Corrigés des TD : Un curieux besoin de modélisation

À partir du code donné en annexe, répondez aux questions suivantes.

1. *En une phrase, quels sont les rôles de chacune des classes ?*

Voici la liste des classes et leur rôle :

- `Repertoire` représente le répertoire (dans l'application il n'y en a qu'un).
- `Personne` représente une fiche d'une personne dans le répertoire.
- `Adress` représente une adresse postale.
- `UIRepertoire` correspond à la représentation graphique d'un répertoire à partir de laquelle les actions sur le répertoire pourront être appelées.
- `UIPersonne` correspond à la représentation graphique d'une fiche d'une personne à partir de laquelle les actions sur la fiche d'une personne pourront être appelées.
- `UIMenuActionListener` représente la classe chargée de capter tous les clics destinés à interagir avec un répertoire et d'appeler les traitements associés.
- La classe anonyme définie dans la classe `UIPersonne` est chargée de capter tous les clics destinés à interagir avec la fiche d'une personne et d'appeler les traitements associés.
- `MyAssistant` est la classe qui contient la fonction principale (le main) de l'application. Elle crée l'interface associée au répertoire géré par l'application.

2. *Peut-on dire qu'il existe des classes représentant des données et des classes représentant des interfaces graphiques ? Si oui, pourquoi et quelles sont ces classes ?*

Toutes les classes commençant par `UI` sont graphiques. Les classes `UIRepertoire` et `UIPersonne` représentent les interfaces graphiques permettant la gestion des réper-

toires et des personnes, tandis que la classe `UIMenuActionListener` est en charge de la gestion des actions sur les interfaces et de la réalisation des fonctionnalités associées à ces actions.

Les classes `Repertoire`, `Personne` et `Adresse` représentent les données manipulées par l'application.

La classe `MyAssistant` ne représente ni des données ni des interfaces graphiques ; elle sert à « lancer » l'application.

3. *Est-il possible que le numéro de téléphone d'une personne soit +33 1 44 27 00 00 ?*

Oui, parce que les propriétés `telephoneMaison`, `telephonePortable` et `telephoneBureau` de la classe `Personne` sont de type string et qu'aucune vérification n'est faite sur sa valeur.

4. *Est-il possible que l'adresse e-mail d'une personne soit « je_ne_veux_pas_donner_mon_email » ?*

Oui, parce que la propriété `mail` de la classe `Personne` (qui correspond à l'adresse e-mail) est de type string et qu'aucune vérification n'est faite sur sa valeur.

5. *Quelles sont les fonctionnalités proposées par les menus graphiques de cette application ?*

Il faut regarder les classes gérant les interfaces graphiques. Elles créent, dans notre cas, soit des menus déroulants, soit des boutons de soumission.

La classe `UIRepertoire` crée trois menus déroulants : Fichier, Organisation et Aide. Les fonctionnalités sont les éléments de chacun de ces trois menus. Mis à part le nom de ces fonctionnalités, le code de cette classe ne permet pas de connaître le service réalisé par chacune de ces fonctionnalités.

Voici chacun des menus avec les éléments qu'il contient :

- Menu Fichier :
  - Nouveau
  - Ouvrir
  - Enregistrer
  - Enregistrer Sous
- Menu Organisation :
  - Ajouter Nouvelle Personne
  - Rechercher Personne(s)
- Menu Aide :
  - A Propos

La classe `UIPersonne` est l'interface graphique associée à une personne. Elle crée deux boutons de soumission : Save et Cancel. Comme pour les menus déroulants, seul le nom des boutons peut nous renseigner sur la fonctionnalité associée.

6. *Quelles sont les fonctionnalités réellement réalisées par cette application ?*

Pour avoir la réponse, il faut regarder dans le code associé aux classes de gestion des interfaces.

Pour un répertoire, il s'agit de la classe `UIMenuActionListener`, dans le code de laquelle nous constatons que seules les fonctionnalités Ajouter Nouvelle Personne et Nouveau sont réalisées. Les autres fonctionnalités ne font qu'afficher un message.

Pour une personne, il n'y a pas de classe nommée associée à la gestion de l'interface. Il faut regarder, dans le code de la classe `UIPersonne`, le code associé à la classe anonyme assurant la gestion de l'interface. Nous constatons alors que les fonctionnalités Save et Cancel sont réalisées.

7. *Est-il possible de sauvegarder un répertoire dans un fichier ?*

C'est a priori possible puisque l'option Enregistrer Sous du menu Fichier le laisse supposer. Cependant, le code nous informe que cette fonctionnalité n'est pas encore réalisée.

8. *Si vous aviez à rédiger un document décrivant tout ce que vous savez sur cette application afin qu'il puisse être lu par un développeur qui veut réutiliser cette application et un chef de projet qui souhaite savoir s'il peut intégrer cette application, quelles devraient être les caractéristiques de votre document ?*

Le document doit contenir toutes les informations relatives aux différentes vues (diversité), aux différents niveaux d'abstraction sans oublier les relations de cohérence reliant tous ces éléments.

Plus précisément, à partir du code de l'application, il est possible (mais pas simple) de rédiger une documentation :

- des services offerts par l'application (utile au développeur et au chef de projet) ;
- de conception de l'application (utile au développeur) ;
- d'architecture de l'application (utile au développeur) ;
- des logiciels nécessaires à l'utilisation de l'application (utile au développeur et au chef de projet).

9. *Rédigez un document présentant l'application* `MyAssistant`*.*

L'application `MyAssistant` permet de gérer des répertoires. Un répertoire contient des informations relatives à des personnes. Pour chaque personne, il est possible de stocker un nom, un prénom, un numéro de téléphone du domicile, un numéro de téléphone du travail, un numéro de téléphone de portable, un numéro de fax, un titre, une société, une adresse et une adresse e-mail.

10. *Rédigez un document décrivant les fonctionnalités de l'application* `MyAssistant`*.*

Attention : l'interprétation des fonctionnalités non réalisées par le code Java fourni se fait en fonction du nom du menu associé et par similitude à ce qui se fait dans la majorité des applications.

Les fonctionnalités offertes par l'application sont de deux sortes. Les fonctionnalités sur un répertoire et les fonctionnalités sur une personne.

Les fonctionnalités de gestion d'un répertoire sont les suivantes :

- Créer un nouveau répertoire (menu Fichier/Nouveau).
- Ouvrir un répertoire déjà existant (menu Fichier/Ouvrir).
- Enregistrer un répertoire, c'est-à-dire enregistrer toutes les informations sur les personnes identifiées dans le répertoire (menu Fichier/Enregistrer). L'enregistrement se fait dans le fichier d'origine du répertoire s'il s'agit du répertoire déjà existant. Sinon, l'enregistrement se fait dans un nouveau fichier que l'utilisateur aura à identifier.
- Enregistrer un répertoire dans un autre fichier que le fichier d'origine, s'il existe (menu Fichier/Enregistrer Sous).
- Ajouter une nouvelle personne dans le répertoire ouvert (menu Organisation/Ajouter Nouvelle Personne). Cette fonctionnalité propose à l'utilisateur de saisir les informations correspondant à une nouvelle personne.
- Rechercher une personne (menu Organisation/Rechercher Personne). Cette fonctionnalité permet de rechercher la totalité des informations sur une personne en n'en saisissant qu'une partie (nom, numéro de téléphone, une partie du nom, etc.). Il faudrait avoir le cahier des charges ou un code complet pour savoir exactement à partir de quoi la recherche peut être faite.
- Afficher l'aide sur l'application (menu Aide/A Propos). Cette fonctionnalité permet à l'utilisateur d'accéder à l'aide disponible sur l'application. Là aussi, il n'y a aucune information sur la forme de cette aide.

Les fonctionnalités de gestion d'une personne sont les suivantes :

- Sauvegarder les informations saisies pour une personne (bouton Save de l'interface graphique associée à une personne). Attention : pour ajouter effectivement une personne à un répertoire, il faut enregistrer le répertoire. Se contenter de sauvegarder les informations relatives à la personne n'est pas suffisant.
- Annuler les modifications faites sur les informations d'une personne (bouton Cancel de l'interface graphique associée à une personne). Cette fonctionnalité annule toutes les modifications faites depuis la dernière sauvegarde des informations.

11. *Rédigez un document décrivant l'architecture générale de l'application* `MyAssistant`.

Une solution (qui n'est pas unique) consiste à considérer un composant pour :

- l'interface graphique ;
- la base de données stockant les informations sur les personnes et les répertoires ;
- l'interface avec la base de données ;
- la réalisation de la « logique » des fonctionnalités de l'application.

La figure 1 illustre les liens de communication qui existent entre les différents composants. Il est important de donner la légende du schéma.

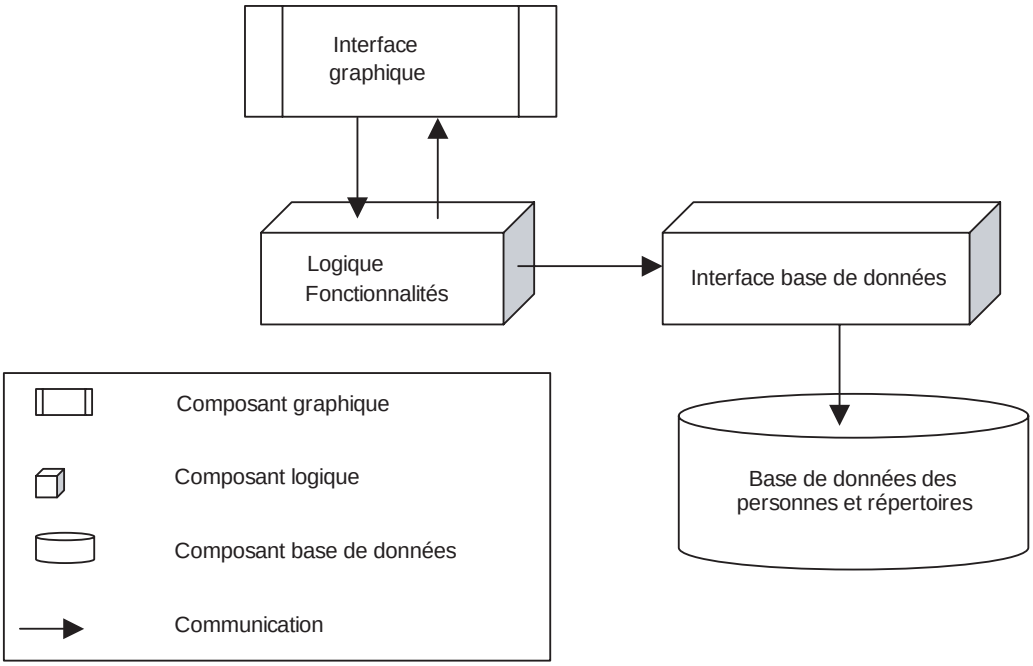


Figure 1
Architecture générale de MyAssistant

TD2. Diagrammes de classes

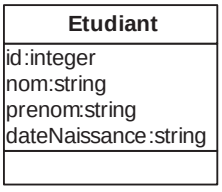
12. *Définissez la classe UML représentant un étudiant, caractérisé, entre autres, par un identifiant, un nom, un prénom et une date de naissance.*

Voici une solution ne prenant en compte que les propriétés d'un étudiant nommées dans la question.

La classe se nomme `Etudiant`, et elle possède les quatre propriétés suivantes, représentées à la figure 2 :

- `id`, de type Integer, qui représente l'identifiant de l'étudiant.
- `nom`, de type String, qui représente le nom de l'étudiant.
- `prenom`, de type String, qui représente le prénom de l'étudiant.
- `dateNaissance`, de type String, qui représente la date de naissance de l'étudiant.

Figure 2
Classe Etudiant

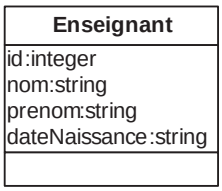


Cette définition n'impose aucune règle sur le format de saisie de la date de naissance.

13. *Définissez la classe UML représentant un enseignant, caractérisé, entre autres, par un identifiant, un nom, un prénom et une date de naissance.*

La réponse est similaire à celle de la question précédente, si ce n'est que le nom de la classe est maintenant `Enseignant`, comme l'illustre la figure 3.

Figure 3
Classe Enseignant



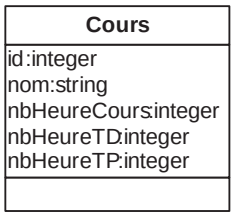
14. *Définissez la classe UML représentant un cours, caractérisé par un identifiant, un nom, le nombre d'heures de cours magistral, le nombre d'heures de travaux dirigés et un nombre d'heures de travaux pratiques que doit suivre un étudiant.*

La question étant exhaustive sur les propriétés de la classe, il n'y a pas de choix. Les seules libertés sont sur les types des propriétés. Voici une solution possible.

La classe se nomme `Cours`, et elle possède les cinq propriétés suivantes (voir figure 4) :

- `id`, de type integer, qui représente l'identifiant du cours.
- `nom`, de type string, qui représente le nom du cours.
- `nbHeuresCours`, de type integer, qui représente le nombre d'heures de cours magistral.
- `nbHeuresTD`, de type integer, qui représente le nombre d'heures de travaux dirigés.
- `nbHeuresTP`, de type integer, qui représente le nombre d'heures de travaux pratiques.

Figure 4 '
Classe Cours



15. *Définissez les associations qui peuvent exister entre un enseignant et un cours.*

La question n'est pas assez précise pour qu'une seule solution soit possible. Toutes les associations réalistes sont possibles.

Nous considérons deux associations entre ces classes. La relation `Responsabilité` permet d'associer à un cours l'enseignant qui en est responsable et d'associer à un

enseignant l'ensemble des cours qu'il gère (un enseignant pouvant ne gérer aucun cours). La relation `Dispenser` permet d'associer à un cours l'ensemble des enseignants qui y participent (nous parlons alors des intervenants d'un cours), cet ensemble ne devant pas être vide. Cette relation permet aussi d'associer à un enseignant l'ensemble des cours qu'il dispense, ensemble qui lui non plus ne peut être vide.

Les associations représentées à la figure 5 prennent en compte toutes les contraintes précédentes.

Figure 5

Associations entre les classes Cours *et* Enseignant

Nous ne nous sommes pas intéressés à la navigabilité des associations. Cependant, même si les associations ne sont pas navigables, des liens peuvent être créés entre objets instances des classes. Dans ce cas, les objets seront liés mais n'auront pas connaissance des liens. Il ne sera, par exemple, pas possible de passer par un cours pour accéder à son responsable. La façon dont les choses seront réellement réalisées dépend de la plate-forme d'exécution utilisée.

16. *Définissez la classe UML représentant un groupe d'étudiants en utilisant les associations.*

Il n'est pas possible d'utiliser une association groupe de la classe `Etudiant` vers elle-même. Cette association permettrait de relier des étudiants entre eux mais ne permettrait pas d'identifier les différents groupes. Il est donc nécessaire d'avoir une représentation nous permettant d'identifier les groupes.

Nous ajoutons donc une classe `Groupe`, que nous associons à la classe `Etudiant`. L'association entre les deux classes se nomme `Groupement`. Nous considérons qu'un étudiant peut appartenir à plusieurs groupes, voire aucun, qu'un groupe contient au moins deux membres mais n'a pas de limite supérieure sur le nombre de ses membres. L'association est graphiquement représentée à la figure 6.

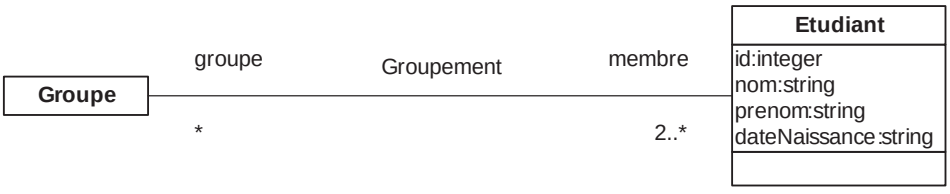


Figure 6

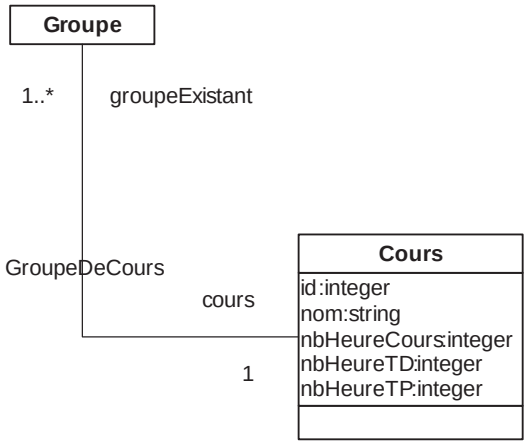
Classe Groupe

17. *Définissez l'association possible entre un groupe d'étudiants et un cours.*

Nous nommons l'association `GroupeDeCours`. Nous considérons que les groupes sont redéfinis pour chaque cours et qu'il faut au moins un groupe pour qu'un cours ait lieu. En conséquence, à un groupe nous n'associons qu'un seul cours, et à un cours nous associons un à plusieurs groupes.

La figure 7 représente graphiquement l'association `GroupeDeCours`.

Figure 7
Association GroupeDeCours



18. *Pensez-vous qu'il soit possible de définir un lien d'héritage entre les classes UML représentant respectivement les étudiants et les enseignants ?*

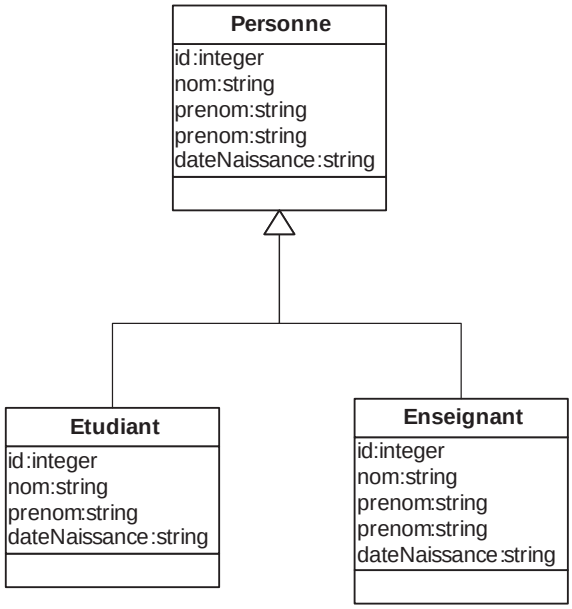
Pour pouvoir définir un lien d'héritage entre deux classes, il faut que l'ensemble des objets instances d'une des deux classes soit inclus dans l'ensemble des objets instances de l'autre. Ce n'est pas le cas ici, puisque tous les étudiants ne sont pas des enseignants et que tous les enseignants ne sont pas des étudiants. Le fait que des enseignants puissent aussi étudier certains cours ne peut se représenter avec un lien d'héritage entre classes.

Le lien entre ces deux classes d'objets est qu'ils partagent certaines propriétés propres à une personne. Si nous voulons faire apparaître ce lien, il nous faut introduire une classe `Personne`, comme le montre la figure 8. Les classes `Etudiant` et `Enseignant` héritent alors de la classe `Personne`.

19. *Pensez-vous qu'il soit possible de définir un lien d'héritage entre les classes UML représentant respectivement les étudiants et les groupes d'étudiants ?*

Ce n'est pas possible pour la même raison que précédemment. Les classes `Etudiant` et `Groupe` représentent des ensembles d'objets différents ne présentant aucune relation d'inclusion. Les groupes ne sont pas des étudiants (et inversement). Par contre, les groupes sont composés d'étudiants, et les étudiants appartiennent à des groupes, ce qui est représenté par l'association `Groupement`.

Figure 8
Héritage pour les classes Etudiant *et* Enseignant



20. *On nomme* `coursDeLEtudiant()` *l'opération permettant d'obtenir l'ensemble des cours suivis par un étudiant. Positionnez cette opération dans une classe, puis précisez les paramètres de cette opération, ainsi que les modifications à apporter aux associations préalablement identifiées pour que votre solution soit réalisable.*

Il semble assez naturel de mettre cette opération dans la classe `Etudiant`, car les associations permettent, à partir d'un étudiant, de retrouver l'ensemble des cours auxquels il assiste. Dans ce cas, les paramètres d'entrée (`in`) sont inexistants, et l'opération s'applique sur l'objet courant (l'étudiant). Le type du retour de l'opération est un ensemble, éventuellement vide, de cours de type `Cours`.

Pour mettre en place cette solution, il est indispensable de pouvoir accéder aux groupes auxquels un étudiant appartient et, depuis ces groupes, de pouvoir accéder au cours associé à chacun d'eux. Il faut alors mettre les navigabilités adéquates sur l'association `Groupement` de la classe `Etudiant` vers la classe `Groupe` et sur l'association `GroupeDeCours` de la classe `Groupe` vers la classe `Cours`.

Si nous acceptons d'ajouter des associations par rapport à ce qui a déjà été fait, une solution pour mettre en place notre solution est de définir une association entre les classes `Etudiant` et `Cours` permettant d'associer directement un étudiant aux cours qu'il suit et un cours aux étudiants qui le suivent. Cette association doit être navigable de la classe `Etudiant` vers la classe `Cours`.

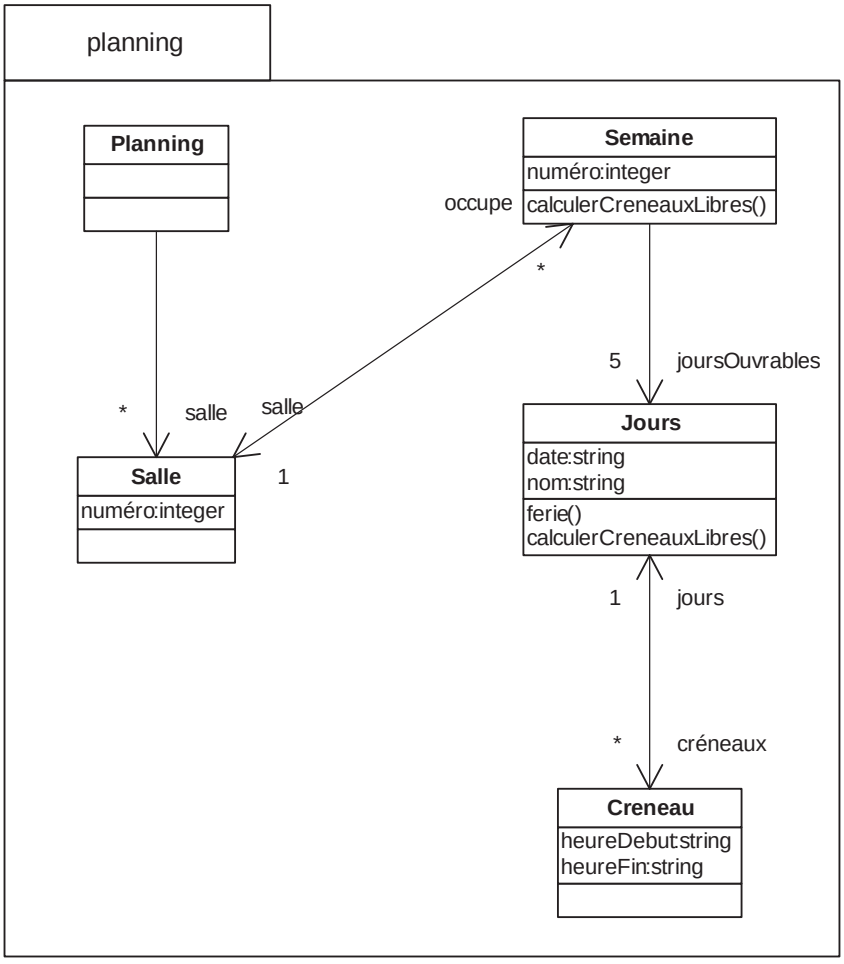
21. *Nous nommons* `coursDeLEnseignant()` *l'opération permettant d'obtenir l'ensemble des cours dans lesquels intervient un enseignant. Positionnez cette opération dans une classe, puis précisez les paramètres de cette opération, ainsi que les modifications à apporter aux associations préalablement identifiées afin que votre solution soit réalisable.*

Il semble assez naturel de mettre cette opération dans la classe `Enseignant`, car les associations permettent, à partir d'un enseignant, de retrouver l'ensemble des cours dans lesquels il intervient. Dans ce cas, les paramètres d'entrée (`in`) sont inexistants, et l'opération s'applique sur l'objet courant (l'enseignant). Le type du retour de l'opération est un ensemble, éventuellement vide, de cours de type `Cours`.

Pour mettre en place cette solution, il est indispensable de pouvoir accéder aux cours que dispense un enseignant. Il faut donc mettre les navigabilités adéquates sur l'association `Dispenser` de la classe `Enseignant` vers la classe `Cours`.

22. *Expliquez le diagramme de classes représenté à la figure 9.*

Figure 9
Diagramme de classes du package planning



Le diagramme de classes du package `planning` représente l'occupation d'un ensemble de salles (éventuellement vide).

Ce package contient cinq classes, `Planning`, `Salle`, `Semaine`, `Jours` et `Creneau`.

La classe `Creneau` contient les informations d'un créneau horaire déterminé par une heure de début et une heure de fin. Un créneau est associé à un jour. Il permet d'identifier un créneau d'occupation d'une salle un jour donné.

La classe `Jours` contient les informations sur l'ensemble de créneaux occupés de ce jour. Un jour est déterminé par une date et un nom (jour dans la semaine). Il est possible de savoir si un jour correspond à un jour férié et quels sont ses créneaux libres. Cette dernière opération est possible grâce à l'association navigable entre les classes `Jours` et `Creneau`, qui permet d'avoir la liste des créneaux occupés pour un jour donné. Cette classe permet donc d'identifier les créneaux d'occupation d'une salle pour un jour donné.

La classe `Semaine` contient les informations sur les créneaux composant l'ensemble des jours de la semaine. Une semaine est représentée par son numéro. L'association navigable de la classe `Semaine` vers la classe `Jours` permet d'avoir accès aux informations sur les créneaux occupés de l'ensemble des jours de la semaine. Cette association est nécessaire pour la réalisation de l'opération `calculerCreneauxLibres()`, qui permet d'identifier les créneaux libres pour une semaine donnée. Nous avons en plus l'information qu'une `Semaine` est composée de cinq jours ouvrables exactement. Cette classe permet donc d'identifier les créneaux d'occupation d'une salle pour une semaine donnée.

La classe `Salle` contient les informations sur l'occupation d'une salle. Une salle est représentée par son numéro. L'association navigable entre la classe `Salle` et la classe `Semaine` permet de récupérer l'ensemble des informations sur l'occupation de la salle pour un nombre quelconque de semaines. Une semaine est associée à une seule salle. Une même semaine (en terme de numéro) existera en autant d'exemplaires que de salles occupées cette même semaine. Chacune de ces semaines représente l'occupation d'une salle en particulier. Cette classe permet donc d'identifier les créneaux d'occupation d'une salle pour un ensemble de semaines.

La classe `Planning` contient les informations sur l'occupation d'un ensemble de salles (éventuellement vide). Elle ne contient aucune propriété. Grâce à une association vers la classe `Salle`, il est possible d'accéder à l'ensemble des salles et de récupérer les informations sur leur occupation.

23. *Positionnez toutes vos classes (`Etudiant`, `Enseignant`, `Cours`, `GroupeEtudiant`) dans un package nommé `personnel`.*

La solution est donnée par la figure 10.

24. *Liez vos classes afin de faire en sorte qu'un créneau soit lié à un cours !*

Il faut mettre une association entre la classe `Cours` du package `personnel` et la classe `Creneau` du package `planning`. Si nous souhaitons pourvoir identifier le cours ayant lieu dans une salle pour un jour et un créneau donnés, il faut que cette association soit navigable de la classe `Creneau` vers la classe `Cours` et que le package `planning` importe le package `personnel`. C'est ce qu'illustre la figure 11.

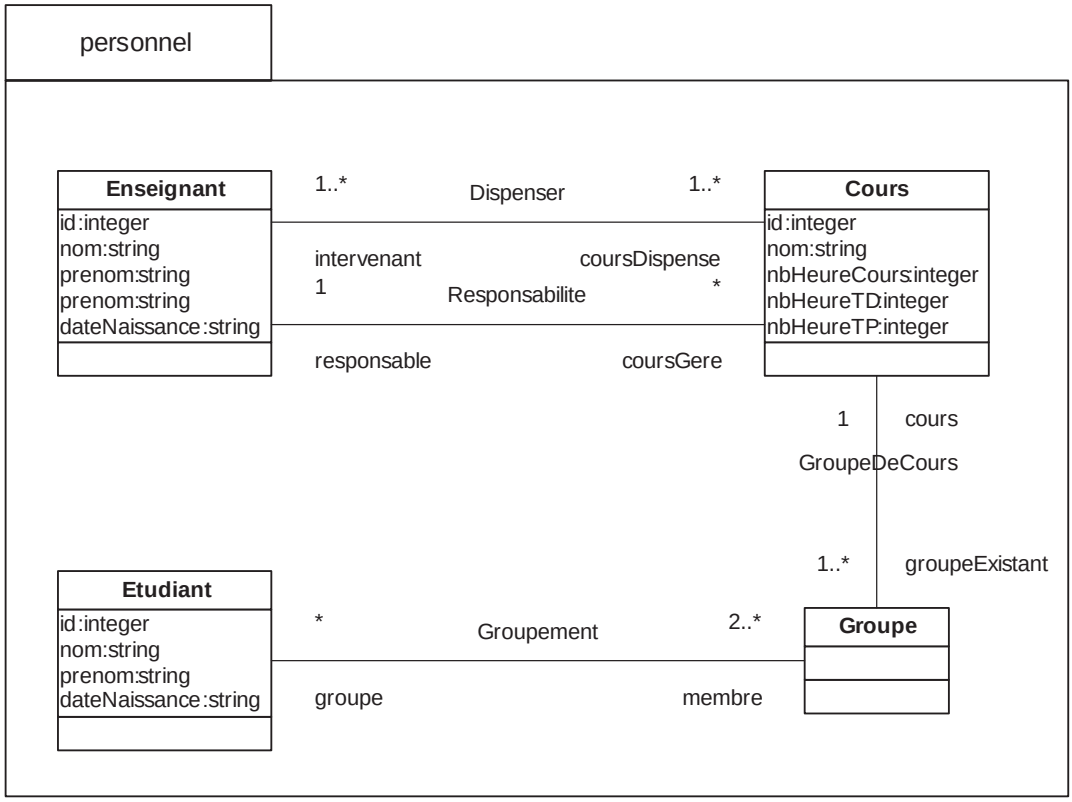
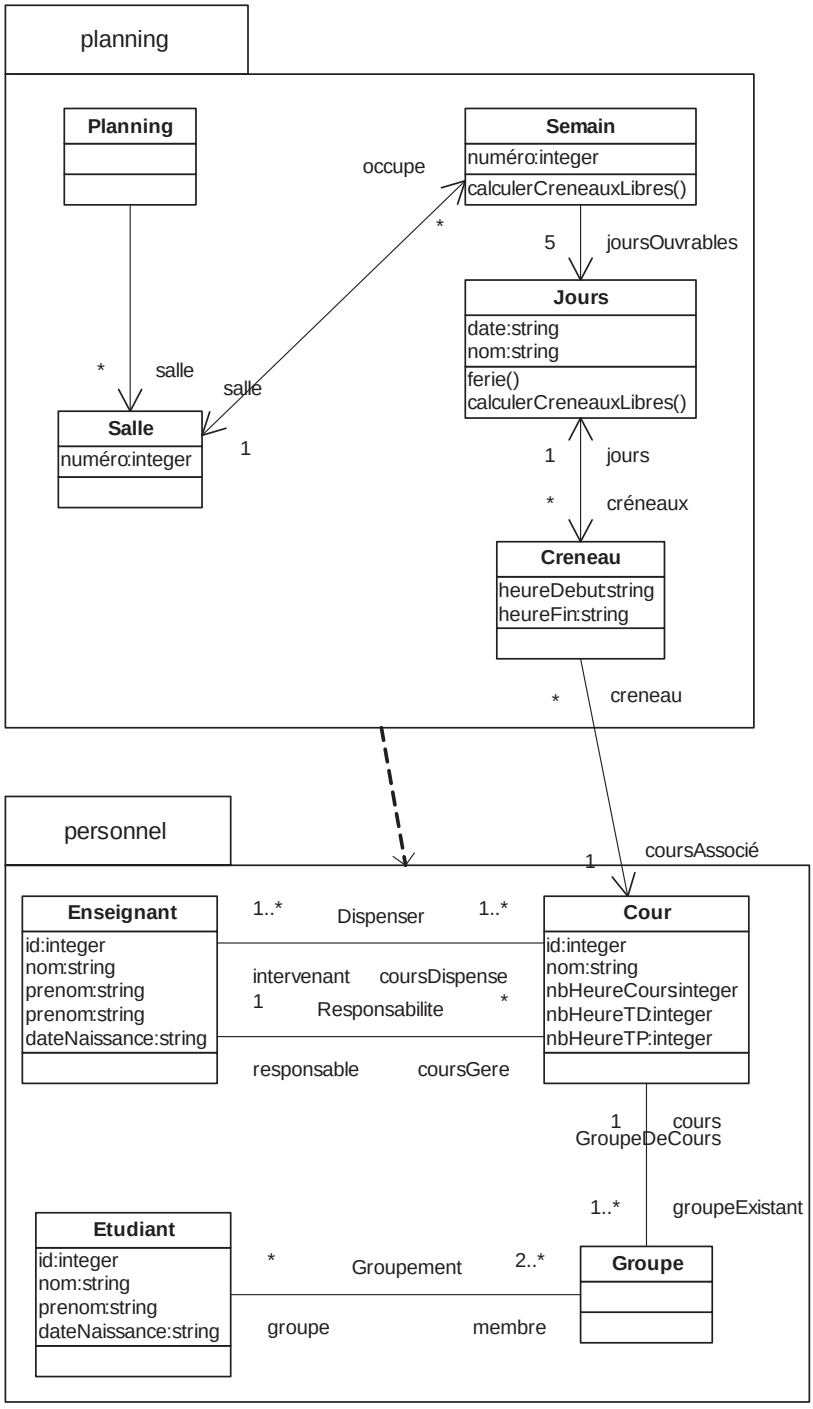


Figure 10
Package personnel

Figure 11
Packages planning *et* personnel



TD3. Reverse Engineering

Les opérations de Reverse Engineering présentées dans ce TD portent sur le code Java de l'application `MyAssistant` donné au TD1. Nous appliquons les règles de correspondance Java vers UML décrites au chapitre 3.

25. *Effectuez le Reverse Engineering de la classe* `Adresse`.

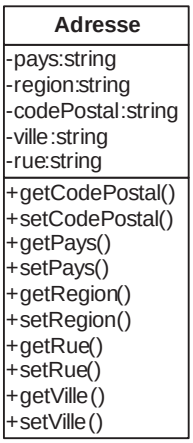
Les informations se trouvent dans le code de la classe `Adresse` :

- Règle 1 : à la classe Java `Adresse` doit correspondre une classe UML `Adresse`.
- Règle 3 : à tout attribut de la classe Java `Adresse` doit correspondre une propriété de même nom dans la classe UML associée. Le type Java de tous les attributs de la classe `Adresse` étant string, le type des propriétés associées sera le type UML `string`. Il y a cinq propriétés : `pays`, `region`, `codePostal`, `ville` et `rue`.
- Règle 4 : à toute opération de la classe Java `Adresse` doit correspondre une opération de même nom appartenant à la classe UML correspondante. La classe Java `Adresse` contient deux opérations. Il s'agit des opérations de lecture de la valeur d'un attribut (`public typeAttribut getNomAttribut()`) et des opérations d'affectation d'une valeur à un attribut (`public void setNomAttribut(typeAttribut nomParamètre)`). Nous retrouvons donc les opérations équivalentes dans la classe UML. Les opérations de lecture d'un attribut sont traduites par une opération ayant un paramètre de sortie du type de l'attribut et aucun paramètre d'entrée (`getNomAttribut():typeAttribut`). Les opérations d'affectation d'une valeur à un attribut sont traduites par une opération avec un paramètre d'entrée du même type que l'attribut et aucun paramètre de sortie (`setNomAttribut(In nomParamètre:typeAttribut`). Les opérations Java étant toutes publiques, les opérations UML correspondantes le seront toutes aussi.

Seules ces trois règles s'appliquent au Reverse Engineering de la classe `Adresse`. Il ne faut pas oublier d'attacher à chacune des opérations une note contenant le code de traitement de l'opération Java associée (règle 9).

Dans la représentation graphique de la classe qui est donnée à la figure 12, les opérations sont précédées d'un +, et leur paramètres sont masqués. Nous avons aussi choisi de masquer les notes attachées aux opérations.

Figure 12
Classe Adresse



26. *Effectuez le Reverse Engineering de la classe* `Personne`.

Les informations se trouvent dans le code de la classe `Personne`. La traduction de la majorité des attributs et opérations Java ne pose pas de problème particulier et se fait de façon similaire à ce que nous avons fait pour la classe `Adresse`.

Pour les opérations, la seule différence vient de l'opération `toString()`, qui ne modifie pas un attribut ni ne retourne sa valeur. Elle retourne en fait une chaîne de caractères composée de la valeur de deux attributs. Sa traduction ne pose pas de problème particulier : nous ajoutons à la classe `Personne` du modèle UML l'opération `+toString():string`. Il ne faut pas oublier d'attacher à chacune des opérations une note contenant le code de traitement de l'opération Java associée.

Pour les attributs, la seule différence vient de l'attribut `adresse` de type `Adresse`. Nous appliquons alors la règle 3 modifiée, qui nous dit de ne pas créer de propriété `Adresse` mais de créer une association navigable entre la classe `Personne` et la classe `Adresse`. Le nom de rôle associé à la classe `Adresse` est le même que celui de l'attribut Java. Il s'agit donc ici de `adresse`. Une personne ayant une seule adresse, nous précisons sur l'association UML qu'à une `Personne` est associée 0 à 1 adresse (nous acceptons d'avoir une propriété dont la valeur n'est pas définie).

La figure 13 représente l'association entre les classes `Personne` et `Adresse`.

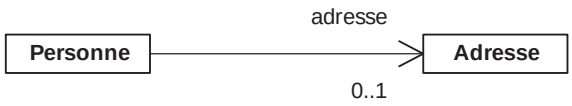


Figure 13
Association entre les classes Personne *et* Adresse

27. *Effectuez le Reverse Engineering de la classe* `Repertoire`.

Les informations se trouvent dans le code de la classe `Repertoire`. Cette classe ne possède qu'un attribut de type `ArrayList`, qui est une classe Java. Comme pour la question précédente, nous appliquons la règle 3 modifiée. Nous ne créons donc pas de propriété dans la classe `Repertoire` mais créons une classe UML `ArrayList` et une association navigable entre les classes UML `Repertoire` et `ArrayList`. Le nom de rôle associé à la classe `ArrayList` est le nom de l'attribut Java, ici `personnes`, et la multiplicité associée à cette classe est 0..1.

La classe Java contient cinq opérations, dont nous retrouvons les opérations correspondantes dans la classe UML `Repertoire`. Il ne faut pas oublier d'ajouter pour chacune d'elles une note contenant le code Java.

Voici la liste de ces opérations :

```
+ajouterPersonne(in p:Personne)
+supprimerPersonne(in p:Personne)
+rechercherPersonnesParNome(in nom:string):
+listerPersonnes():Personnes[*]
+Repertoire():Personnes[*]
```

Les deux dernières opérations retournent un tableau de personnes dont la taille n'est pas précisée.

La figure 14 représente l'association entre les classes `Repertoire` et `ArrayList`.

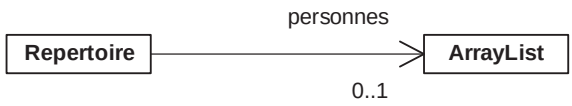


Figure 14
Association entre les classes Repertoire *et* ArrayList

28. *Pourquoi n'y a-t-il pas d'association entre la classe* `Repertoire` *et la classe* `Personne` *alors qu'un répertoire contient des personnes ?*

La déclaration des attributs de la classe Java `Repertoire` ne mentionne aucun attribut de type `Personne`, mais mentionne un attribut de type `ArrayList`. Le lien entre le répertoire et les personnes se fait grâce au lien qui existe entre les classes Java `ArrayList` et `Personne`. Ces classes sont liées par la classe Java `Object`. Il existe une association entre les classes Java `ArrayList` et `Object` parce que, d'une part, un objet de type `ArrayList` est composé d'un tableau d'éléments de type `Object` et que, d'autre part, toute classe Java hérite de la classe `Object`. La classe `Personne` hérite donc de la classe `Object`.

Ces relations font qu'un répertoire peut contenir un tableau de personnes. Cette relation n'est pas reflétée par le modèle UML, car il ne contient pas la classe `Objet`. Ajouter cette classe au modèle UML nécessiterait d'établir les liens entre elle et toutes les autres classes, ce qui n'est guère envisageable.

29. *Comment modifier les règles du Reverse Engineering pour faire en sorte qu'une association soit établie entre la classe* `Repertoire` *et la classe* `Personne` *?*

Dans notre cas, nous pouvons établir cette association, puisque l'attribut de type `ArrayList` de la classe `Repertoire` ne contient que des éléments de type `Personne`. Cet attribut peut donc être représenté par une association entre les classes `Repertoire` et `Personne`. Cela n'est cependant pas toujours possible, car un objet de type `ArrayList` contient des éléments de type `Object`. Puisque toutes les classes héritent de la classe `Object`, un objet de type `ArrayList` peut théoriquement contenir des éléments de n'importe quel type, et ils ne sont pas tous obligatoirement de même type. L'association entre la classe ayant une propriété de type `ArrayList` et la classe représentant le type des objets contenus dans le tableau dynamique (`ArrayList`) n'est donc pas toujours possible.

Nous dirons donc que si, dans Java, un attribut d'une classe `A` est de type `ArrayList` et que le code montre que tous les objets ajoutés au tableau dynamique sont de même type (représenté par une classe `B`), nous pouvons mettre une association de la classe `A` vers la classe `B`.

Attention : les associations peuvent se déduire des types des attributs, mais non des types des paramètres des opérations de la classe. Si une opération de la classe `A` a un paramètre de type `B`, rien ne dit que ce paramètre est une information accessible directement depuis la classe `A`. Il peut être le résultat d'une opération appelée par la classe `A`. Il n'y a donc pas lieu d'associer les classes `A` et `B`.

30. *Effectuez le Reverse Engineering de la classe* `UIPersonne`*.*

Les informations se trouvent dans le code de la classe `UIPersonne`. Nous constatons tout d'abord que la classe `UIPersonne` hérite de la classe `JPanel`, relation qui doit apparaître dans le modèle UML. Nous ajoutons donc une classe `JPanel` dont hérite la classe `UIPersonne`. La classe `UIPersonne` contient onze attributs, un de type `Personne` et les dix autres de type `JtextField`. L'attribut de type `Personne` est représenté par une association navigable entre la classe `UIPersonne` et la classe `Personne`. Le nom de rôle associé à la classe `Personne` est `personne`, et la multiplicité associée est 0..1.

Pour les autres attributs, nous ajoutons au modèle la classe `JFextField`, et nous créons une association navigable entre la classe `UIPersonne` et la classe `JTextField` pour chacun des attributs. Pour chacune de ces associations, le rôle associé à la classe `JTextField` correspond au nom de l'attribut Java, et la multiplicité associée est 0..1.

Nous ajoutons à la classe UML les cinq opérations suivantes, auxquelles nous ajoutons en note le code Java associé :

```
+UIPersonne()
+UIPersonne(in p:Personne)
+getPersonne():Personne
+setPersonne(in personne:Personne)
+init()
```

Il est important de noter que les règles de Reverse Engineering que nous avons définies ne permettent pas de prendre en considération la classe anonyme créée par l'opération `init`.

La figure 15 représente les liens (héritage ou association) entre la classe `UIPersonne` et les classes `JPanel`, `Personne` et `JTextField`.

31. *Comment introduire les classes Java dans le modèle UML ? À quoi cela sert-il ?*

Il faut appliquer une opération de Reverse Engineering sur le code de l'API Java qui se trouve dans le fichier **rt.jar.** Il faut donc disposer des sources de l'API et avoir des règles de Reverse Engineering s'appliquant à ce type de source. Il n'est pas forcément judicieux de traiter de la même manière le Reverse Engineering d'une application complète et celui d'une archive (fichier **.jar** en Java) regroupant un ensemble de classes et les ressources associées.

En fait, il n'est pas intéressant d'introduire dans le modèle UML toutes les informations qu'il est possible d'obtenir sur l'API Java (telles que les liens entre les classes de l'API). L'objectif est uniquement d'introduire les classes de l'API apparaissant directement dans le code afin de pouvoir les référencer et les lier aux classes de l'application. Ces liens sont indispensables pour pouvoir réaliser la génération de code.

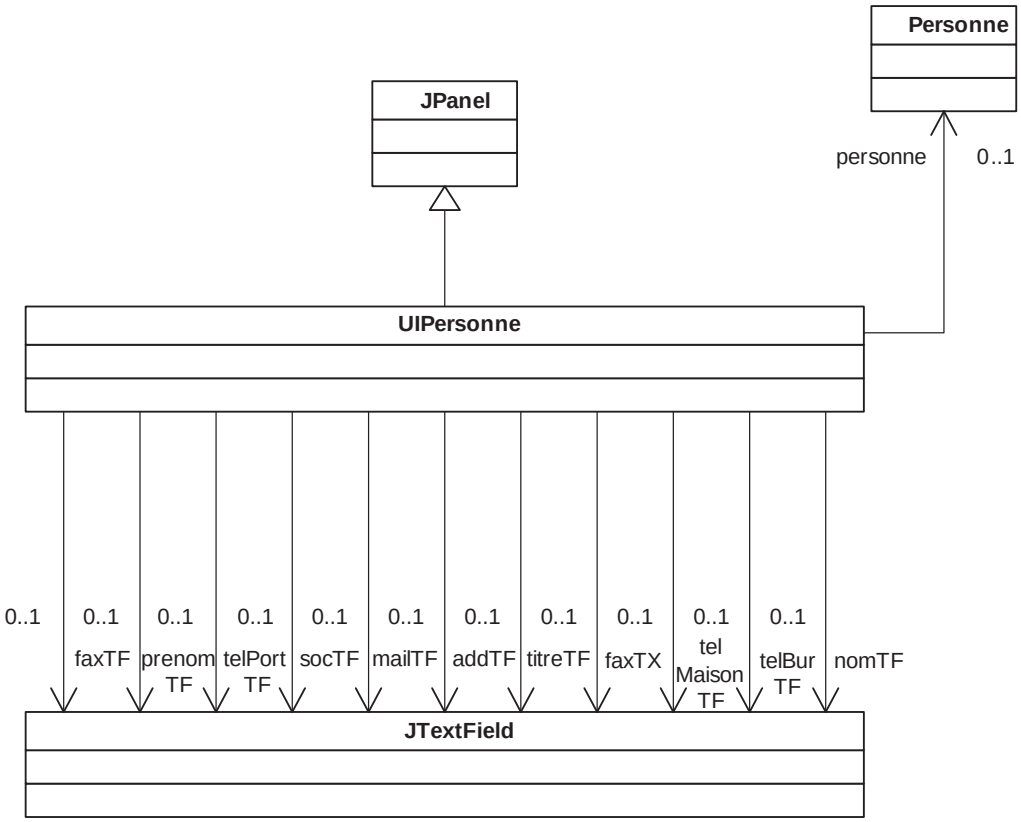


Figure 15

Associations entre UIPersonne *et* JTextField

32. *Est-il plus facile de comprendre une application après en avoir effectué le Reverse Engineering ?*

L'opération de Reverse Engineering ne facilite pas la compréhension d'une application. Il ne s'agit que de représenter graphiquement le code. Les diagrammes obtenus ne sont que la vue structurelle au niveau d'abstraction le plus bas de l'application. Pour comprendre facilement l'application, il faut produire les diagrammes associés à chacune des vues pour chacun des niveaux d'abstraction.

33. *Les informations obtenues après Reverse Engineering sont-elles plus abstraites que le code Java ?*

Non, les informations sont équivalentes. Le modèle obtenu après Reverse Engineering ne représente que les informations contenues dans le code. La preuve en est que notre modèle contient des classes Java.

34. *Le modèle obtenu par Reverse Engineering contient-il plus de diversité que le code ?*

Si, comme nous le préconisons, le Reverse Engineering produit plusieurs diagrammes de classes, le modèle obtenu offre un peu plus de diversité que le code. Cependant, il nous permet uniquement d'obtenir la vue structurelle de l'application.

Une analyse plus poussée du code pourrait nous permettre d'obtenir des informations sur le comportement et les fonctionnalités de l'application. Mais la mise en place de cette analyse rendrait l'opération de Reverse Engineering beaucoup plus délicate, car l'identification des nouveaux diagrammes à produire et des classes à faire apparaître dans ces diagrammes n'est pas évidente. Il est difficile d'automatiser ces opérations, dont le résultat dépend beaucoup de l'expérience de celui qui les réalise. Les règles que nous donnons pour le Reverse Engineering sont facilement automatisables et doivent donc toujours produire le même résultat pour un code donné.

35. *Si vous aviez un modèle UML et le code Java correspondant, comment pourriez-vous savoir si le modèle UML a été construit à partir d'un Reverse Engineering ?*

Comme nous venons de le voir, l'application de nos règles de Reverse Engineering produit des résultats déterministes. Pour un code donné, pour un ensemble de règles donné, le résultat du Reverse Engineering sera toujours le même, alors qu'un être humain ne ferait pas toujours les mêmes choix et ajusterait ses décisions en fonction de divers facteurs. Dans notre cas, nous pouvons dire que, si le modèle contient, par exemple, des liens avec la classe `ArrayList` ou si toutes les associations sont navigables à l'une de leurs extrémités et que les multiplicités soient 0..1, alors le modèle a été obtenu par Reverse Engineering.

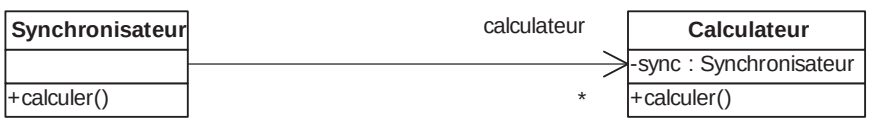
De façon générale, plus les mécanismes de Reverse Engineering sont « simples » à mettre en œuvre, plus ils sont identifiables, puisqu'ils n'autorisent pas de variété dans les traitements réalisés. Cependant, il est parfois possible de paramétrer l'opération de Reverse Engineering et donc de diversifier les traitements en fonction des particularités du code considéré (application complète, API Java, etc.).

TD4. Rétroconception et patrons de conception

La figure 16 représente les relations entre les classes `Synchronisateur` et `Calculateur`. Un calculateur permet d'effectuer des calculs. Etant donné que n'importe qui peut demander à un calculateur d'effectuer des calculs, la classe `Synchronisateur` a été construite pour réguler les calculs.

Les personnes qui souhaitent demander la réalisation d'un calcul doivent passer par le synchronisateur (*via* l'opération `calculer()`). Celui-ci distribue les calculs aux différents calculateurs avec lesquels il est lié (c'est lui qui appelle l'opération `calculer()` sur les calculateurs). Un calculateur connaît le synchronisateur auquel il est relié grâce à la propriété `sync` de type `Synchronisateur`. Sa valeur doit être déterminée lors de la création des objets de type `Calculateur`.

Figure 16
Classes Synchronisateur *et* Calculateur



36. *Exprimez en les justifiant les dépendances entre les classes* `Synchronisateur` *et* `Calculateur`*.*

L'association navigable de la classe `Synchronisateur` vers la classe `Calculateur` établit une dépendance de la classe `Synchronisateur` vers la classe `Calculateur`. La propriété `sync` de type `Synchronisateur` de la classe `Calculateur` établit une dépendance de la classe `Calculateur` vers la classe `Synchronisateur`.

37. *Nous souhaitons que les classes* `Synchronisateur` *et* `Calculateur` *soient dans deux packages différents. Proposez une solution.*

Comme nous venons de le voir à la question précédente, il y a un cycle de dépendances entre les classes `Calculateur` et `Synchronisateur`. Nous ne pouvons nous contenter de les mettre dans deux packages différents, car il faudrait alors établir une dépendance mutuelle entre ces deux packages. Nous devons donc « déporter » une des causes du cycle de dépendances hors de sa classe d'origine.

Nous ne considérons que la classe `Calculateur` et choisissons de déporter l'opération `calculer()` de la classe `Calculateur`, qui est à l'origine de la dépendance de la classe `Synchronisateur` vers la classe `Calculateur`. L'association est là pour permettre à un synchronisateur d'identifier les calculateurs qui dépendent de lui et de pouvoir appeler leur opération `calculer()`. Nous créons donc une classe `CalculateurSup`, dont hérite la classe `Calculateur` et qui contient l'opération `calculer()`. L'association est de la sorte elle aussi déportée de la classe `Calculateur` vers la classe `CalculateurSup`.

Nous pouvons mettre dans un même package les classes `Synchronisateur` et `CalculateurSup` et dans un autre la classe `Calculateur`. Les dépendances sont alors internes à un package (association entre les classes `Synchronisateur` et `CalculateurSup`) ou du package contenant la classe `Calculateur` vers l'autre package (dépendances dues à la relation d'héritage de la classe `CalculateurSup` par la classe `Calculateur` et à la propriété de type `Synchronisateur` de la classe `Calculateur`).

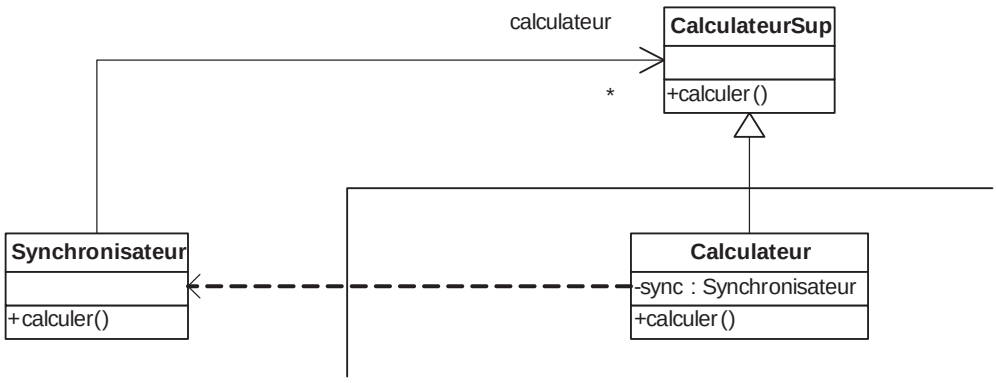


Figure 17

Suppression du cycle de dépendances

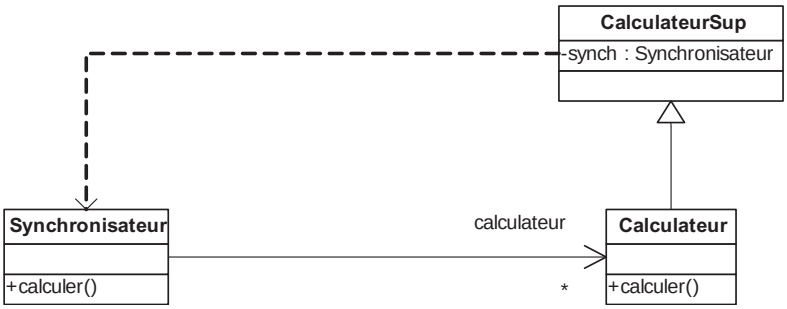
La figure 17 représente l'ensemble des liens existant entre les classes `Synchronisateur`, `CalculateurSup` et `Calculateur`. Nous avons matérialisé par un trait la séparation en deux packages. Il est à noter que l'opération `calculer()` est bien déclarée dans la

classe `CalculateurSup`, alors qu'elle n'est que répétée dans la classe `Calculateur`, ce qui n'est pas obligatoire.

Nous pourrions aussi envisager de résoudre le problème en déportant la propriété `sync` de la classe `Calculateur`, mais cela ne ferait qu'« agrandir » le cycle de dépendances. La dépendance de la classe `Synchronisateur` vers la classe `Calculateur` ne peut être déportée, car elle est liée à l'opération `calculateur()`, qui reste dans la classe `Calculateur`. La classe `Calculateur` hérite de la classe `CalculateurSup`, qui, elle même, dépend de la classe `Synchronisateur` en raison de sa propriété.

La figure 18 représente cette mauvaise solution en mettant en évidence le cycle de dépendances.

Figure 18
Échec de la suppression du cycle de dépendances



38. *Nous souhaitons ajouter à la classe* `Synchronisateur` *une opération* `ajouterCalculateur()` *qui permette d'assigner un calculateur à un synchronisateur, l'identité du calculateur étant un paramètre d'entrée de l'opération. Définissez cette opération.*

La seule difficulté est de faire attention à ne pas créer un cycle de dépendances en ajoutant cette opération. Les dépendances nouvelles créées lors de l'ajout d'une opération proviennent des paramètres de l'opération. Il ne faut absolument pas qu'une dépendance soit créée de la classe `Synchronisateur` vers la classe `Calculateur`. Le type du paramètre de l'opération doit donc être la classe `CalculateurSup`. L'opération à ajouter dans la classe `Synchronisateur` est alors `declarerCalculateur(in recepteur:CalculateurSup)`.

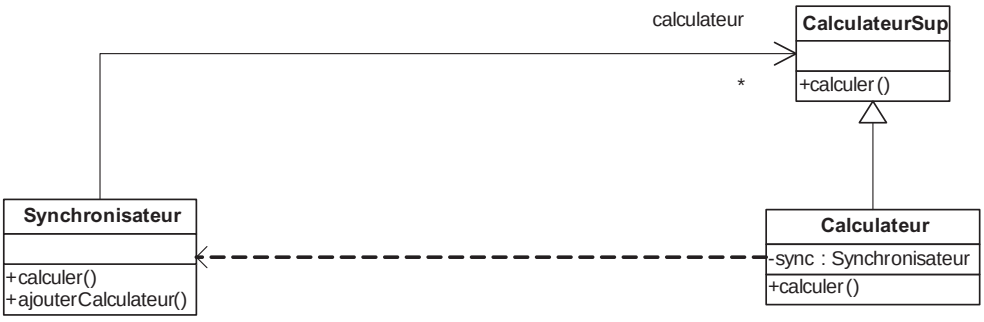


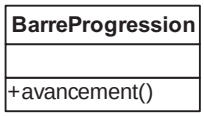
Figure 19
Ajout de l'opération ajouterCalculateur*()*

La figure 19 représente les trois classes `Synchronisateur`, `CalculateurSup` et `Calculateur`, ainsi que les liens établis entre ces classes.

39. *Nous souhaitons maintenant définir une classe représentant une barre de progression. Cette barre affiche l'état d'avancement du calcul (en pourcentage). Une barre de progression reçoit des messages d'un calculateur qui l'informe que l'état d'avancement du calcul a changé. Définissez cette classe.*

La figure 20 représente la classe `BarreProgression`, qui contient l'opération `avancement()`.

Figure 20
Classe BarreProgression



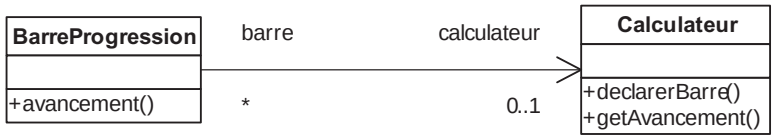
40. *Tout comme le synchronisateur, une barre de progression doit se déclarer auprès d'un calculateur. De plus, le calculateur doit offrir une opération permettant de connaître le pourcentage d'avancement du calcul. Définissez les associations et opérations nécessaires.*

Il faut modifier la classe `Calculateur`. La barre de progression devant se déclarer auprès du calculateur, c'est ce dernier qui contient l'opération `declarerBarre()`. Pour que le calculateur puisse identifier la barre qui se déclare, il faut que son identité soit passée en paramètre de l'opération.

Pour qu'une barre de progression puisse avoir accès à l'état d'avancement du calcul, il faut créer une association allant de la classe `BarreProgression` vers la classe `Calculateur` et l'opération `getAvancement():integer` dans la classe `Calculateur`. La multiplicité de l'association est 0..1 à l'extrémité du calculateur et 0..* à l'extrémité de la barre de progression. Une barre de progression est associée à au plus un calculateur, et nous faisons l'hypothèse qu'un calculateur est associé à plusieurs barres de progression.

La figure 21 représente les associations et opérations ajoutées aux classes `BarreProgression` et `Calculateur`.

Figure 21
Classes BarreProgression *et* Calculateur *après modification*



41. *Appliquez le patron de conception Observer, et faites en sorte que ces deux classes soient dans deux packages différents.*

Le diagramme donné en réponse à la question précédente montre qu'il y a un cycle de dépendances entre les classes `BarreProgression` et `Calculateur`. Il y a une association de la classe `BarreProgession` vers la classe `Calculateur`, et le paramètre de l'opération `declarerBarre()` de la classe `Calculateur` est de type `BarreProgression`. Nous devons donc « casser » ce cycle afin de pouvoir mettre les classes dans des packages différents.

Nous souhaitons le faire en appliquant le patron de conception Observer, ce qui est pertinent puisque le problème résolu par ce patron de conception est : « Créer un lien entre un objet source et plusieurs objets cibles permettant de notifier les objets cibles lorsque l'état de l'objet source change. De plus, il faut pouvoir dynamiquement lier à (ou délier de) l'objet source autant d'objets cibles que nous le voulons. »

Dans notre cas, l'objet `Observer` est la barre de progression et l'objet `Subject` le calculateur, puisque notre problème est d'informer la barre de progression des avancements du calculateur.

La classe `Observer` du patron de conception correspond à une abstraction de la classe `BarreProgression` dont cette dernière hérite. C'est elle qui doit contenir la méthode par laquelle la barre de progression est informée des avancements du calculateur. La classe `Subject` du patron de conception correspond à une abstraction de la classe `Calculateur` dont cette dernière hérite. Dans notre cas, cette classe contient l'équivalent de l'opération `attach(in obs:Observer)`, qui, dans notre cas, est l'opération `declarerBarre(in barre:Progression)` ainsi que l'opération de notification à tous les observateurs (`notify()`). À l'aide de ce patron de conception, il est possible de faire une découpe en packages séparant les classes `BarreProgression` et `Calculateur`.

La figure 22 représente l'application du patron de conception Observer sur nos classes.

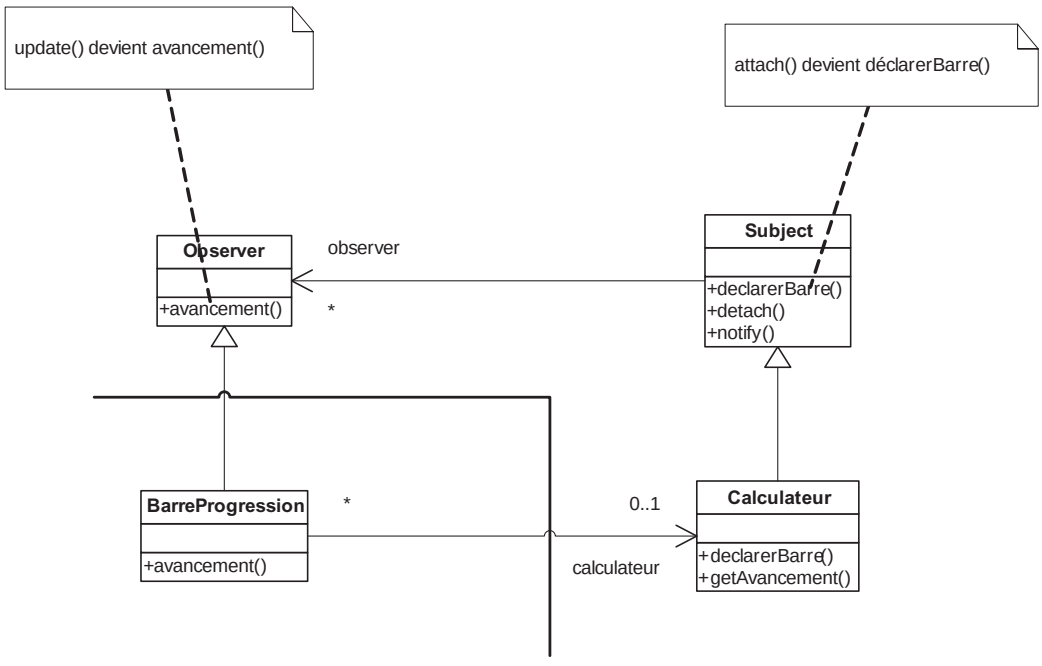


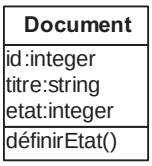
Figure 22

Application du patron Observer

TD5. Génération de code

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

Figure 23
Classe Document



D'après les règles de correspondance UML vers Java que nous avons définies, le code suivant est obtenu :

```
public class Document
{
    public int id;
    public String titre;
    public int etat;

    public void définirEtat(
        int etat)
    {
    }
}
```

Remarquons que les règles suivantes ont été appliquées :

- règle n° 1 pour la classe ;
- règle n° 3 pour toutes les propriétés de la classe ;
- règle n° 4 pour l'opération de la classe.

43. *Écrivez le code généré à partir de la classe* `Bibliothèque` *illustrée à la figure 24.*

Figure 24
Classes Bibliothèque *et* Document

D'après les règles de correspondance UML vers Java que nous avons définies, le code suivant est obtenu :

```
public class Bibliothèque
{
    public java.util.ArrayList doc = new java.util.ArrayList();
    public void ajouterDocument()
    {
    }

    public void listerDocument()
```

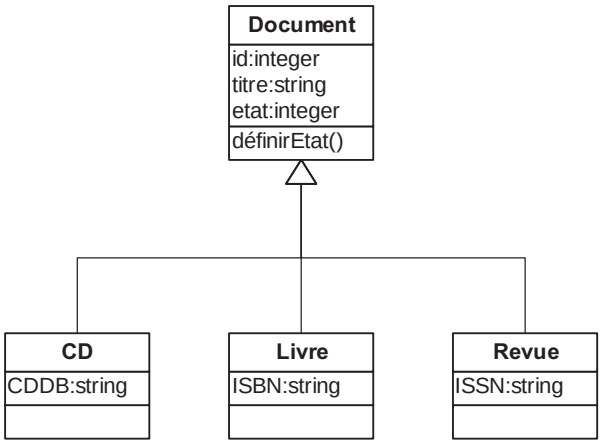
```
    {
    }
}
```

Remarquons que les règles suivantes ont été appliquées :

- règle n° 1 pour la classe ;
- règle n° 4 pour les opérations de la classe ;
- règle n° 5 (deuxième version) pour l'association vers la classe `Document`.

44. *Écrivez le code généré à partir des classes* `Livre`, `CD` *et* `Revue` *représentées à la figure 25.*

Figure 25
Classes CD, Livre *et* Revue



D'après les règles de correspondance UML vers Java que nous avons définies, et plus particulièrement grâce à la règle n° 6, le code suivant est obtenu :

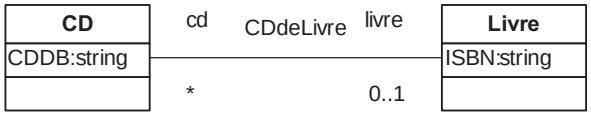
```
public class CD extends Document {
    Public String CDDB;
}

public class Livre extends Document {
    Public String ISBN;
}

public class Revue extends Document {
    Public String ISSN;
}
```

45. *Écrivez le code généré à partir de l'association* `CDdeLivre` *représentée à la figure 26 après avoir défini les règles de génération de code que vous comptez utiliser.*

Figure 26
Association CDdeLivre



Aucune des règles de correspondance UML vers Java que nous avons rappelées au chapitre 5 ne permet de traiter les associations non navigables. Nous pouvons proposer la règle suivante :

Pour toute association non navigable, créer une classe Java. Le nom de la classe Java doit correspondre au nom de l'association. Pour chacunr des deux extrémités de l'association, créer un attribut dans la classe Java. Le nom de l'attribut doit correspondre au nom du crochet. Le type de l'attribut doit correspondre à la correspondance Java de la classe UML attachée à l'extrémité. Si l'extrémité spécifie que le nombre maximal d'objets pouvant être reliés est supérieur à 1, l'attribut Java est un tableau.

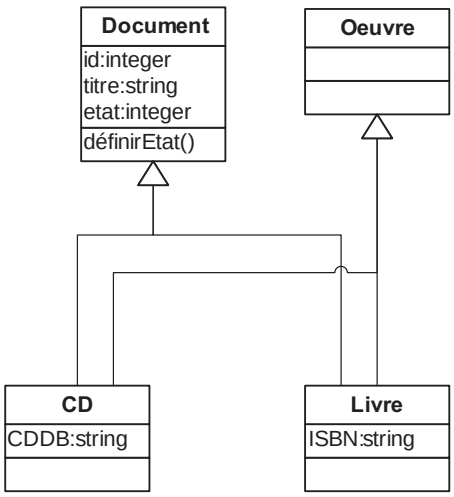
En appliquant cette règle à l'association `CDdeLivre`, nous obtenons le code suivant :

```
public class CDdeLivre {
    CD[] cd ;
    Livre livre ;
}
```

Les objets Java instances de cette classe permettront ainsi de représenter des liens entre un livre et des CD.

46. *Écrivez le code généré à partir des classes représentées à la figure 27 après avoir défini les règles de génération de code que vous comptez utiliser.*

Figure 27
Héritage multiple



Aucune des règles de correspondance UML vers Java rappelées au chapitre 5 ne permet de traiter l'héritage multiple. Soulignons que la transformation d'un modèle à héritage multiple vers un modèle à héritage simple est un problème complexe très connu, mais sans solution facile à mettre en place. Nous pouvons cependant proposer la règle suivante, qui ne s'applique qu'à certains modèles, dont celui de notre question :

Pour tout modèle à héritage multiple dont seule une classe héritée possède des propriétés (les autres classes héritées n'en possèdent pas) et dont toutes les classes

héritées n'héritent pas d'autres classes, créer une classe Java pour la classe héritée qui possède des propriétés, créer une interface Java pour les autres classes héritées, créer une classe Java pour la classe qui hérite et faire en sorte que celle-ci étende la classe Java créée et réalise les interfaces Java créées.

En appliquant cette règle à notre cas, nous obtenons le code ci-dessous.

Le code de la classe `Document` est le même que celui de la question 42 :

```
public interface Oeuvre {
}
```

Le code des classes `CD` et `Livre` change afin de réaliser l'interface `Oeuvre` :

```
public CD extends Document realize Œuvre {
}

public Livre extends Document realize Œuvre {
}
```

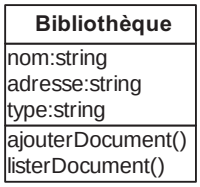
Un mécanisme d'update permet de faire remonter les modifications du code Java dans le modèle UML avec lequel il est déjà synchronisé. Par exemple, si nous considérons que le modèle UML et le code Java de la classe `Bibliothèque` sont synchronisés depuis la question 43 et que nous ajoutions dans le code 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.

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.*

Ces trois attributs, qui n'apparaissaient pas dans le modèle, y sont ajoutés selon les règles de correspondance du Reverse Engineering. Nous obtenons la classe représentée à la figure 28.

Figure 28
Classe Bibliothèque

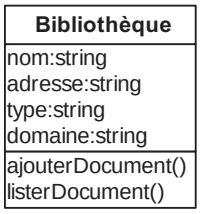


48. *Nous voulons maintenant, toujours dans le code Java, changer l'attribut* `type` *en attribut* `domaine`. *Pensez-vous qu'il soit possible, après un update, que les deux attributs* `type` *et* `domaine` *puissent être présents dans le modèle ? Si oui, à quoi est dû ce comportement bizarre ?*

À la question précédente, nous avons défini l'update comme une opération ne faisant qu'ajouter de nouveaux éléments au modèle UML. De ce fait, les éléments ne sont jamais supprimés du modèle UML. L'update ne sait donc pas faire la diffé-

rence entre l'ajout d'un nouvel attribut et la modification d'un attribut existant. Dés lors, il est tout à fait envisageable d'obtenir le modèle UML de la classe `Bibliothèque` représenté à la figure 29 avec les deux attributs `type` et `domaine`.

Figure 29
Classe Bibliothèque *après modification de la propriété* type *en* domaine



La synchronisation entre le modèle et le code n'est plus faite.

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

Ce nouveau mécanisme d'update doit être capable de faire la différence entre l'ajout d'un nouvel attribut et la modification d'un attribut existant.

Pour ce faire, l'update doit être capable de reconnaître chaque attribut Java et de vérifier qu'il n'existe pas une propriété correspondante dans le modèle UML. Si une propriété existe déjà, il faut la modifier. L'update doit donc lier les attributs Java aux propriétés UML pour chaque classe.

Ce lien ne peut se faire sur le nom des attributs et des propriétés. En effet, la modification du nom de l'attribut Java rendrait impossible la recherche de la propriété UML correspondante et entraînerait les mêmes problèmes de perte de synchronisation.

La solution classique à ce problème consiste, d'une part, à introduire dans le code Java des commentaires permettant d'identifier chaque attribut et, d'autre part, à lier ces identifiants aux identifiants des propriétés UML.

Le code suivant de la classe `Bibliothèque` illustre cette utilisation des commentaires :

```
public class Bibliothèque
{
    //attribut id1
    public String nom;

    //attribut id2
    public String adresse;

    //attribut id3
    public String type;

    public java.util.ArrayList doc = new java.util.ArrayList();
    public void ajouterDocument()
    {
    }

    public void listerDocument()
```

```
        {
        }
}
```

En liant ces identifiants aux identifiants des propriétés UML, il est possible de réaliser une correspondance entre chaque attribut Java et chaque propriété UML et ainsi d'effectuer les modifications des propriétés UML lorsque les attributs Java sont modifiés.

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.*

Ce mécanisme ressemble fortement à l'update, si ce n'est qu'il s'appuie sur l'opération de génération de code. Les identifiants des attributs Java doivent tout autant être utilisés afin de permettre la modification des propriétés UML et la modification des attributs Java correspondants.

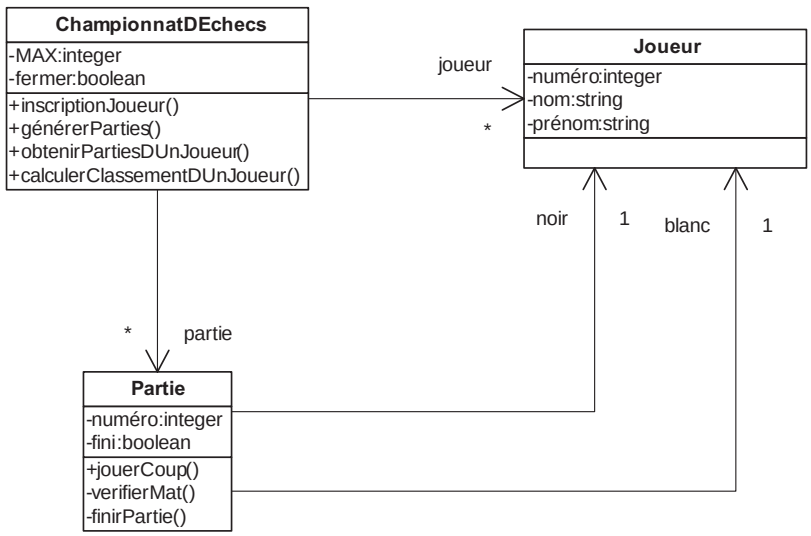
51. *Dans quelle approche de programmation par modélisation (Model Driven, Code Driven et Round Trip) ces mécanismes d'update sont-ils fondamentaux ?*

Ce mécanisme est réellement nécessaire pour l'approche Round Trip, qui permet la modification du modèle et du code à n'importe quel moment et doit donc assurer finement la synchronisation. Ce mécanisme n'est pas intéressant pour les approches Model Driven ou Code Driven, car ces deux approches n'utilisent l'opération de génération de code ou de Reverse Engineering qu'une fois.

TD6. Diagrammes de séquence

L'application `ChampionnatEchecs`, qui doit 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 30)*.

Figure 30
Classes de l'application ChampionnatEchecs



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.

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 non.

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.

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

Les administrateurs et les participants ne font pas partie de l'application, mais ils l'utilisent. Voilà pourquoi, il n'existe pas de classe `Administrateur` ni `Participant`. Il est très important de distinguer le participant de l'instance de la classe `Joueur`. L'instance de la classe `Joueur` est un objet qui fait partie de l'application et qui contient différentes informations sur un participant.

Pour faire apparaître les participants et les administrateurs dans les diagrammes de séquence, il faut utiliser des objets non typés.

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 30.*

Le diagramme de séquence solution représenté à la figure 31 contient six objets. L'objet `l'administrateur` n'est pas typé et représente l'administrateur. Les objets `MrBF` et `MrBS` ne sont pas typés non plus et représentent respectivement les deux participants au championnat. L'objet `umlOne` est instance de la classe `ChampionnatDEchecs` et représente le championnat. Les objets `bf` et `bs` sont instances de la classe `Joueur` et représentent les deux joueurs du championnat.

Le diagramme de séquence spécifie que `l'administrateur` commence par créer l'objet `umlOne` puis que `MrBF` et `MrBS` demandent à s'inscrire au championnat en donnant leur nom et leur prénom. En réponse aux deux demandes d'inscription, `umlOne` créé les deux objets `bf` et `bs` instances de la classe `Joueur` et retourne les numéros d'identification des joueurs.

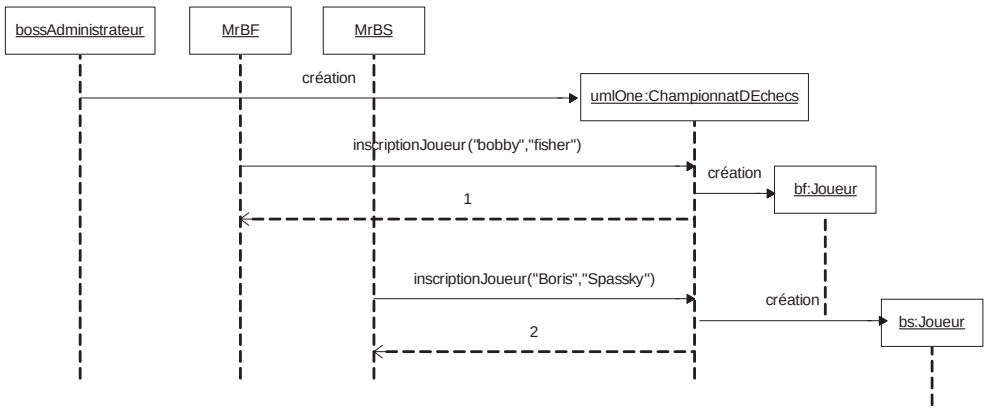


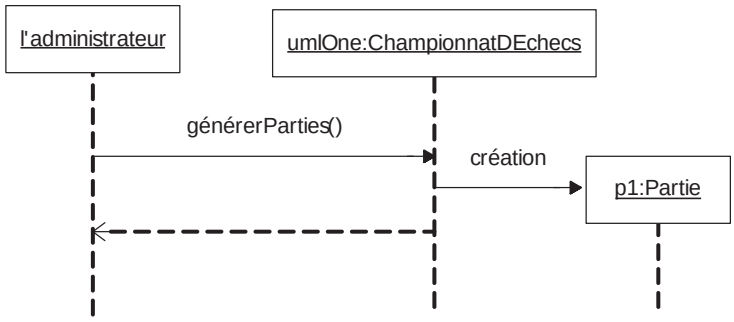
Figure 31

Diagramme de séquence représentant des inscriptions

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 30.*

Le diagramme de séquence solution représenté à la figure 32 contient trois objets. L'objet `l'administrateur` est le même qu'à la question 53. Il demande à l'objet `umlOne` de réaliser l'opération `générerPartie()`. Celui-ci, qui est aussi le même objet qu'à la question précédente, crée une seule partie, représentée par l'objet `p1` instance de la classe `Partie`, car le championnat ne contient que deux joueurs.

Figure 32
Diagramme de séquence représentant la génération des parties



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 30.*

Le diagramme de séquence solution représenté à la figure 33 contient trois objets. Les objets `MrBF` et `MrBS` sont les mêmes qu'à la question 53. L'objet `p1` est le même qu'à la question 54. Ce diagramme spécifie les différents coups joués par les deux participants et le fait que nous vérifions à chaque coup que la position n'est pas mat. Au dernier coup, la position est mat. La partie est donc terminée.

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

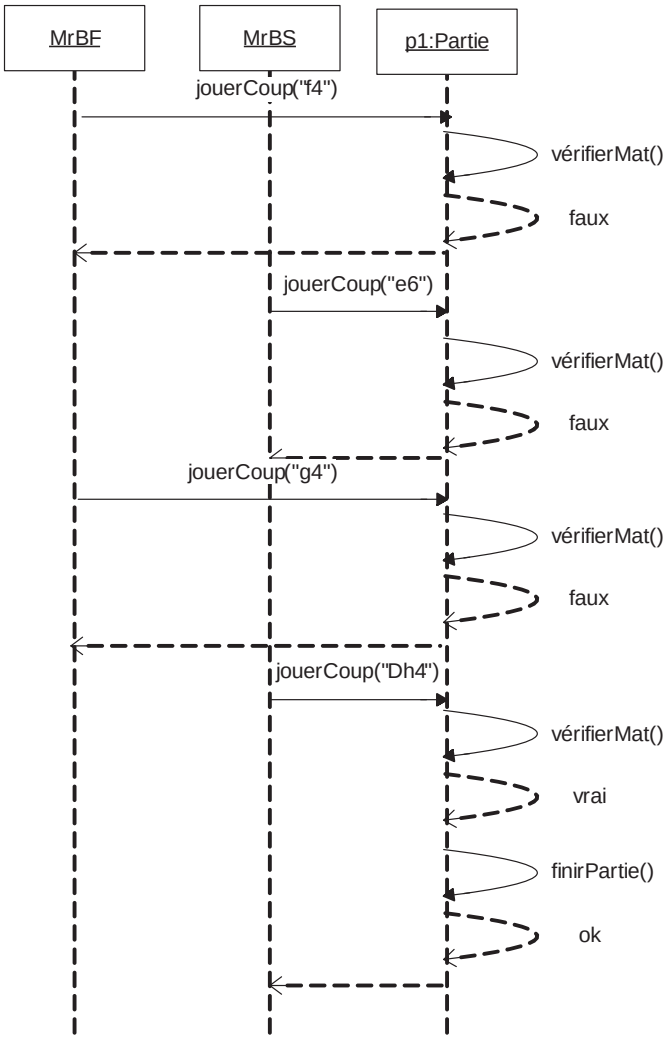
Comme nous l'avons vu en cours, il n'est pas réellement possible de générer le code d'une opération. Cela est particulièrement flagrant avec l'opération `jouerCoup`. Nous voyons bien qu'il n'est pas possible de réaliser tous les diagrammes de séquence correspondant à toutes les exécutions possibles de cette opération.

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

Lorsque nous disposons du code d'une application, il est possible de l'exécuter. De ce fait, il est possible de construire un diagramme de séquence représentant scrupuleusement cette exécution.

Par exemple, nous pourrions exécuter l'application de gestion de championnat d'échecs pour un championnat particulier et modéliser cette exécution dans un diagramme de séquence. Notons que cette fonctionnalité est par ailleurs souvent proposée par les outils du marché.

Figure 33
Diagramme de séquence représentant une partie



Il est important de noter que les diagrammes obtenus ne représentent qu'une exécution de l'application. L'ensemble des diagrammes de séquence obtenus à partir du code de l'opération ne peuvent donc être utilisés ultérieurement à des fins de génération automatique de code.

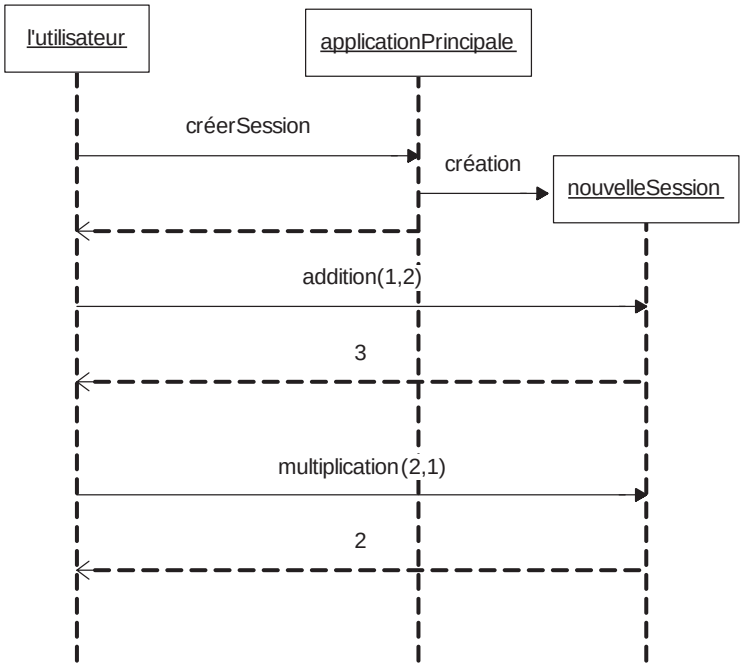
Une équipe de développement souhaite réaliser une application `Calculus` permettant à des utilisateurs d'effectuer des opérations arithmétiques simples sur des entiers : addition, soustraction, produit, division. Cette application a aussi une fonction mémoire, qui permet à l'utilisateur de stocker un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

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

L'objectif de cette question est d'illustrer le fait qu'il est possible de commencer à modéliser une application par l'élaboration de séquences plutôt que par l'élaboration de classes.

Le diagramme représenté à la figure 34 spécifie une demande de création de session puis la réalisation d'une addition et d'une multiplication.

Figure 34
Diagramme de séquence représentant une addition et une multiplication



Le diagramme représenté à la figure 35 spécifie une demande de création de session puis l'affectation de la mémoire à 2, puis l'ajout de 2 à la mémoire (la mémoire contient donc 4).

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

Les diagrammes de séquence que nous avons proposés à la question 58 nous permettent de proposer le diagramme de classes représenté à la figure 36.

Ainsi, l'objet `applicationPrincipale` est-il instance de la classe `ApplicationArtihmétique` et l'objet `nouvelleSession` est-il instance de la classe `Session`. En multipliant ainsi les diagrammes de séquence, nous pouvons obtenir un diagramme de classes beaucoup plus complet.

Figure 35
Diagramme de séquence représentant un ajout dans la mémoire

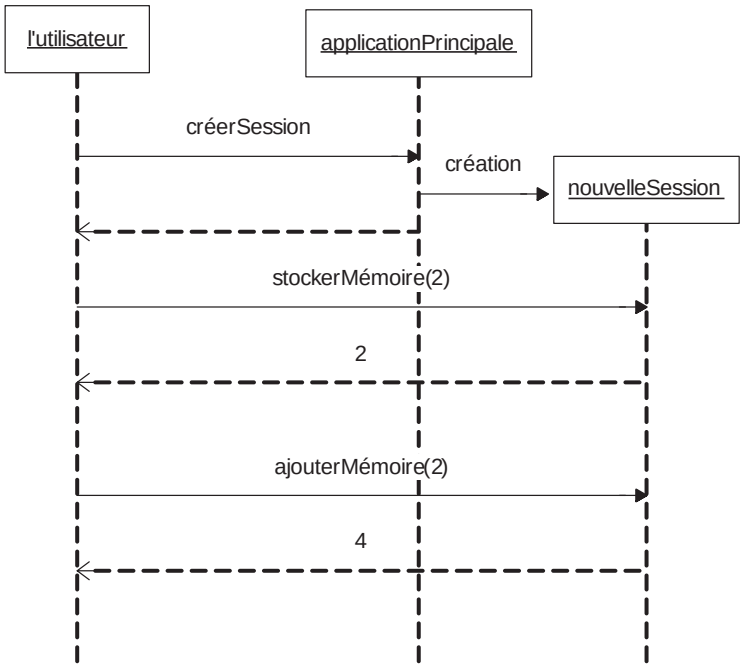
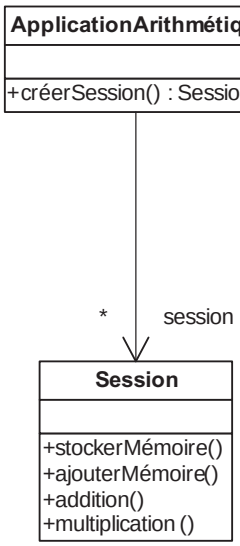


Figure 36
Classe ApplicationArithmétique



Nos diagrammes de séquence ne font pas apparaître les opérations `soustraction()`, `division()`, `soustraireMémoire()`, `diviserMémoire()`, `multiplierMémoire()`, etc., que l'application devrait logiquement offrir. Comme nous déduisons notre diagramme de classes des diagrammes de séquence, il est logique que ces opérations n'apparaissent pas non plus dans la classe `Session`.

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).

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

Une faute potentielle serait que l'opération `vérifierMat()` ne retourne pas vrai alors que la partie est réellement mat. Si tel était le cas, la partie ne serait pas finie, et les joueurs pourraient continuer à jouer leurs coups.

À l'inverse, une autre faute serait que l'opération `vérifierMat()` retourne vrai alors que la partie n'est pas mat. Si tel était le cas, la partie serait finie, et les joueurs ne pourraient plus continuer à jouer leurs coups.

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

Si nous nous concentrons sur la première faute que nous avons identifiée à la question 60, un cas de test abstrait visant à révéler cette faute serait de simuler le jeu d'une partie jusqu'à un mat. Le résultat attendu serait que la partie soit fermée à l'issue de la simulation.

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

Comme illustré à la figure 37, le diagramme de cas de test doit contenir le testeur (objet à gauche du diagramme). Celui-ci doit créer l'objet à tester (l'objet instance de la classe `Patie`), puis le testeur doit stimuler l'objet à tester. Dans notre cas, le testeur fait semblant de jouer une partie jusqu'au mat. Enfin, le diagramme de cas de test doit spécifier le résultat attendu. Dans notre cas, la partie doit être finie.

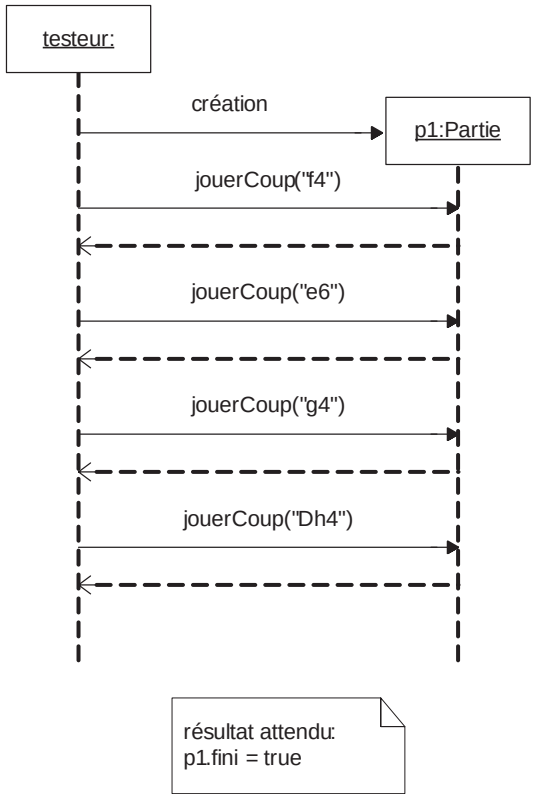
La suite de coups représentée dans le diagramme suivant correspond à une partie réelle amenant au mat du roi blanc en quatre coups.

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

En suivant les règles de correspondance décrites dans le cours, nous obtenons le code suivant :

```
public class Test extends TestCase{
    public void testExecutable() {
        p1 = new Partie();
        p1.jouerCoup("f4") ;
        p1.jouerCoup("e6") ;
        p1.jouerCoup("g4") ;
        p1.jouerCoup("Dh4") ;
        assertTrue(p1.fini) ;
    }
}.
```

Figure 37
Diagramme de séquence de test de la classe Partie



La dernière ligne de ce code permet de vérifier que la partie est bien terminée.

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

Absolument pas. Cela signifie simplement que, pour cette suite de coups (f4, e6, g4, Dh4), l'application ne révèle pas de faute. Une autre suite de coups pourrait révéler une faute qui indiquerait que l'application contient une défaillance.

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

Malheureusement, réaliser un grand nombre de cas de test n'offre pas plus de garantie sur la non-existence de défaillance dans une application. Cela se voit bien dans ce cas, puisqu'il faudrait réaliser un cas de test abstrait pour chaque partie d'échecs imaginable.

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).

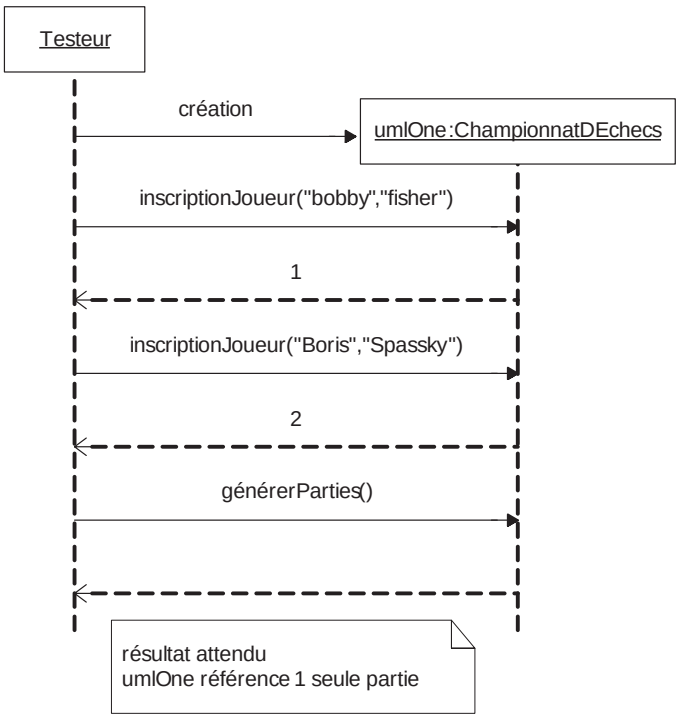
66. *Identifiez une faute qui pourrait intervenir lors de la création des parties d'un championnat. Définissez un cas de test abstrait visant à révéler cette faute, et construisez un diagramme de séquence de test modélisant ce cas de test abstrait.*

Une faute potentielle serait que toutes les parties du championnat ne soient pas créées ou qu'il y en ait plus que nécessaire. Il serait alors impossible à tous les joueurs de jouer leurs parties ou la fin du championnat ne serait jamais atteinte.

Un cas de test abstrait permettant de révéler cette faute serait de construire un championnat avec deux joueurs et de vérifier qu'après construction des parties, il existe bien exactement une seule partie.

Le diagramme représenté à la figure 38 spécifie ce cas de test abstrait. Le testeur crée l'instance de la classe `ChampionnatDEchecs`. Il simule ensuite l'inscription de deux joueurs puis demande la génération des parties. Le résultat attendu est que le championnat ne contienne qu'une seule partie.

Figure 38
Diagramme de séquence de test de la classe ChampionnatDEchecs



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) ?*

Il est en effet possible de démarrer le cas de test abstrait de la question 61 après le cas de test abstrait de la question 66. Pour ce faire, il faudrait modifier un peu le cas de test de la question 61 afin de préciser que le testeur n'a pas à créer la partie à tester mais peut l'obtenir puisqu'elle est liée au championnat déjà créé.

TD8. Plates-formes d'exécution

68. *Le diagramme de classes de l'agence de voyage représenté à la figure 39 correspond-t-il à un modèle conceptuel ou à un modèle physique ?*

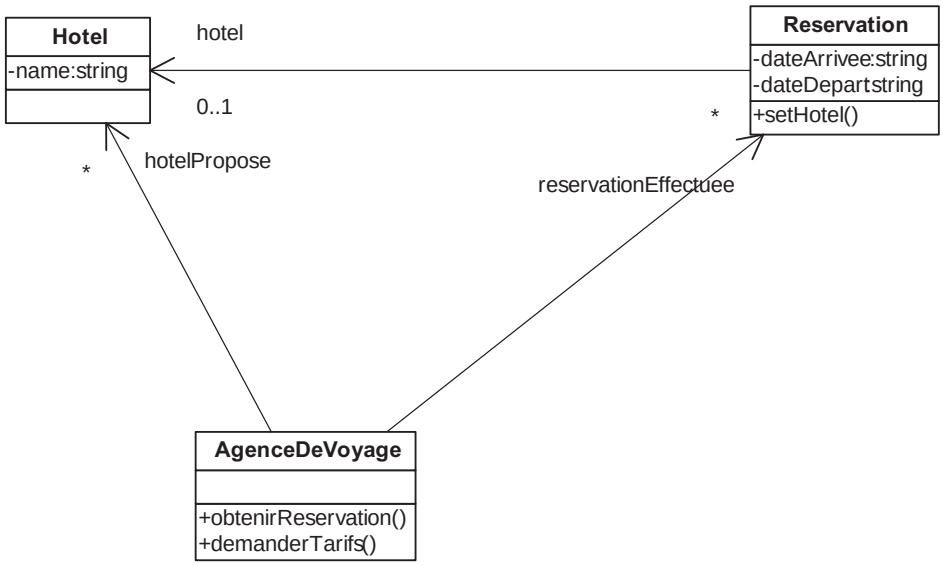


Figure 39
Classes de l'agence de voyage

L'application « agence de voyage » doit gérer les réservations d'hôtels effectuées par des clients. Les trois classes qui apparaissent dans le diagramme sont les objets métier de l'application. Les opérations offertes par ces classes sont les fonctions que nous souhaitons pouvoir réaliser lors de la réservation d'un hôtel : obtenir les tarifs d'un hôtel et effectuer la réservation. Le diagramme représente donc les objets et fonctions métier. Aucune information ne nous permet de l'assimiler à un modèle physique : il n'y a pas de classe Java et il n'y a pas de classe ne représentant pas un objet métier. Nous pouvons donc dire qu'il s'agit d'un modèle conceptuel.

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

Tout dépend du but dans lequel le patron de conception est utilisé.

Si l'objectif est de « casser » les dépendances et/ou de proposer une découpe du modèle en packages, cela peut être intéressant. Nous obtenons des classes abstraites, des interfaces, des associations, des packages, etc., et ces objets ont tout à fait leur place dans un modèle 100 % UML sans qu'il soit nécessaire de le lier à Java. Ils sont donc pertinents dans un modèle conceptuel.

Si l'objectif est d'appliquer les patrons en vue de bénéficier du code qu'ils proposent, cela n'est pas intéressant, car nous intégrons à un modèle conceptuel des contraintes spécifiques à une plate-forme d'exécution.

Prenons l'exemple du patron de conception Singleton.

Ce patron fait en sorte qu'il n'y ait qu'une seule et unique instance d'une classe dans une application. La mise en place de ce patron ne se fait que par du code (Java par exemple) :

```
public class A {
    static singleton = new A();
    public A () {
        return singleton ;
}
```

Ce patron de conception n'est donc pas applicable en 100 % UML (modèle conceptuel), car sa mise en place nécessite l'intégration de code dans le modèle. Le patron de conception Observer, que nous avons déjà vu, est en partie applicable, car une partie de son utilisation consiste en la création de classes et d'associations entre elles (découpe en classe abstraite, héritage, etc.). Par contre, le code des opérations du patron de conception n'est pas utilisable (méthodes `attach()`, `notifyAll()`, etc.) puisqu'il ne s'agit pas d'informations 100 % UML.

Nous venons de voir que l'application de patrons de conception au niveau conceptuel était possible et utile mais qu'elle devait être effectuée avec précaution pour ne pas introduire des considérations physiques là où elles ne doivent pas se trouver.

70. *Le diagramme de séquence représenté à la figure 40 est-il conceptuel ou physique ? Notez qu'il fait intervenir une opération qui n'apparaît pas dans le diagramme de classes initial. Quelle classe doit posséder cette opération ?*

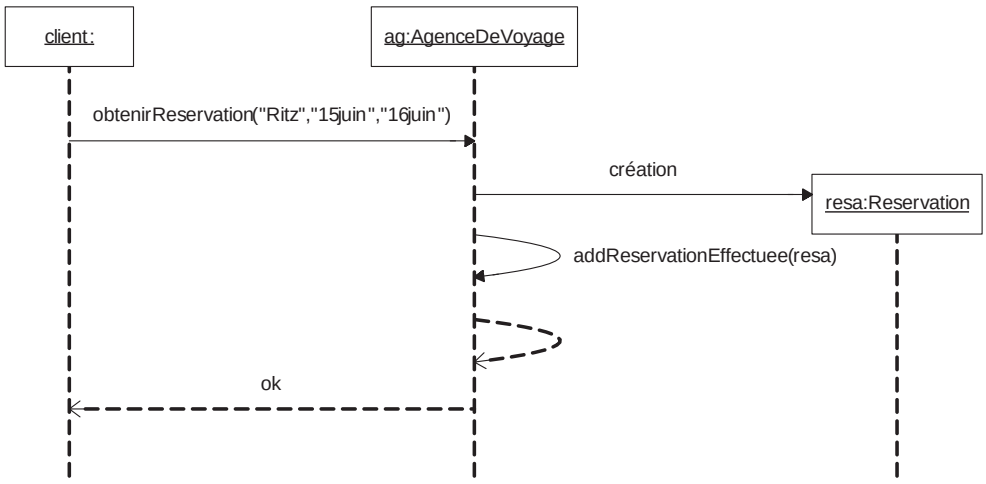


Figure 40

Interaction représentant une réservation

Les classes nommées qui apparaissent dans ce diagramme de séquence sont les classes du modèle conceptuel. Il s'agit donc d'un diagramme de séquence conceptuel. Un diagramme de séquence d'un certain niveau d'abstraction ne peut faire intervenir que des classes du même niveau d'abstraction.

L'opération qui a été ajoutée est `addReservationEffectuee()`. Le diagramme de séquence montre qu'elle doit être mise dans la classe `AgenceDeVoyage` et qu'il s'agit d'une opération interne à cette classe.

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

Les modèles UML que nous considérons utilisent les diagrammes de séquence pour représenter la partie comportementale d'une application. Pour pouvoir spécifier en 100 % UML le comportement de l'agence de voyage, il faudrait donc produire un ensemble de diagrammes de séquence représentant de manière exhaustive l'ensemble des comportements possibles de l'application. En règle générale, c'est impossible puisque le nombre de diagrammes à produire est trop important.

Dans notre cas, c'est totalement impossible, parce qu'un certain nombre de données peuvent prendre un nombre infini de valeurs (attributs de type string) et que les diagrammes de séquence doivent prendre en compte toutes les valeurs possibles. Nous ne pouvons donc pas spécifier en 100 % UML le comportement de l'agence de voyage à l'aide de diagrammes de séquence.

Il est important de noter que si d'autres diagrammes UML sont pris en considération, le problème peut trouver une solution en 100 % UML. Nous avons pris le parti dans ce cours de ne présenter que les trois plus importants des diagrammes UML, car ils sont suffisants pour l'approche que nous utilisons.

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

Oui, rien ne l'empêche. Il faut toutefois avoir conscience que ces tests seraient des tests abstraits, donc non exécutables. Il faudrait dès lors faire les tests physiques correspondants pour qu'ils soient exécutables après génération du code.

73. *Le diagramme représenté à la figure 41 est une concrétisation du diagramme conceptuel de l'agence de voyage. Exprimez les relations d'abstraction entre les éléments des deux diagrammes.*

Il s'agit d'identifier les relations d'abstraction entre les éléments du diagramme physique et ceux du diagramme conceptuel. Les éléments à considérer sont les classes et les associations. Il faut garder en mémoire que tout élément du niveau conceptuel doit être associé à au moins un élément du niveau physique. L'inverse n'est pas vrai.

Nous devons donc trouver les éléments représentant la concrétisation de :

- la classe `Hotel` ;
- la classe `Reservation` ;
- la classe `AgenceDeVoyage` ;
- l'association `hotelPropose` ;

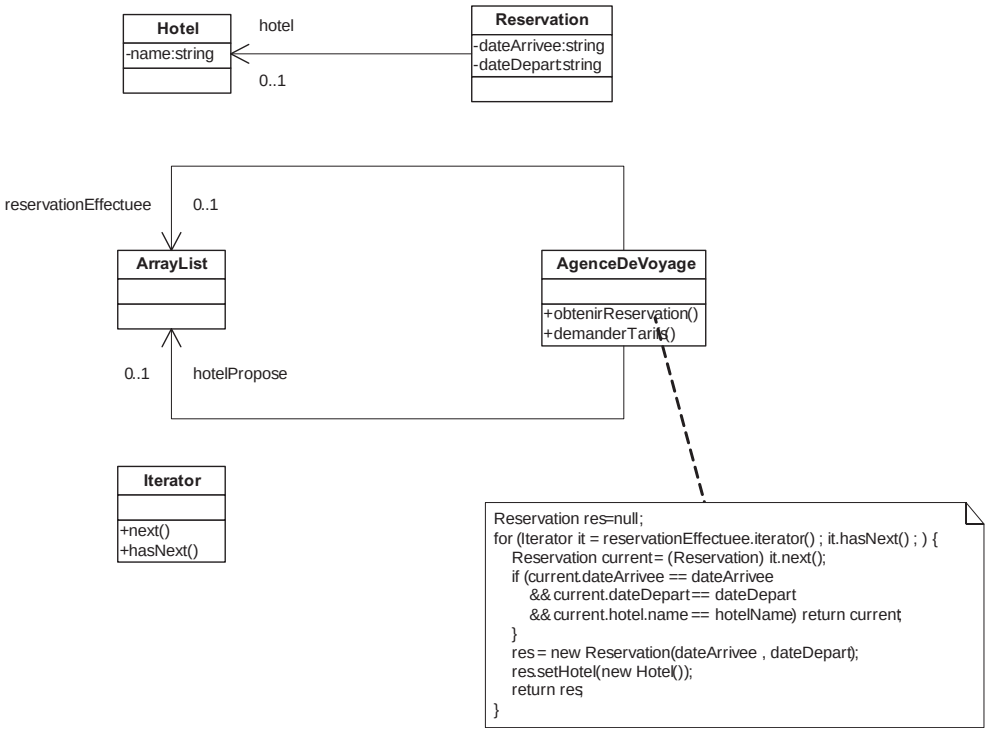


Figure 41

Classes du niveau physique de l'agence de voyage

- l'association `hotel` ;
- l'association `reservationEffectuee`.

Pour les classes, c'est assez facile et naturel. Chaque classe du modèle conceptuel est l'abstraction de la classe de même nom du modèle physique. L'association `hotel` de multiplicité 0..1 est concrétisée par l'association de même nom avec la même multiplicité. Les associations `hotelPropose` et `reservationEffectuee` sont un peu plus délicates à gérer car elles ont une multiplicité *. Elles sont concrétisées par l'association de même nom, et la classe `ArrayList` est utilisée pour représenter la multiplicité *. La classe `ArrayList` participe donc à la concrétisation de deux relations. Il faut noter que la classe `Iterator` n'est la concrétisation d'aucune classe du niveau conceptuel.

Les relations d'abstraction entre diagrammes de classes peuvent être ajoutées aux diagrammes UML.

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) ?*

L'intérêt est d'avoir un modèle physique très proche du code Java. Ainsi, l'opération de génération de code (et de Reverse Engineering) est-elle beaucoup moins complexe et donc beaucoup plus sûre. Mais, comme nous l'avons vu avec la classe `ArrayList` et les

deux associations du niveau conceptuel à la concrétisation desquelles elle participe, établir les relations d'abstraction entre le modèle conceptuel et les éléments physiques très près du code n'est pas simple. La classe `Iterator` nous montre en outre que certains éléments du modèle physique ne sont reliés avec aucun élément du modèle conceptuel. Elle est juste présente pour la génération de code Java.

En fait, plus le modèle physique est proche de Java, moins la génération de code est complexe, mais plus les relations d'abstraction sont importantes et complexes. À l'inverse, moins le modèle physique est proche de Java, plus la génération de code est complexe, mais l'établissement des relations d'abstraction en est normalement facilité.

75. *Construisez le diagramme de séquence concrétisant le diagramme de séquence présenté à la question 70.*

Il s'agit quasiment du même diagramme, si ce n'est que le `addReservationEffectuée` devient `add` et se fait directement sur l'instance de `ArrayList` liée à l'objet `ag`. Cette instance de `ArrayList` est identifiée par `reservationEffectuee` dans le diagramme représenté à la figure 42.

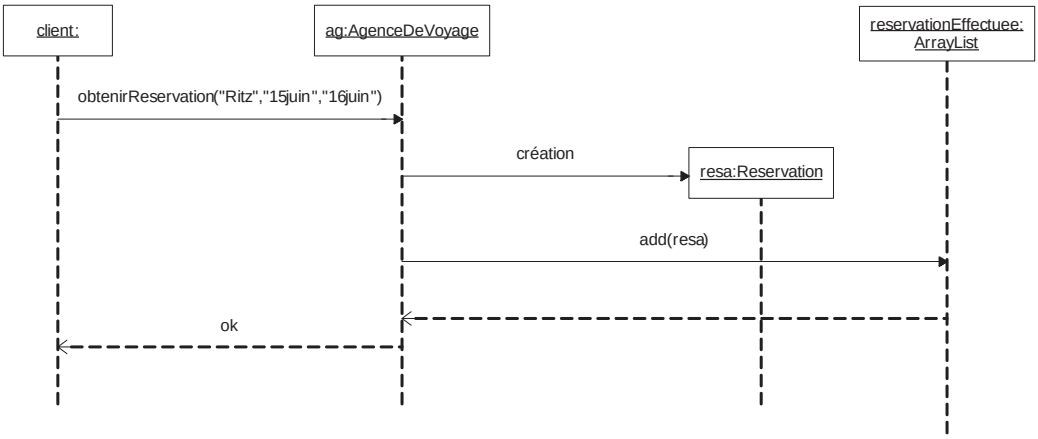


Figure 42

Interaction du niveau physique représentant une réservation

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

Les relations d'abstraction ne peuvent apparaître entre diagrammes de séquence. Notre modèle ne nous permet donc pas de les exprimer.

TD9. Diagrammes de cas d'utilisation

Le diagramme de cas d'utilisation de la figure 43 représente les fonctionnalités d'une agence de voyage classique.

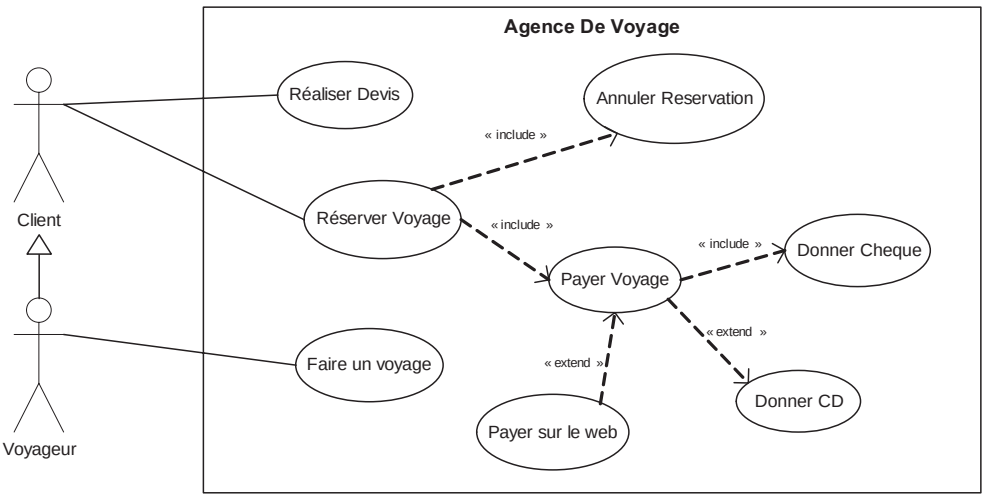


Figure 43

Diagramme de cas d'utilisation de l'agence de voyage

77. *Commentez les acteurs du diagramme de cas d'utilisation.*

L'acteur `Client` représente les clients de l'agence et l'acteur `Voyageur` représente les voyageurs. Comme il s'agit d'entités externes à l'application, rien ne garantit qu'un voyageur soit obligatoirement client de l'agence. Donc, même si certains voyageurs peuvent être clients de l'agence, il ne faut pas mettre de relation d'héritage entre ces deux classes.

78. *Commentez les cas d'utilisation du diagramme de cas d'utilisation.*

Ce diagramme laisse croire que c'est l'agence de voyage qui s'occupe de la réalisation du voyage, ce qui n'est généralement pas le cas. L'agence se charge normalement de vendre des voyages réalisés par d'autres. Il n'est dès lors pas souhaitable d'associer ce cas d'utilisation à l'application « agence de voyage ». Nous supprimons donc ce cas, et, en conséquence, nous supprimons l'acteur `Voyageur`, qui n'est plus relié à aucun cas d'utilisation du diagramme.

Le diagramme d'utilisation de cette question représente les fonctionnalités du système vis-à-vis des entités externes qui interagissent avec lui. Il s'agit du diagramme de cas d'utilisation décrivant les fonctionnalités de l'application au niveau besoin. Il ne faut donc faire apparaître que les fonctionnalités du système (quels services rend le système ?) et ne donner aucune information sur la façon dont ces fonctionnalités sont réalisées (comment les services sont rendus ?).

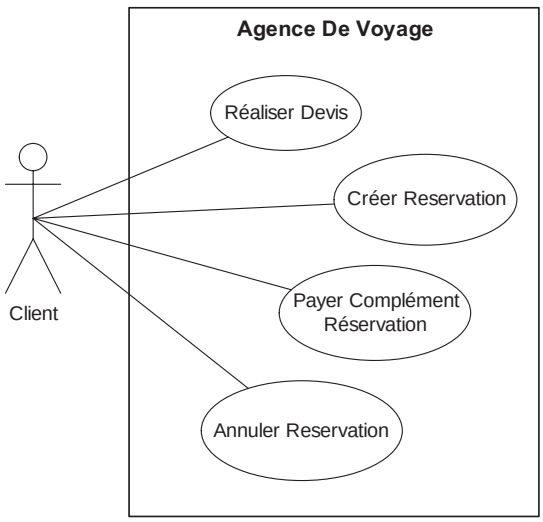
Les relations d'inclusion (`include`) représentent une découpe fonctionnelle. Elles nous informent que, pour réaliser le cas d'utilisation `Reserver Voyage`, il peut être nécessaire de réaliser les cas d'utilisation `Annuler Reservation` et `Payer Voyage`. Ces informations n'ont rien à faire au niveau besoin ; il s'agit d'informations qui ont leur place au niveau conceptuel. Il en va de même des relations d'inclusion entre le cas d'utilisation `Payer Voyage` et les cas d'utilisation `Donner Cheque` et `Donner CB`.

La relation d'extension (`extend`) représente l'ajout d'une fonctionnalité non prévue initialement dans le cas d'utilisation `Payer Voyage`. Ici, il a été ajouté la possibilité de payer la réservation par le Web. Dans ce cas précis, cette extension n'est pas justifiée, car ce cas d'utilisation ne présente pas une extension de comportement mais un troisième moyen de paiement, lequel n'a pas davantage sa place dans ce diagramme que les deux précédents.

Il est important de retenir que l'application doit offrir au client la possibilité de faire une réservation, de payer une réservation, d'annuler une réservation et d'obtenir un devis.

La figure 44 représente le diagramme de cas d'utilisation que nous obtenons.

Figure 44
Diagramme de cas d'utilisation de l'agence de voyage après correction



79. *Donnez la liste des acteurs du système.*

Nous souhaitons réaliser le diagramme de cas d'utilisation du championnat d'échecs présenté au TD6.

La description de l'application donnée au TD6 nous permet d'en identifier trois : l'administrateur, le participant, qui représente un individu qui va s'inscrire à un championnat, et le joueur, qui représente un individu inscrit à un championnat et qui peut donc participer aux parties auxquelles il est inscrit.

Il ne faut pas établir de lien d'héritage entre les acteurs `Participant` et `Joueur`, car tous les participants ne sont pas des joueurs. Ils ne le deviendront qu'après s'être

inscrits. Or tous les joueurs ne sont pas obligatoirement des participants puisque rien n'oblige un joueur à s'inscrire à un nouveau championnat.

80. *Donnez la liste des cas d'utilisation du système en les liant aux acteurs.*

Les cas d'utilisation associés à l'administrateur sont :

- créer un championnat ;
- créer l'ensemble des parties associées à un championnat.

Les cas d'utilisation associés au participant sont :

- s'inscrire à un championnat.

Les cas d'utilisation associés au joueur sont :

- consulter le calendrier lui donnant les informations sur sa liste de parties ;
- jouer les parties d'un championnat ;
- consulter son classement.

81. *Donnez le diagramme de cas d'utilisation du système.*

Le diagramme de la figure 45 représente les cas d'utilisation de l'application du championnat d'échecs.

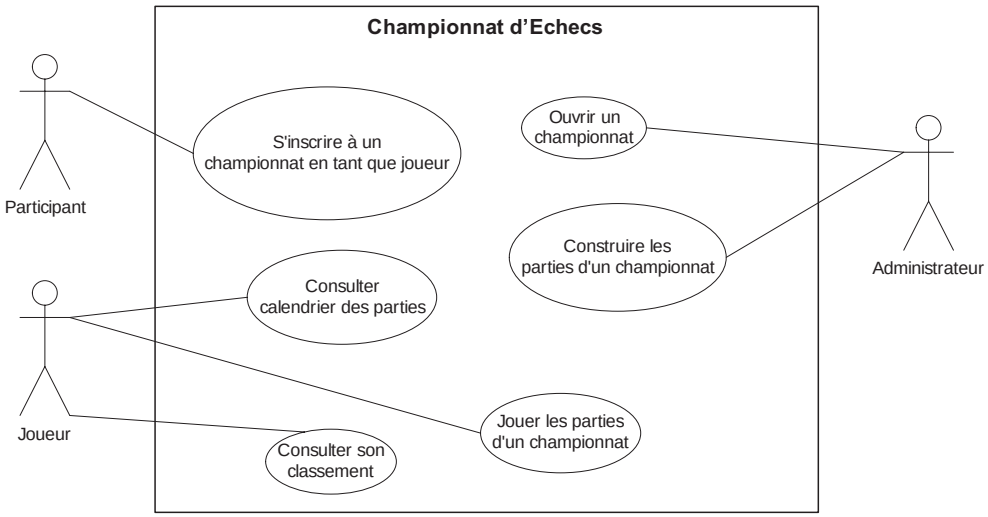


Figure 45

Diagramme de cas d'utilisation du composant Championnat d'Echecs

82. *Reprenez les diagrammes de séquence réalisés au TD6 pour l'application de championnat d'échecs, et expliquez comment les relier au diagramme de cas d'utilisation obtenu à la question précédente.*

Il faut juste faire le lien entre les objets non typés des diagrammes de séquence et les acteurs du diagramme de cas d'utilisation. Ainsi, l'objet `L'administrateur` devient-il

une instance de l'acteur `Administrateur`. Dans le diagramme de séquence représentant l'inscription à un championnat, les objets `MrBF` et `MrBS` deviennent des instances de l'acteur `Participant`. Dans le diagramme de séquence représentant le déroulement d'une partie, les objets `MrBF` et `MrBS` deviennent des instances de l'acteur `Joueur`.

TD10. Développement avec UML

Une association d'ornithologie vous confie la réalisation du système logiciel de recueil et de gestion des observations réalisées par ses adhérents (le logiciel DataBirds). L'objectif est de centraliser toutes les données d'observation arrivant par différents canaux au sein d'une même base de données, qui permettra ensuite d'établir des cartes de présence des différentes espèces sur le territoire géré par l'association.

Les données à renseigner pour chaque observation sont les suivantes :

- Nom de l'espèce concernée. Il y a environ trois cents espèces possibles sur le territoire en question. Si l'observation concerne plusieurs espèces, renseigner plusieurs observations.
- Nombre d'individus.
- Lieu de l'observation.
- Date de l'observation.
- Heure de l'observation.
- Conditions météo lors de l'observation.
- Nom de chaque observateur.

Quelle que soit la façon dont sont collectées les données, celles-ci sont saisies dans la base dans un état dit « à valider ». Tant que les données ne sont pas validées par les salariés de l'association, des modifications peuvent être apportées aux données.

La validation des données se fait uniquement par les salariés de l'association qui ont le droit de modifier la base de DataBirds. Ils doivent vérifier que les données saisies sont cohérentes. Plus précisément, ils doivent valider les noms des observateurs (les noms doivent correspondre à des noms d'adhérents) et l'espèce (celle-ci doit correspondre à une espèce connue sur le territoire).

Après validation, une saisie se trouve soit dans l'état dit « validé », soit dans l'état dit « non validé ». Les saisies dans l'état « non validé » sont automatiquement purgées de la base une fois par semaine.

Grâce aux données saisies et validées, l'association souhaite pouvoir établir différents types de cartes de présence des différentes espèces :

- Cartes géographiques par espèce présentant un cumul historique des populations. Ce traitement peut être demandé par un adhérent.
- Cartes des observations réalisées par chaque observateur. Ce traitement peut être demandé par un salarié uniquement.

Ces cartes de présence des oiseaux sont générées par DataBirds et accessibles soit par le Web, soit par demande *via* un courrier électronique ou postal.

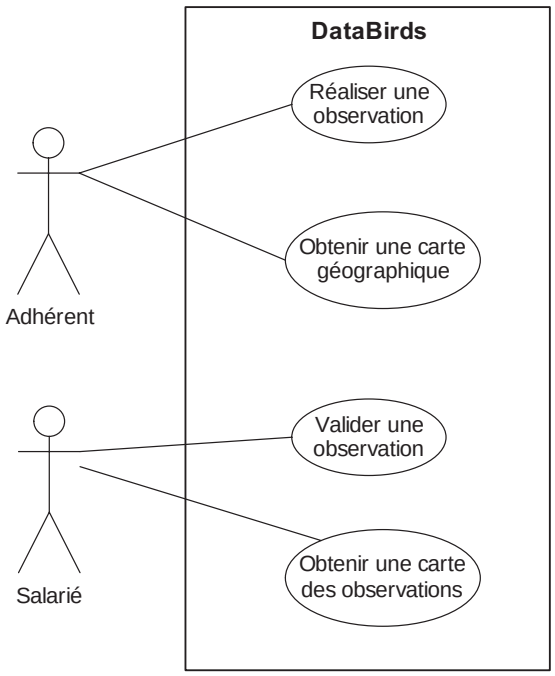
83. *Effectuez la première étape de la méthode.*

L'objectif de cette étape est de produire le diagramme de cas d'utilisation du niveau besoin. Nous identifions deux acteurs, les adhérents et les salariés. Il n'y a pas de relation d'héritage entre ces acteurs, car rien n'oblige un salarié à être aussi adhérent de l'association.

Les cas d'utilisation associés à l'adhérent sont la réalisation d'une observation (la saisie des informations relatives à l'observation) et l'obtention de la carte géographique relative à une espèce. Les cas d'utilisation associés au salarié sont la validation d'une observation et l'obtention de la carte des observations d'un adhérent. L'application doit aussi permettre la suppression des observations qui n'ont pas été validées, mais cette fonctionnalité n'étant reliée à aucun acteur, elle doit être effectuée automatiquement une fois par semaine. Etant donné que les cas d'utilisation représentent les fonctionnalités offertes par l'application à son environnement, cette fonctionnalité de suppression des observations ne sera pas représentée au niveau besoin.

La figure 46 représente le diagramme de cas d'utilisation du niveau besoin de l'application DataBirds.

Figure 46
Diagramme de cas d'utilisation de l'application DataBirds



84. *Effectuez la deuxième étape de la méthode (niveau besoin – comportement).*

L'objectif de cette étape est de produire un diagramme de séquence nominal par cas d'utilisation ainsi qu'un diagramme de séquence pour chaque erreur possible.

Pour le fonctionnement nominal du cas d'utilisation `Réaliser une observation`, nous considérons que l'adhérent `Jean` réalise l'observation qu'il a faite sur trois merles à Paris, le 1er juillet 2006, à 12 heures. Le message `réaliserObservation()` est envoyé à l'application DataBirds, qui crée l'objet `obs1` de type `Observation`.

L'interaction illustrée à la figure 47 représente cette exécution.

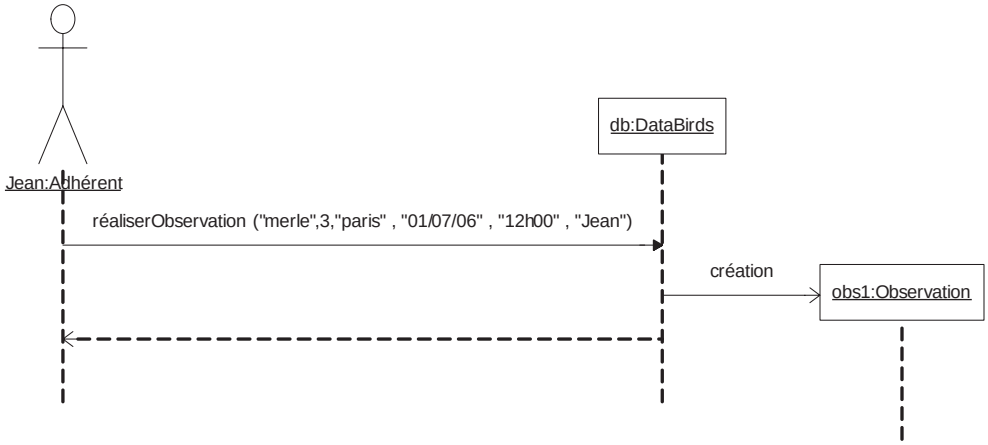


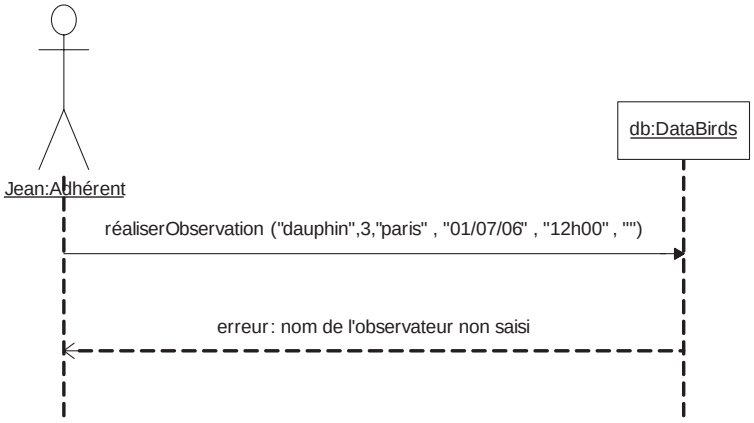
Figure 47

Interaction représentant la réalisation d'une observation

Pour un fonctionnement soulevant une erreur du cas d'utilisation « Réaliser une observation », nous considérons le cas où l'adhérent oublie de donner le nom de l'observateur. Dans ce cas, l'application DataBirds doit signaler une erreur. Ce fonctionnement est représenté par l'interaction illustrée à la figure 48.

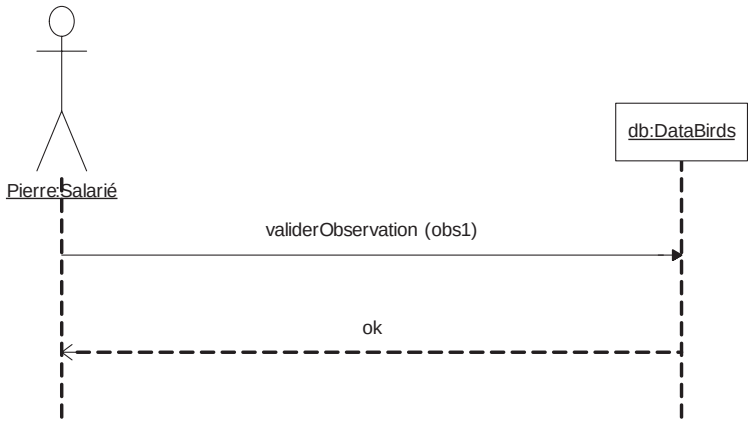
Figure 48

Interaction représentant la réalisation d'une observation levant une erreur



Nous considérons maintenant le fonctionnement nominal du cas d'utilisation « Valider une observation ». Un salarié demande à l'application de valider une observation, et cette dernière lui répond que tout est correct. L'interaction illustrée à la figure 49 représente ce fonctionnement.

Figure 49
Interaction représentant la validation d'une observation



Nous n'avons présenté ici qu'une petite partie du diagramme de séquence à produire. Ce dernier doit contenir un diagramme pour le fonctionnement nominal de chaque cas d'utilisation et un ou plusieurs diagrammes pour chaque fonctionnement erroné.

Pour assurer la cohérence entre les parties fonctionnelle, comportementale et structurelle, nous rappelons que les interactions doivent faire apparaître les acteurs identifiés à l'étape 1 et les classes qui seront utilisées à l'étape 3.

85. *Effectuez la troisième étape de la méthode.*

L'objectif de cette étape est de produire un diagramme de classes représentant les données spécifiées dans la description de l'application. Les interactions que nous avons données en solutions de la question précédente font apparaître les classes `DataBirds` et `Observation`.

Nous introduisons la classe `Adhérent`, car l'application doit pouvoir vérifier que les observations sont faites par des adhérents. Les opérations associées aux cas d'utilisation sont des opérations de la classe `DataBirds`. Les attributs de la classe `Observation` permettent de stocker toutes les caractéristiques d'une observation.

Nous avons introduit deux booléens. Le booléen `aValider` est à vrai lorsque l'observation n'a pas encore été validée par un salarié. Lorsque ce booléen est à vrai, il faut regarder le booléen validé pour savoir si l'observation est effectivement validée. Les classes `DataBirds` et `Observation` sont associées, car plusieurs observations sont gérées par l'application. De la même façon, nous représentons le fait que plusieurs adhérents peuvent être identifiés.

La figure 50 représente le diagramme de classes obtenu.

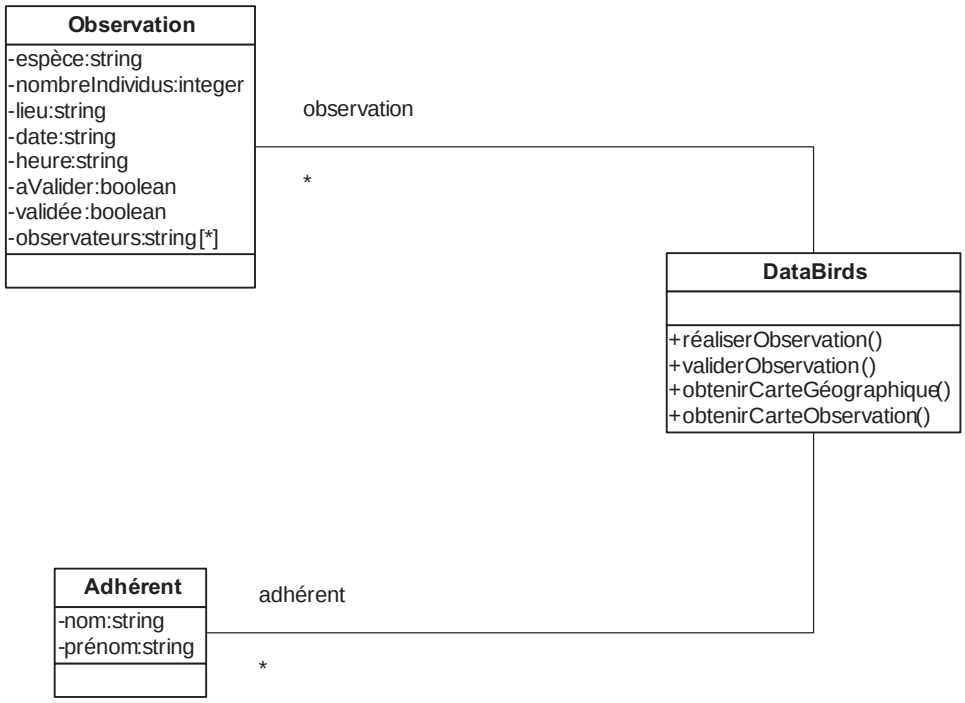


Figure 50

Classes de l'application DataBirds *au niveau besoin*

86. *Effectuez la quatrième étape de la méthode.*

L'objectif de cette étape est de produire la liste des composants du système et un diagramme de cas d'utilisation par composant.

Nous décomposons notre système en deux composants : le gestionnaire des observations et le gestionnaire des cartes.

Le gestionnaire des observations s'occupe de la création, de la validation et de la suppression des observations. Étant donné que nous sommes au niveau conceptuel, nous représentons cette fois la suppression des observations. Ce composant est relié aux acteurs `Adhérent` et `Salarié`, qui sont concernés par la création et la validation des observations. Il est aussi relié à la base des adhérents, qui est un composant externe à l'application.

Dans le descriptif de l'application, aucune information n'est donnée sur la gestion des adhérents. Nous savons juste qu'il est nécessaire d'avoir accès à une base des adhérents. Il s'agit donc d'un composant externe à l'application DataBirds.

La figure 51 représente le diagramme de cas d'utilisation du composant de gestion des observations.

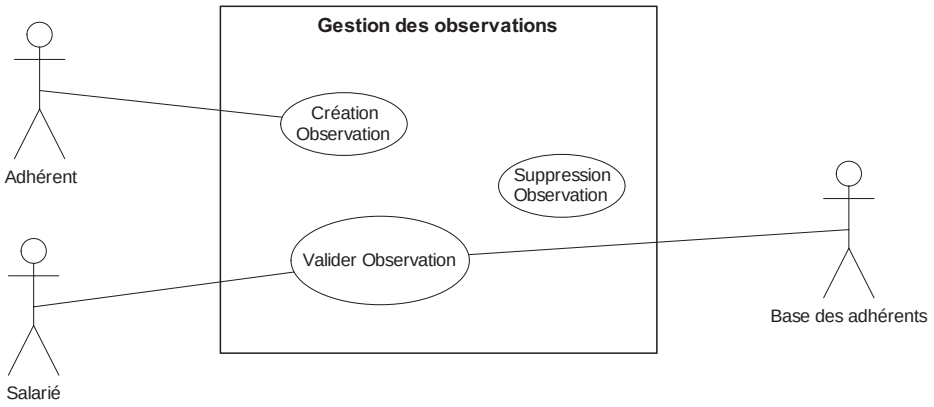


Figure 51

Diagramme de cas d'utilisation du composant Gestion des observations

Le gestionnaire de cartes s'occupe de la construction des différentes cartes que peuvent demander les acteurs. Il contient deux cas d'utilisation, la construction des cartes géographiques et celle des cartes d'observation. Ce composant est relié aux acteurs `Adhérent` et `Salarié`, car ils ont chacun accès à un des cas d'utilisation qu'il contient. Il est aussi relié à une base de données des observations, car il en a besoin pour produire les cartes demandées.

Cette base de données est en fait produite par le composant de gestion des observations, mais comme il n'est pas possible d'établir des liens entre composants, nous devons le représenter par un acteur. Notons que l'acteur `BDObservation` n'est pas une personne physique, ce qui ne pose aucun problème puisque nous sommes au niveau conceptuel.

La figure 52 représente le diagramme de cas d'utilisation du composant de gestion des cartes.

Il ne faut pas oublier de préciser les relations de résolution entre les cas d'utilisation du niveau d'analyse et ceux des composants. Dans notre cas, cette tâche est assez simple :

- Le cas d'utilisation « Réaliser une observation » est résolu par le cas d'utilisation « Création Observation ».
- Le cas d'utilisation « Obtenir une carte géographique » est résolu par le cas d'utilisation « Construire Carte géographique ».
- Le cas d'utilisation « Valider une observation » est résolu par le cas d'utilisation « Valider Observation ».
- Le cas d'utilisation « Obtenir une carte des observations » est résolu par le cas d'utilisation « Construire Carte observations ».

87. *Effectuez la cinquième étape de la méthode.*

L'objectif de cette étape est de produire un diagramme de séquence nominal par cas d'utilisation, ainsi qu'un diagramme de séquence pour chaque erreur possible.

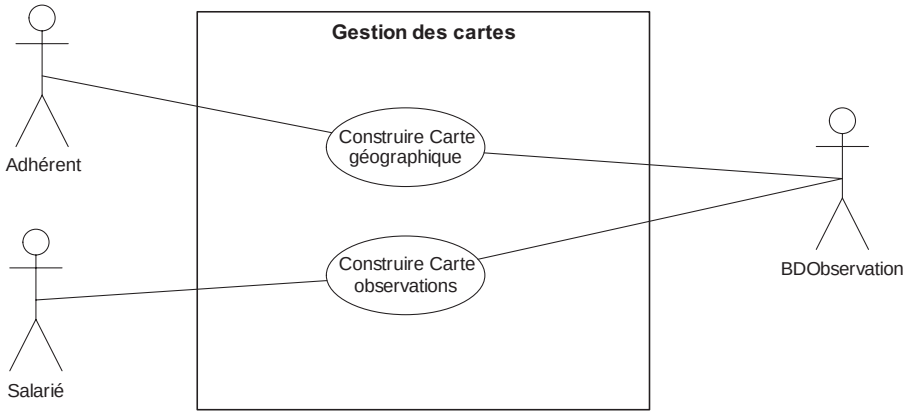


Figure 52

Diagramme de cas d'utilisation du composant de gestion des cartes

Nous reprenons le cas d'utilisation qui consiste à valider une observation. Cette fois, comme nous sommes au niveau conceptuel, nous devons faire apparaître les différentes étapes de la validation : la vérification que les observateurs sont bien des adhérents (ce qui nécessite un accès à la base de données des adhérents) et la vérification que l'espèce observée est bien répertoriée dans le territoire concerné.

La figure 53 représente l'interaction associée à ce cas d'utilisation.

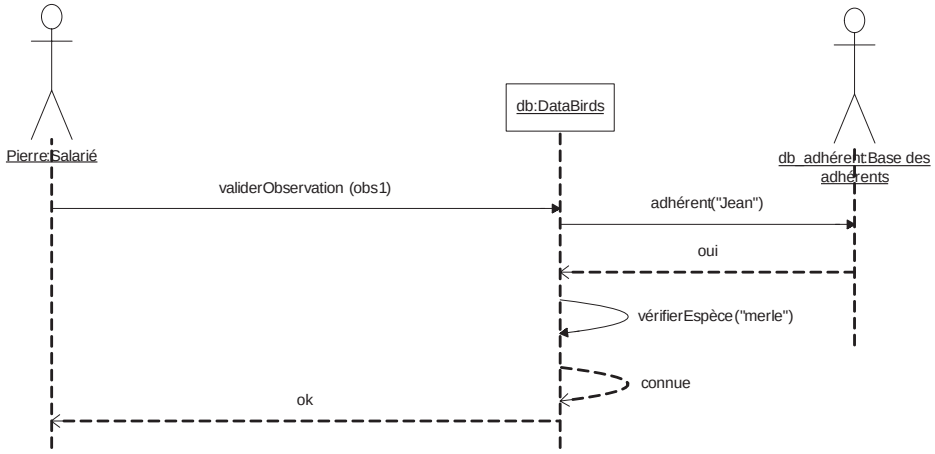


Figure 53

Interaction au niveau conceptuel représentant la validation d'une observation

Nous n'avons présenté ici qu'une petite partie du diagramme de séquence à produire. Ce dernier doit en effet contenir un diagramme pour le fonctionnement

nominal de chaque cas d'utilisation et un ou plusieurs diagrammes pour chaque fonctionnement erroné.

Pour assurer la cohérence entre les parties fonctionnelle, comportementale et structurelle, nous rappelons que les interactions doivent faire apparaître les acteurs identifiés à l'étape 4 et les classes utilisées à l'étape 6.

88. *Effectuez la sixième étape de la méthode.*

L'objectif de cette étape est de produire les classes des composants. Toutes les classes d'un même composant sont regroupées dans un même package. Il est important de préciser aussi les relations de dépendance entre les composants.

Le composant « Gestion des Observation » est représenté par le package `gestionObservation`. Il contient les classes `Espèce` et `Observation`, qui représentent respectivement les espèces visibles sur le territoire et les observations faites par les adhérents. Le composant contient aussi la classe `BaseDeDonnées`, qui contient (relation de composition) l'ensemble des espèces et l'ensemble des observations. Cette classe possède aussi les opérations qui seront utilisées par l'environnement du composant (`ajouterObservation()`, `validerObservation()`, `supprimerObservationsNonValidées()`).

Le composant « Gestion des cartes » est représenté par le package `gestionCartes`. Nous avons fait le choix de créer la classe `Carte`, qui est une classe abstraite. Notre intention est de stocker dans une mémoire cache n'importe quelle carte (géographique ou observation) déjà construite. Les classes `CarteGéographique` et `CarteObservation` héritent donc de cette classe abstraite. Celles-ci sont associées aux classes du composant « Gestion des Observation » dont elles ont besoin. La classe `GestionnaireCarte` contient l'ensemble des cartes déjà créées et possède les opérations qui seront utilisées par l'environnement du composant (`construireCarteGeographique()`, `construireCarteObservation()`, `rechercheCarte()`).

Soulignons que cette spécification de la structure des composants n'est pas complète. Nous la jugeons cependant suffisante pour notre propos.

89. *Effectuez la septième étape de la méthode.*

L'objectif de cette étape est de produire les classes des composants en intégrant les classes de la plate-forme d'exécution. À partir de la spécification des composants faite à la question précédente, il suffit de remplacer les associations de multiplicité * par des liens vers la classe `ArrayList`. Ces modifications sont représentées à la figure 55.

Les liens d'abstraction entre les niveaux conceptuel et physique sont relativement triviaux car ils apparaissent entre les packages de même nom. Pour ce qui est des liens d'abstraction pour les associations à multiplicité *, il faut refaire ce qui a été présenté au TD8.

90. *Effectuez la huitième étape de la méthode sur une seule classe.*

L'objectif de cette étape est de produire les cas de test abstraits.

À titre d'exemple, nous pouvons spécifier les cas de test abstrait de la classe `BaseDeDonnées` et écrire les tests abstraits concernant la validation d'une observation. Une

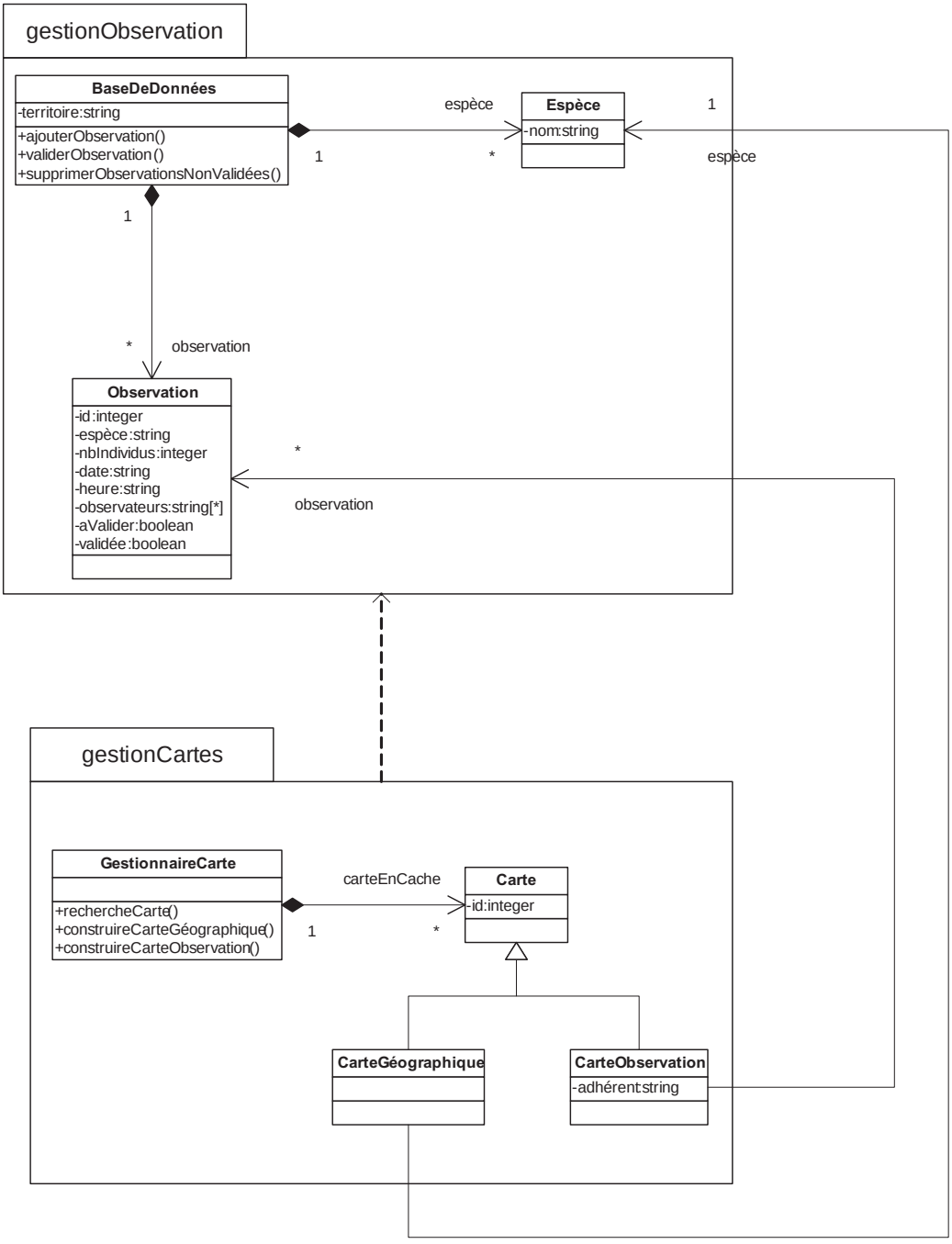


Figure 54

Classes des composants de gestion des observations et de gestion des cartes

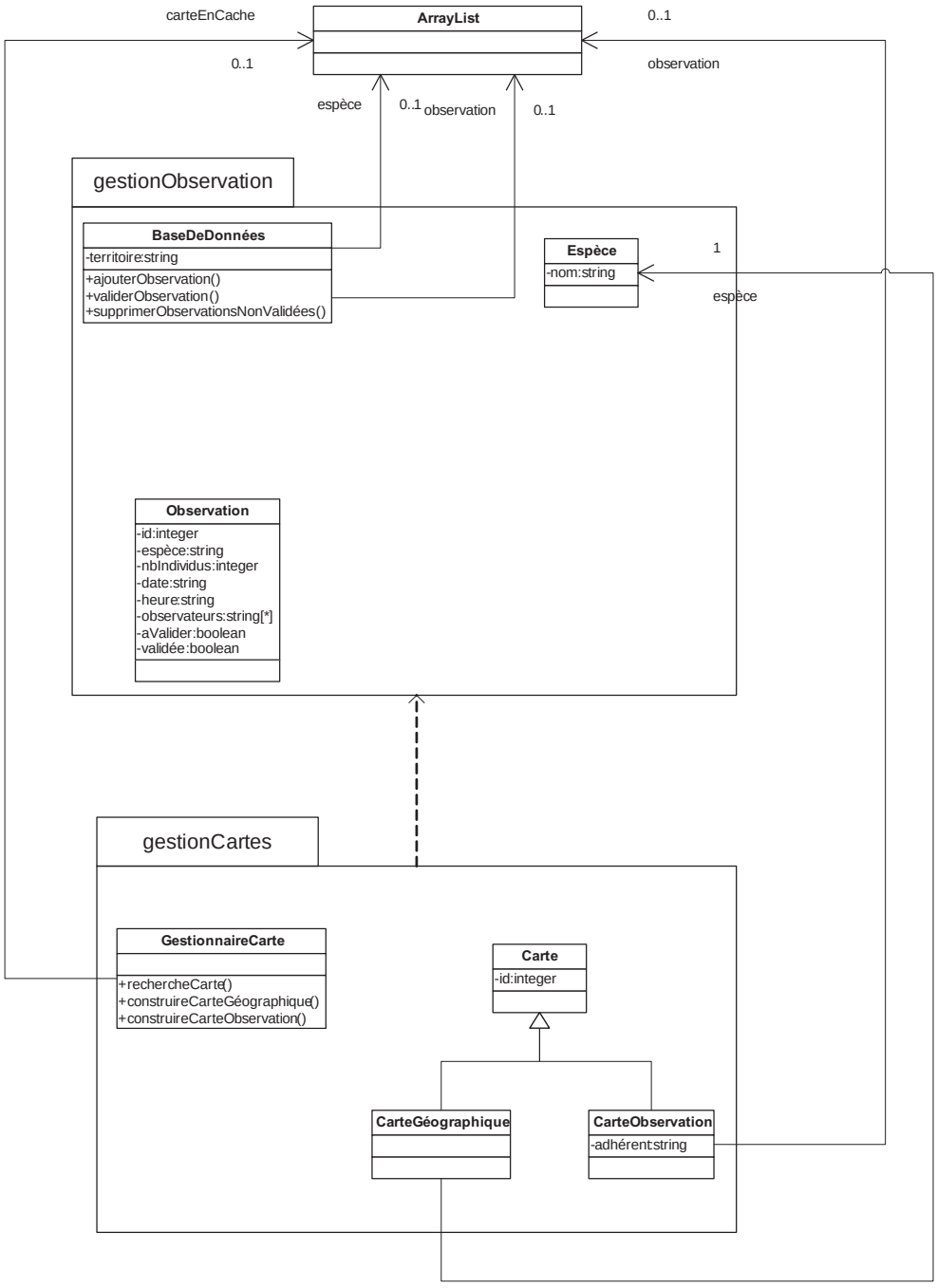


Figure 55

Classes des composants au niveau physique

faute éventuelle serait de valider une observation portant sur une espèce n'appartenant pas au territoire.

Un cas de test visant à révéler une telle faute est spécifié à la figure 56.

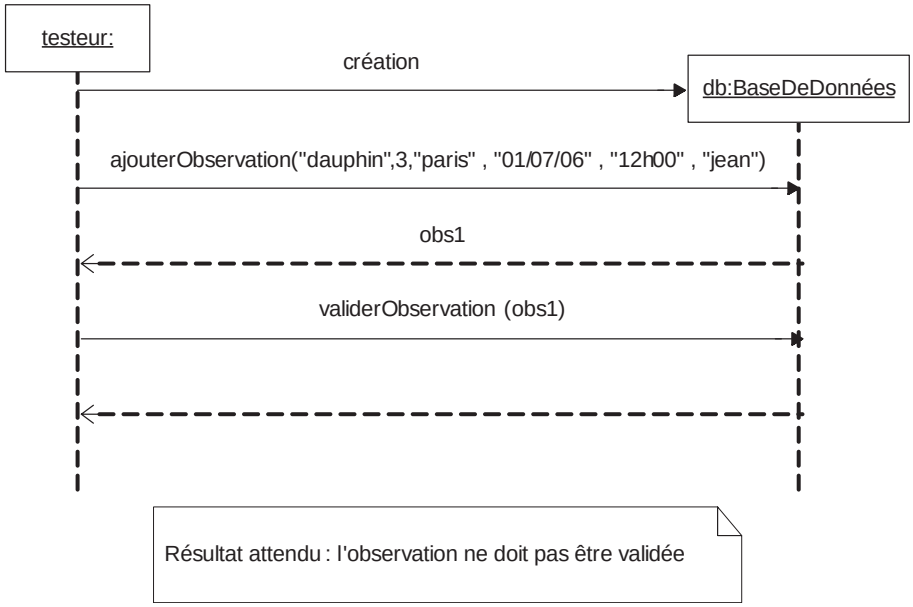


Figure 56

Diagramme de cas de test de la classe BaseDeDonnées

91. *Effectuez la neuvième étape de la méthode.*

L'objectif de cette étape est de produire le diagramme de cas d'utilisation représentant les fonctionnalités offertes par les composants mais au niveau physique.

Cependant, cette étape n'est nécessaire que si certaines fonctionnalités sont réalisées par la plate-forme d'exécution. Cela permet en ce cas de faire apparaître les fonctionnalités directement offertes par la plate-forme et celles qu'il faudra développer.

Étant donné que, dans notre cas, aucune fonctionnalité n'est directement offerte par la plate-forme, cette étape n'est pas nécessaire.

Annexes