi` (x/envoi) avant `mi` (x/réception).

Grâce à ces déductions, nous pouvons voir, par exemple, qu'il n'existe pas d'ordre entre `m5` (appel/envoi) et `m6` (appel/envoi). Cela signifie que l'interaction ne précise pas que l'envoi de l'appel asynchrone de l'opération `m5` se fait avant l'envoi de l'appel asynchrone de l'opération `m6`, même si le diagramme semble indiquer le contraire. Notons cependant que la réception de l'appel asynchrone de l'opération `m5` se fait avant la réception de l'appel asynchrone de l'opération `m6`.

Liens avec la vue structurelle du modèle

Nous avons insisté fortement dans les premiers chapitres de ce cours sur les relations de cohérence qui existent entre les différentes parties d'un même modèle.

Nous avons pour l'instant présenté les vues structurelle et comportementale d'un modèle UML. Nous précisons dans cette section les règles de cohérence entre ces deux vues.

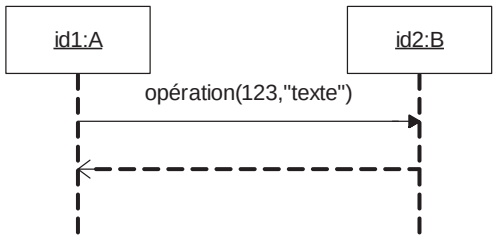
Objet et classe

Les seules relations de cohérence que nous considérons entre les diagrammes de séquence et les diagrammes de classes dans le cadre de ce cours sont les suivantes :

- Tout objet participant à une interaction doit obligatoirement avoir son type décrit sous forme de classe dans la partie structurelle. Nous déconseillons fortement l'utilisation d'objets non typés.
- Tout message d'appel d'opération (synchrone ou asynchrone) doit cibler une opération spécifiée dans la vue structurelle. Cette opération doit appartenir à la classe dont l'objet qui reçoit le message est instance.
- Tout message d'appel d'opération (synchrone ou asynchrone) doit porter les valeurs des paramètres de l'opération ciblée par le message.

Considérons, par exemple, l'interaction représentée par le diagramme de la figure 6.4.

Figure 6.4
Objets typés dans une interaction



Les règles de cohérence entre parties de modèles nous imposent d'avoir dans la partie structurelle la définition des classes `A` et `B` ainsi que la définition de l'opération `opération1`

contenue dans la classe `B`. Notons que l'opération `opération1` possède deux paramètres de direction `in` et dont les types sont respectivement `integer` et `string`

La figure 6.5 représente le diagramme de classes correspondant à cette partie structurelle (les paramètres de l'opération de la classe `B` sont masqués).

Figure 6.5
Classes des objets typés

Les règles de cohérence que nous venons de présenter imposent des contraintes sur la partie comportementale du modèle ainsi que sur la partie structurelle. Pour autant, elles n'imposent aucune contrainte sur la façon de créer un modèle cohérent.

Il est donc parfaitement envisageable de commencer la construction d'un modèle cohérent par la partie comportementale puis de finir par créer une partie structurelle cohérente. Inversement, il est aussi possible de commencer par la partie structurelle puis de finir par la partie comportementale. La troisième approche possible est de construire en parallèle les parties comportementale et structurelle. Nous conseillons fortement de tester chacune de ces trois approches afin de trouver celle qui convient le mieux à sa propre façon de penser.

Diagramme et modèle

Nous avons vu au chapitre 3 la différence entre diagramme UML et modèle UML. Rappelons qu'un diagramme est une représentation graphique d'un modèle et qu'à un modèle peuvent correspondre plusieurs diagrammes. Cette relation que nous avons illustrée sur la partie structurelle du modèle est tout aussi importante pour la partie comportementale du modèle.

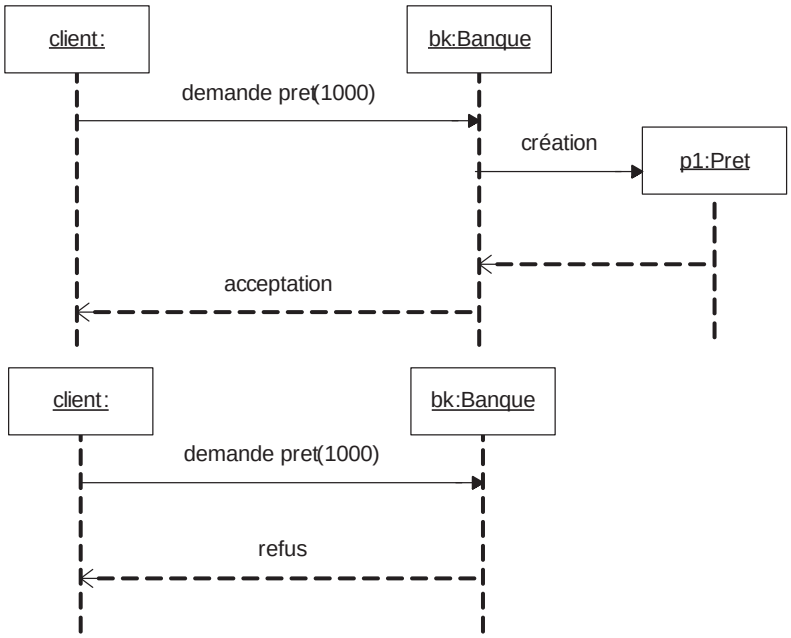
Un diagramme de séquence est la représentation graphique de la partie comportementale (interaction) d'un modèle UML. Toute les informations (objets, messages, etc.) sont contenues dans le modèle et représentées graphiquement à l'aide des diagrammes. Il est donc possible de représenter une même information dans différents diagrammes.

En fait, seuls les objets sont représentés dans plusieurs diagrammes. Cela permet de représenter graphiquement le fait qu'un même objet participe à plusieurs interactions. L'objectif est de représenter différentes possibilités d'un même comportement. Nous conseillons, par exemple, de spécifier les comportements nominaux (normal et sans erreur) et les comportements soulevant des erreurs avec les mêmes objets.

L'exemple illustré à la figure 6.6 présente deux interactions qui peuvent s'exécuter dans une application de gestion de prêts bancaires. Ces interactions font intervenir les mêmes objets (le client `c1` et la banque `bk`). Notons que nous avons spécifié le client avec un objet

non typé. La première interaction représente un cas nominal : le prêt est accordé. La deuxième montre un cas soulevant une erreur : le prêt est refusé.

Figure 6.6
Deux diagrammes partageant de mêmes objets



Le fait qu'un objet appartienne à plusieurs interactions n'a pas de conséquence sur l'ordre entre les événements des interactions. Il existe un ordre pour chaque interaction, et ces ordres sont indépendant les uns des autres.

Concepts avancés

Les concepts avancés que nous présentons dans cette section permettent de bien comprendre le rôle des interactions dans un modèle UML vis-à-vis de la génération du code. Les concepts ajoutés dans la version UML 2.1 renforcent d'ailleurs ce rôle.

Interactions et génération de code

Même si nous avons déjà indiqué que les interactions permettaient uniquement de spécifier des exemples d'exécution d'application, il est important de montrer pourquoi elles ne peuvent être utilisées pour spécifier intégralement des algorithmes et ainsi servir à la génération de code.

Il est important de préciser qu'une interaction ne définit qu'une seule exécution possible, alors qu'un algorithme définit l'ensemble des exécutions possibles. Pour spécifier un

algorithme à l'aide d'interactions, il faudrait pouvoir spécifier chacune des exécutions possibles sous la forme d'une interaction. Il faudrait donc que l'ensemble des exécutions possibles soit fini mais aussi que les résultats retournés par l'algorithme ne dépendent que des valeurs données en entrée (déterminisme) et que l'algorithme ne modifie pas les états des objets participant à sa réalisation.

Prenons, par exemple, l'algorithme correspondant à la porte logique ET réalisant l'opération booléenne ET. Celui-ci semble pouvoir être spécifié intégralement à l'aide d'interactions puisque l'ensemble des exécutions possibles est fini, que le résultat ne dépend que de l'entrée et que l'état de l'objet ne change pas après exécution de l'algorithme.

Les quatre exécutions possibles de l'algorithme semblent pouvoir être représentées de la manière illustrée à la figure 6.7.

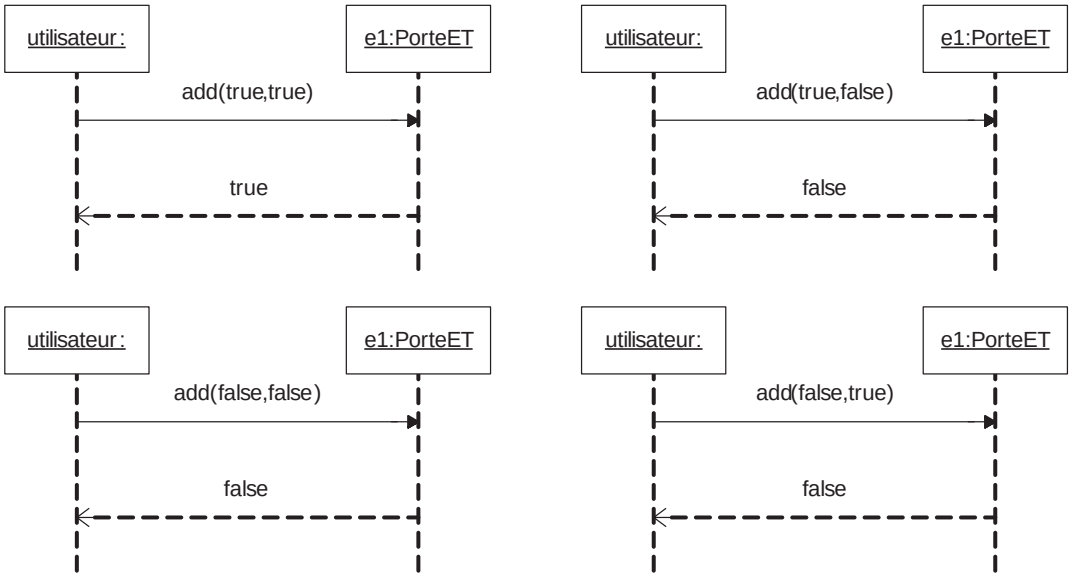


Figure 6.7
Diagrammes de séquence spécifiant l'algorithme de la porte logique ET

Pour pouvoir être totalement sûr que ces interactions spécifient l'intégralité des exécutions de la porte ET, il faut savoir, d'une part, que la porte ET a un comportement déterministe (le résultat ne dépend que de l'entrée) et, d'autre part, que l'exécution de la porte ET ne modifie par l'état de l'objet. En effet, si la porte ET avait un comportement indéterministe ou si le comportement modifiait l'état de l'objet, il ne serait pas possible de générer le code. Malheureusement, il n'est pas possible de préciser ces informations en UML.

En posant toutes ces conditions (ensemble fini d'exécutions, déterminisme et pas de modification de l'état de l'objet), il serait possible de générer le code suivant à partir des diagrammes que nous venons de présenter :

```
public boolean add(boolean a , boolean b) {
    if (a && b) return true ;
    else if (a && !b) return false ;
    else if ( !a && !b) return false ;
    else if ( !a && b) return false ;
}
```

En conclusion, nous pouvons dire qu'il est possible de spécifier des algorithmes et de générer du code à partir d'interactions si l'ensemble des comportements possibles est fini, si le comportement spécifié est déterministe et si le comportement ne modifie pas les états des objets. En d'autres termes, cela n'est pas impossible mais reste suffisamment rare pour ne jamais être employé.

Fragment d'interaction

Dans UML 1.4 et les versions précédentes, il n'était pas possible de composer des interactions entre elles, ni d'intégrer une interaction dans une autre interaction. Cela posait problème parce qu'il n'était pas possible de spécifier et de réutiliser des interactions afin d'en construire de plus complexes.

Ce problème a été complètement résolu avec UML 2.1, qui supporte le concept de fragment d'interaction. Un fragment permet d'identifier une sous-partie d'une interaction afin que celle-ci soit référencée par d'autres interactions. Associé au concept d'interaction, UML 2.1 propose des opérateurs permettant de spécifier des conditions d'exécution telles que les boucles (loop) ou les tests (if then else) sur les fragments d'interactions.

Grâce à ces nouveaux concepts, il est possible de spécifier des exemples d'exécution beaucoup plus complexes et beaucoup plus lisibles grâce à la possibilité de décomposer les interactions en sous-interactions. Cependant, dans le contexte de ce cours, nous n'utilisons pas ces concepts avancés, car nous n'en avons pas besoin pour présenter les avantages qu'apportent les interactions lorsque nous utilisons UML pour le développement d'applications.

Limites intrinsèques des interactions

Les interactions permettent uniquement de présenter des échanges de messages entre objets. Il n'est donc pas possible de spécifier :

- Les accès aux propriétés des objets, à moins d'utiliser des opérations d'accès aux propriétés.
- Les navigations sur les associations navigables.
- Les créations de liens entre objets.
- Les appels aux opérations de classes, car aucun objet n'est directement responsable de la réalisation de l'opération.

Synthèse

Nous avons montré dans ce chapitre comment la vue comportementale pouvait être modélisée en UML à l'aide du concept d'interaction. Une interaction UML permet de spécifier un exemple d'exécution d'une application. Plus précisément, une interaction spécifie les échanges de messages effectués entre plusieurs objets qui composent l'application.

Après avoir présenté les concepts de base des interactions (objet et message), nous avons spécifié les contraintes de cohérence qui existent entre la partie structurelle et la partie comportementale du modèle. Nous avons aussi présenté les différentes approches de construction permettant la réalisation d'un modèle cohérent.

Nous avons en outre montré pourquoi les interactions ne permettaient pas de spécifier des algorithmes et ne permettaient donc pas de générer du code. Nous avons en particulier insisté sur le fait qu'une interaction ne spécifiait qu'une seule exécution possible d'une application.

Pour finir, nous avons introduit les concepts avancés des interactions UML en présentant notamment le concept de fragment, qui a été défini dans la version 2.0 d'UML. Soulignons que nous n'utiliserons pas ces concepts dans la suite de notre cours.

Travaux dirigés

TD6. Diagrammes de séquence UML

L'application `ChampionnatEchecs`, qui devra permettre de gérer le déroulement d'un championnat d'échecs est actuellement en cours de développement. L'équipe de développement n'a pour l'instant réalisé qu'un diagramme de classes de cette application *(voir figure 6.8)*.

La classe `ChampionnatDEchecs` représente un championnat d'échecs. Un championnat se déroule entre plusieurs joueurs (voir classe `Joueur`) et se joue en plusieurs parties (voir classe `Partie`). La propriété `MAX` de la classe `ChampionnatDEchecs` correspond au nombre maximal de joueurs que le championnat peut comporter. La propriété `fermer` permet de savoir si le championnat est fermé ou si de nouveaux joueurs peuvent s'inscrire.

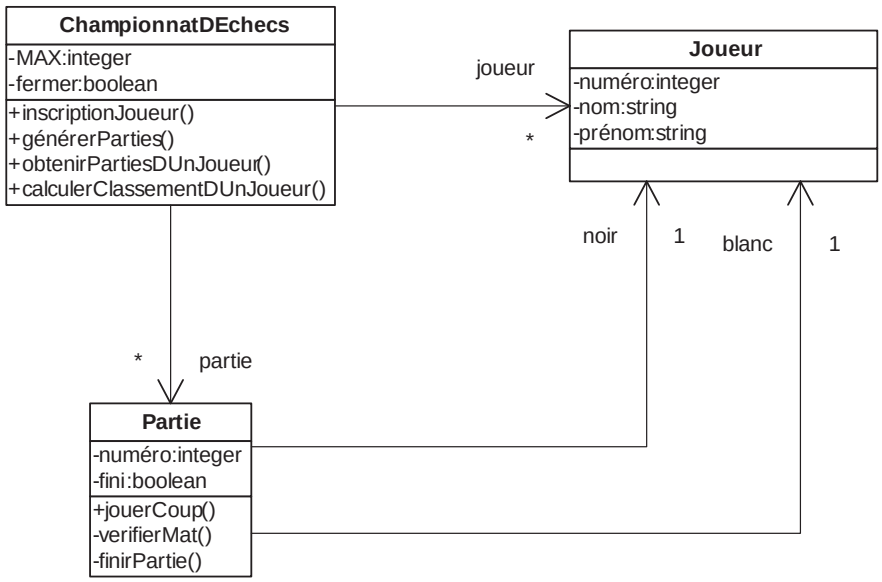
`ChampionnatDEchecs` possède les opérations suivantes :

- `inscriptionJoueur(in nom:string, in prénom:string)` : integer permettant d'inscrire un nouveau joueur dans le championnat si le nombre de joueurs inscrits

n'est pas déjà égal à `MAX` et si le championnat n'est pas déjà fermé. Si l'inscription est autorisée, cette opération crée le joueur et retourne son numéro dans le championnat.

- `générerPartie()` : permet de fermer le championnat et de générer toutes les parties nécessaires.
- `obtenirPartieDUnJoueur(in numéro :integer) : Partie[*]` : permet d'obtenir la liste de toutes les parties d'un joueur (dont le numéro est passé en paramètre).
- `calculerClassementDUnJoueur(in numéro :interger)` : integer permettant de calculer le classement d'un joueur (dont le numéro est passé en paramètre) pendant le championnat.

Figure 6.8
Classes de l'application ChampionnatEchecs



La classe `Partie` représente une des parties du championnat. La classe `Partie` est d'ailleurs associée avec la classe `ChampionnatDEchecs`, et l'association précise qu'un championnat peut contenir plusieurs parties. Une partie se joue entre deux joueurs. Un joueur possède les pièces blanches et commence la partie alors que l'autre joueur possède les pièces noires. Les associations entre les classes `Partie` et `Joueurs` précisent cela. La propriété `numéro` correspond au numéro de la partie (celui-ci doit être unique). La propriété `fini` permet de savoir si la partie a déjà été jouée ou pas.

La classe `Partie` possède les opérations suivantes :

- `jouerCoup(in coup:string)` : permet de jouer un coup tant que la partie n'est pas finie. Le traitement associé à cette opération fait appel à l'opération `vérifierMat` afin de savoir si le coup joué ne met pas fin à la partie. Si tel est le cas, l'opération `finirPartie` est appelée.
- `vérifierMat()` : boolean permettant de vérifier si la position n'est pas mat.
- `finirPartie` : permet de préciser que la partie est finie. Il n'est donc plus possible de jouer de nouveaux coups.

La classe `Joueur` représente les joueurs du championnat. La classe `Joueur` est d'ailleurs associée avec la classe `ChampionnatDEchecs`, et l'association précise qu'un championnat peut contenir plusieurs joueurs. La propriété `numéro` correspond au numéro du joueur (celui-ci doit être unique). Les propriétés `nom` et `prénom` permettent de préciser le nom et le prénom du joueur.

Un championnat d'échecs se déroule comme suit :

- Un administrateur de l'application crée un championnat avec une valeur `MAX`.
- Les participants peuvent s'inscrire comme joueurs dans le championnat.
- L'administrateur crée l'ensemble des parties.
- Les participants, une fois inscrits, peuvent consulter leur liste de parties.
- Les participants, une fois inscrits, peuvent jouer leurs parties. Nous ne nous intéressons qu'aux coups joués par chacun des deux joueurs. Nous ignorons l'initialisation de la partie (identification du joueur qui a les pions blancs et donc qui commence la partie).
- Les participants peuvent consulter leur classement.

Dans les questions suivantes, nous allons spécifier des exemples d'exécution de `ChampionnatDEchecs` avec des diagrammes de séquence.

Question 52 *Comment modéliser les administrateurs et les participants ?*

Question 53 *Représentez par un diagramme de séquence le scénario d'exécution correspondant à la création d'un championnat et à l'inscription de deux joueurs. Vous assurerez la cohérence de votre diagramme avec le diagramme de classes fourni à la figure 6.8.*

Question 54 *Représentez par un diagramme de séquence le scénario d'exécution correspondant à la création de l'ensemble des parties pour le championnat créé à la question 53. Vous assurerez la cohérence de votre diagramme avec le diagramme de classes fourni à la figure 6.8.*

Question 55 *Représentez par un diagramme de séquence le scénario d'exécution correspondant au déroulement de la partie d'échecs entre deux joueurs. Vous pouvez considérer une partie qui se termine en quatre coups. Vous assurerez la cohérence de votre diagramme avec le diagramme de classes fourni à la figure 6.8..*

Question 56 *Est-il possible de générer automatiquement le code d'une opération de cette application à partir de plusieurs diagrammes de séquence ?*

Question 57 *Est-il possible de construire des diagrammes de séquence à partir du code d'une application ?*

Une équipe de développement souhaite réaliser une application `Calculus` qui permet à des utilisateurs d'effectuer des opérations arithmétiques simples sur des entiers : addition, soustraction, produit, division. Cette application a aussi une fonction mémoire qui permet à l'utilisateur de stocker un nombre entier qu'il pourra ensuite utiliser pour n'importe quelle opération. Les opérations peuvent directement s'effectuer sur la mémoire. L'utilisateur se connecte et ouvre ainsi une nouvelle session. Puis, dans le

cadre d'une session, l'utilisateur peut demander au système d'effectuer une suite d'opérations.

Question 58 *Utilisez des diagrammes de séquences pour représenter les différents scénarios d'exécution du service* `Calculus`*.*

Question 59 *Pour chacune des instances apparaissant dans votre diagramme de classes, créez la classe correspondante.*

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

- Vous savez élaborer un diagramme de séquence cohérent avec un diagramme de classes.
- Vous savez élaborer un diagramme de classes cohérent avec un ensemble de diagrammes de séquence.
- Vous avez compris la relation qui existe entre une interaction et du code.