Exception : Agendas hiérarchiques

Nous allons modéliser des agendas simplifiés (exercices 1, 2, 3 et 4) et des groupes hiérarchiques d'agendas (exercice 5). L'exercice 6 demande de dessiner le diagramme de classe de l'application mais il est bien sûr conseillé de le dessiner et le faire évoluer au fil des exercices. L'exercice 7 propose de généraliser les agendas.

Exercice 1 : Agenda

Pour simplifier, on considère un agenda pour une année avec un seul rendez-vous possible par journée. Les créneaux sont repérés par un numéro de 1 à 366 correspondant au jour du rendez-vous. Par exemple, 10 correspond au créneau du 10 janvier.

Un agenda, modélisé par l'interface `Agenda`, fournit les quatre opérations suivantes :

— obtenir le nom d'un agenda,

— enregistrer pour un créneau un rendez-vous qui sera ici limité à une chaîne de caractères non vide (`String`). Ainsi on peut enregistrer le rendez-vous "Examen" pour le créneau 14. Si le créneau est déjà occupé, une exception `OccupeException` est levée,

— annuler le rendez-vous d'un créneau donné. Par exemple, on peut supprimer le rendez-vous du créneau 14. Si le créneau est libre, rien ne se passe. La valeur de retour de cette opération indique si l'agenda a été modifiée (valeur vrai, le rendez-vous a été annulé) ou non (valeur faux, le créneau était libre).

— obtenir le rendez-vous correspondant à un créneau. Si ce créneau ne contient pas de rendez-vous enregistré, une erreur est signalée [1] grâce à l'exception `LibreException`.

Les trois dernières opérations attendent un créneau valide. Si ce n'est pas le cas, elles lèveront l'exception `CreneauInvalideException`.

1.1. `CreneauInvalideException` est une exception non vérifiée car on s'attend à ce que l'appelant fournisse un créneau valide. Il n'est donc pas souhaitable que le compilateur lui dise que le créneau pourrait être invalide. Les exceptions `OccupeException` et `LibreException` sont vérifiées car l'appelant ne peut généralement pas connaître le contenu de l'agenda. Il est donc souhaitable que le compilateur lui signale qu'il pourrait y avoir une erreur qu'il devra prendre en compte (enregistrer un rendez-vous sur un créneau occupé ou obtenir le rendez-vous d'un créneau libre).

Écrire ces trois classes d'exception.

1.2. Exécuter le programme de test ExceptionsTest et corriger les éventuelles erreurs signalées.

1.3. L'interface agenda contient des erreurs. Les identifier et les corriger.

1.4. Expliquer pourquoi Agenda a été modélisé par une interface plutôt qu'une classe.

1. Un autre choix possible aurait été de retourner une valeur particulière. Ici **`null`** est un bon candidat pour indiquer qu'il n'y a pas de rendez-vous pour le créneau demandé.

Exercice 2 : ObjetNomme

La classe `ObjetNomme` modélise un objet qui a un nom. Même si cette classe est complètement écrite, elle est déclarée abstraite car on considère qu'elle ne devra pas être utilisée telle quelle mais devra être spécialisée.

2.1. Compléter `ObjetNomme` : une exception `IllegalArgumentException` doit être levée si le nom fourni en paramètre du constructeur n'a pas au moins un caractère.

2.2. La tester avec la classe `ObjetNommeTest`.

Exercice 3 : AgendaAbstrait

La classe abstraite `AgendaAbstrait` a été définie. Elle réalise l'interface `Agenda` et hérite de `ObjetNomme`. Elle permet donc de factoriser la définition du nom.

On constate que de nombreuses opérations de l'agenda prennent en paramètre un créneau. Il est nécessaire que ce créneau soit valide. Pour éviter d'écrire de nombreuses fois les mêmes instructions, on va définir une méthode [2] `verifierCreneauValide(int creneau)` qui lèvera l'exception `CreneauInvalideException` si le créneau n'est pas dans les limites attendues.

3.1. Écrire dans la classe `AgendaAbstrait` la méthode `verifierCreneauValide`.

3.2. Expliquer l'intérêt d'avoir à la fois l'interface `Agenda` et la classe abstraite `AgendaAbstrait`.

Exercice 4 : AgendaIndividuel

La classe `AgendaIndividuel` est une implantation de l'agenda pour laquelle on décide de stocker les rendez-vous dans un tableau.

4.1. Compléter la classe `AgendaIndividuel` pour ajouter les aspects liés aux exceptions.

4.2. La tester grâce à la classe `AgendaIndividuelTest`.

Exercice 5 : GroupeAgenda

Un groupe d'agendas est utilisé pour manipuler plusieurs agendas, qu'ils soient des agendas individuels ou des groupes d'agendas. Un groupe permet d'enregistrer un rendez-vous sur tous les agendas d'un groupe ou d'obtenir le rendez-vous commun à l'ensemble des agendas du groupe. Une opération permet d'ajouter un nouvel agenda dans un groupe.

L'opération « enregistrer » garantit que soit le rendez-vous est enregistré dans tous les agendas du groupe, soit dans aucun. Dans le cas où le rendez-vous n'a pas pu être enregistré, l'exception `OccupeException` est levée.

Quand on demande le rendez-vous d'un créneau, on obtient l'exception `LibreException` si tous les agendas sont libres pour ce créneau, **`null`** si deux agendas ont des rendez-vous différents ou le rendez-vous commun à tous les agendas.

5.1. Écrire la classe `GroupeAgenda` sachant que l'on doit utiliser l'interface `java.util.List` pour stocker les agendas contenus dans le groupe.

5.2. La tester avec la classe `GroupeAgendaTest`.

Exercice 6 : Diagramme de classe

Dessiner le diagramme de classe [3] de l'application faisant apparaître toutes les classes et interfaces de cette partie. On ne fera pas apparaître les constructeurs, attributs et opérations.

2. En Java8, cette méthode pourrait être définie comme méthode de classe de l'interface `Agenda`. La classe `AgendaAbstrait` deviendrait inutile.

3. Normalement, il a été construit au fur et à mesure de la réalisation des exercices précédents.

Exercice 7 : Généralisation

Envisageons plusieurs extensions à notre application.

7.1. Définir une opération « proposer » sur un groupe d'agenda qui propose un nouveau rendez-vous à tous les agendas du groupe. Ce rendez-vous sera enregistré si l'agenda est libre sur le créneau considéré, ignoré sinon. La tester grâce à la classe `GroupeAgendaProposerTest`.

7.2. Ajouter des informations [4] sur l'exception OccupeException : le créneau qui est occupé, le rendez-vous sur le créneau et le nom de l'agenda.

7.3. Souvent un agenda a peu de rendez-vous par rapport au nombre de créneaux disponibles. Par exemple, si on considère une année avec des créneaux de 15 minutes, on a plus de 35000 créneaux. Utiliser un tableau consomme donc beaucoup de place. Comment faire pour être plus économe en place mémoire tout en conservant des opérations efficaces pour les agendas ?

7.4. Dans la solution proposée, un rendez-vous est réduit à une chaîne de caractères. Dans un autre contexte, on pourrait vouloir indiquer en plus un numéro de salle pour le rendez-vous. On pourrait aussi dans encore un autre contexte vouloir préciser un ordre du jour, etc.

Indiquer comment il serait possible de définir des agendas qui puissent simplement être adaptés à ces types de changement.

4. Attention aux informations qui sont associées à une exception car ceci peut conduire à violer le principe d'encapsulation. Par exemple, si un GroupeAgenda transmet celui de ses agendas occupé via une exception alors, quiconque récupèrera l'exception pourra modifier cet agenda occupé, et par exemple annuler le rendez-vous, sans passer par `GroupeAgenda`.