NFA032 – TP 6 (Suite et fin de l'application Librairie)

17 mars 2019

Objectifs

Dans ce Tp nous écrirons une version très basique d'application de gestion d'une librairie en utilisant les classes écrites lors des tps précédents. Ce Tp est long et sera travaillé en séance, mais aussi à la maison. Sur le plan des apprentissages nous allons nous concentrer sur deux points : (1) comment organiser le code de notre application ; (2) comment gérer les erreurs pendant l'exécution. **Récupérez l'archive zip contenant le squelette de code fourni** : il contient un paquetage avec toutes ces classes (incomplètes pour la plupart). Ajoutez ce paquetage dans un nouveau projet NFA032-Tp6. Les questions de ce tp vous guideront dans l'étude et modification de ce code.

1 Organisation du code

Notre application est destinée à interagir avec des utilisateurs. Dans ce cas, un principe très utile est de **séparer en parties bien distinctes** le code destiné à la présentation visuelle et à l'interface d'entrée /sortie avec l'utilisateur, du code qui permet l'accès aux données et la réalisation des traitements sur ces données et/ou sur celles entrées par l'utilisateur. La première « couche » du code est dite « de présentation ou d'interaction » (ex : saisie des données pour la recherche d'un livre, affichage de tous les résultats d'une recherche). La deuxième est appelé couche « métier ou logique » (ex : méthode de recherche qui prend en paramètre un titre et auteur et retourne le livre trouvé en stock). Les méthodes de la couche métier ont pour principe de ne réaliser aucune interaction avec l'utilisateur. La mise en place du programme final intègre les deux couches : la couche d'interaction présente les opérations à l'utilisateur, acquiert les données entrées par celui-ci (p.e. pour la recherche d'un livre), puis « appelle » la couche logique pour réaliser l'opération (de recherche dans le stock contenant les données de l'application). La couche métier retourne un résultat pour l'opération invoquée (et pas un affichage) vers la couche d'interaction que cette dernière finit par afficher. Nous compterons cinq classes dans notre application. Les classes `Livre` et `StockLivre` déjà écrites lors de précédents tps, et trois nouvelles classes : `Librairie`, `ApplicationTexte` et `RunAppli`.

La figure 1 montre l'architecture de notre application, les dépendances entre classes et les classes métier et d'interaction. Par exemple, une variable d'instance dépend de la classe qui a permis de la définir. Ainsi, la variable stock de la classe StockLivres dépend de la classe Livre, puisque cette variable est définie comme une liste de Livre.

La séparation entre code métier et code d'interaction est intéressante non seulement en termes de clarté, uniformité et simplicité du code. Elle permet surtout ne pas faire dépendre (de manière artificielle) les opérations métier sur autre chose que les données sur lesquelles elles opèrent. Il devient alors beaucoup plus

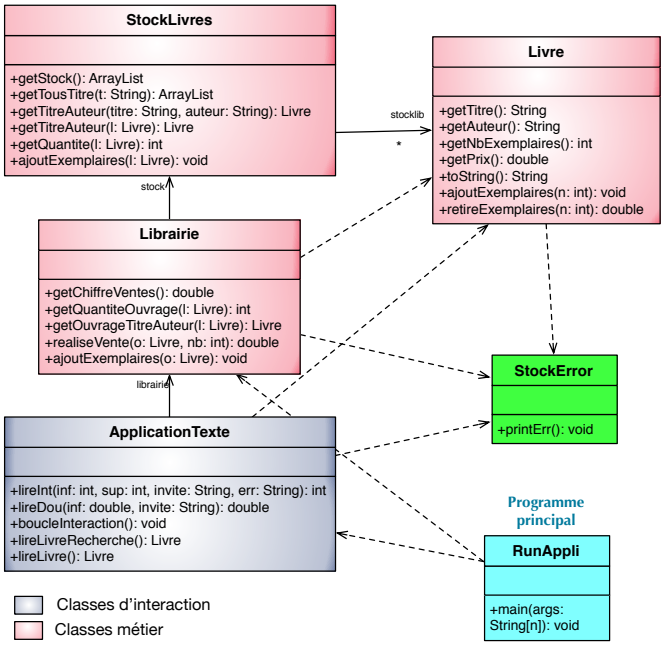

FIGURE 1 – Les classes de l'application librairie

facile de changer le mode d'interaction avec l'utilisateur, par exemple pour changer les E/S clavier par une interface graphique. Il n'y aura alors rien à changer (ou si peu) sur le code des classes métier.

- `ApplicationTexte` : regroupera le code d'interaction, avec des méthodes de saisie et d'affichage, la méthode `boucleInteraction()` pour exécuter la boucle d'opérations proposée à l'utilisateur, et la gestion des erreurs. *Absolument toutes les saisies et affichages seront cloisonnées dans cette classe*.
- `Librairie` : contiendra les opérations métier de la librairie, nécessaires aux appels depuis la boucle d'interactions. Par exemple `getChiffreVentes()` renvoie le chiffre de ventes courant, et `getQuantiteStockOuvrge(l)` la quantité en stock pour un ouvrage, etc. (Voir commentaires dans le code fourni).
- le constructeur de `ApplicationTexte` prend en paramètre un objet `Librairie` (regardez le code de ce constructeur) et l'utilise pour initialiser sa variable interne `librairie`. Cela permet de « lier » les deux couches du code : les méthodes de `ApplicationTexte` pourront faire appel aux méthodes de cet objet `Librairie`.

— `Livre` et `StockLivre` : seront modifiées (par rapport à ce qui aura été fait lors des tps précédents) pour éliminer toute méthode d'entrée/sortie. Ces méthodes pourront maintenant rendre un résultat. Par exemple, la méthode qui recherche un titre et auteur dans le stock n'affichera pas le livre trouvé mais le renverra résultat ; la méthode qui affiche tous les ouvrages d'un auteur sera modifiée pour renvoyer la liste de tous les livres de cet auteur, etc.
— `RunAppli` contient le programme main chargé de lancer la boucle d'opérations.

Question 1 : étude du code fourni

On vous conseille de suivre ces pas :

1. Commencez par regarder le code de la classe Livre qui devrait être proche de votre solution (tp 4). Un changement notable : l'utilisation de l'exception `StockError` expliquée en partie 2 de ce tp.
2. Etudiez maintenant la classe `StockLivre` sur laquelle vous avez déjà travaillé. Essayez de comprendre son code. Si des méthodes sont à compléter/modifier, attendez d'avoir compris qui les utilise (et surtout comment) avant de les changer.
3. Exécutez le programme principal de l'application. Il tourne mais ne fait pas toujours ce que l'on attend. On vous demande votre choix d'opération : entrez un numéro d'opération non valide ; donnez une entrée non numérique. Le programme plante-t-il ? Allez voir le code qui implante la lecture d'une option du menu et expliquez ce comportement.
4. Testez chaque opération du menu :
   — Si elle ne fonctionne pas correctement, essayez de trouver le problème et de corriger le code associé.
   — Si elle fonctionne, remontez la chaîne des méthodes appelées, et regardez s'il n'y pas quelque chose à modifier dans le code.
5. Penchez vous sur la gestion des erreurs : quelles sont les méthodes qui lèvent des exceptions ? Où sont elles traitées ?

2 Gestion des erreurs

Dans sa version basique il y a au moins quatre sortes d'erreurs pouvant survenir au cours des opérations sur une librairie : les erreurs de saisie de nombres, un paramètre passé qui ne peut pas être négatif, un livre non trouvé en stock ou un livre demandé pour un retrait du stock en trop grand nombre d'exemplaires.

1. Les erreurs de saisie lorsqu'un nombre est attendu seront gérées par des méthodes dédiées à la lecture validée d'un entier et d'un double. Leur squelette et spécification se trouve dans la classe `ApplicationTexte`. Commencez par aller voir et éventuellement compléter leur code.
2. Au Tp 5, nous avions utilisé les exceptions `NbExemplairesInsuffisant` pour signaler un nombre d'exemplaires insuffisant en stock et `OuvrageInexistant` si cet ouvrage n'existe pas au catalogue. Nous adopterons une approche qui unifie ces deux erreurs sous le même nom d'exception : `StockErr`. Son code est reproduit plus bas.

Exception `StockErr`

Il est possible d'écrire cette classe de multiples manières. Dans la solution que nous proposons, l'idée est d'englober sous un même nom d'exception toutes les erreurs de traitement sur un stock. Nous incorporons

un code interne d'erreur : 0 si l'échec survient lors d'un ouvrage non trouvé, 1 si la quantité trouvée en stock est insuffisante pour l'opération qui l'a déclenché. Il y a un constructeur différent par code d'erreur et une méthode `printErr()` qui affiche le bon message pour chaque type d'erreur. Etudiez ce code et posez des questions si necéssaire.

```
public class StockError extends RuntimeException {
  private int nbRequis; // nombre demandé (ex: pour une vente)
  private int nbDispo; // nombre trouvé en stock
  private int codeErr; // code de l'erreur
  private String [] err = {"Ouvrage non trouvé au catalogue", "Nb d'exemplaires insuffisant"};


  public StockError( int requis, int dispo) { // Quantité insuffisante
        this.codeErr = 1; this.nbRequis=requis; this.nbDispo=dispo;
  }
  public StockError() { // Ouvrage inexistant
        this.codeErr = 0;
  }
  public void printErr() {
        System.out.println("Stock error: "+ err[codeErr]);
        if (codeErr>0)
              System.out.println(" nombre requis: "+ this.nbRequis+ ", disponible: "+ this.nbDispo);
  }
}
```

Comment utiliser cette exception ?

- Pour lever une exception si ouvrage inexistant : `throw new StockError();`
- Pour lever une exception si ouvrage trouvé en quantité q mais requis en quantité plus grande r : `throw new StockError(r,q);`
- Si on veut traiter l'une au l'autre erreur **uniquement** par l'affichage d'un message qui décrit la nature précise de l'erreur :

```
try {
   // ICI LE CODE POUVANT ECHOUER
} catch ( StockError e){
     e.printErr(); // affiche le message adéquat selon le code interne.
}
```

Un exemple de traitement est donné dans la classe `ApplicationTexte`. Des exemples de levée se trouvent dans plusieurs classes métier. Allez voir où ces exceptions sont utilisées !

Adaptation du code fourni

Suivez les indications (commentaires) dans le code pour compléter ce qui manque. Vous pouvez récupérer du code de vos anciennes classes si cela semble pertinent. Pour la gestion des erreurs vous pouvez soit utiliser l'approche unifiée expliquée plus haut soit continuer avec les exceptions `NbExemplairesInsuffisant` et `OuvrageInexistant` introduites lors du Tp5. Au final on attend d'avoir une application qui réalise (au minimum) toutes les opérations du menu et qui ne plante pas en cas d'erreurs. Vous pouvez également réaliser une application avec gestion de commandes (voir « Exercice optionnel ») mais ceci est optionnel.

Exercice optionnel : pour aller plus loin, gestion de commandes

Une librairie n'a pas toujours en stock le livre que vous cherchez. Ajoutez dans votre application une ou plusieurs classes afin de modéliser les commandes effectuées par les clients, et la gestion de ces commandes : que se passe-t-il si des nouveaux livres rentrent au stock, à quel moment peut-on sortir une commande d'un client de la liste de commandes en attente, dans quels conditions ajoute-t-on une nouvelle commande ?