TP3 : l'application Game

Principaux concepts discutés:

- Conception dirigée par les responsabilités, cohésion, couplage, refactoring

Idée de départ:

Il est possible d'implanter une application comprenant des classes mal conçues, mais qui produit les résultats attendus. Les problèmes apparaissent quand une personne chargée de la maintenance doit modifier l'application.

Dans ce TP nous allons utiliser un exemple qui réalise une implantation rudimentaire d'un jeu d'aventure en mode texte. Dans son état initial le jeu n'est pas très ambitieux : d'un certain point de vue, il n'est pas achevé, mais on va s'en contenter.

1 - L'application Game

L'application **Game** de départ est un framework assez simple pour un jeu d'aventure. Dans cette première version il possède quelques pièces et la possibilité pour un joueur de marcher entre les pièces. C'est tout !

Pour démarrer cette application, il faut créer une instance de la classe Game et appeler la méthode play (c'est le rôle de GameLauncher).

Cette version du jeu contient une très mauvaise conception des classes. Nous allons étudier en quoi sa conception n'est pas satisfaisante.

1.1 – Compréhension de l'application

1. Télécharger l'application game, compiler et lancer le jeu. La méthode main est dans GameLauncher.
2. Faire la rétro-conception du diagramme de classes UML (à la main ou avec un outil par exemple Visual Paradigm-UML).
3. Explorer le code de l'application, les commentaires fournissent quelques informations et répondre aux questions suivantes : Que fait cette application ? Quelles sont les commandes acceptées par le jeu ? Que fait chaque commande ? Combien y-a-t-il de pièces dans ce scénario ? Dessiner une carte des pièces existantes.
4. Identifier le travail réalisé par chaque classe : noter son objectif.

1.2 – Personnalisation du jeu (facultatif si vous avez le temps)

Vous pouvez modifier le jeu en ajouter des pièces.

Modifier la méthode **createRooms** de la classe **Game** pour créer les pièces et sorties que vous avez inventées pour votre jeu; puis tester votre jeu.

2 - Résolution des problèmes de conception

2.1 – Duplication de code

La duplication de code est un indicateur d'une mauvaise conception. Le problème de la duplication de code est que dès qu'une modification est apportée à une version, elle doit être également apportée à l'autre version pour éviter une inconsistance.

- Dans la classe **Game** les méthodes **printWelcome** et **goRooms** contiennent toutes deux des lignes de code identiques. Repérer ces lignes de code et définir une nouvelle méthode que vous pourrez appeler **printLocationInfo** dont la tâche sera d'imprimer la situation courante.
- Tester vos modifications

2.2 – Créations d'extensions et couplage

Nous aimerions faire des extensions dans notre jeu afin de tenir compte des bâtiments à plusieurs niveaux (caves ou donjons, ou autres) et définir les directions haut et bas. Le joueur pourra ainsi aller en bas pour se déplacer dans une cave par exemple.

L'étude des classes fournies met en évidence qu'au moins deux classes sont impliquées dans cette modification : Room et Game. Trouver tous les endroits pertinents dans ces classes demande un peu de travail.

Le fait qu'il y ait tellement d'endroits où toutes les sorties sont énumérées est symptomatique d'une conception maladroite de classes. Dans notre exemple si on remplace les variables de sortie de la classe **Room** et que nous les remplaçons par un objet de type **HashMap**, la classe **Game** ne compilera plus car elle fait de nombreuses références aux variables de sortie. Nous avons affaire à un **couplage fort**.

Cela devrait nous permettre de gérer un nombre quelconque de sorties sans trop de modifications. L'objet de type **HashMap** contiendra un lien entre un nom de direction (par exemple “nord”) et la pièce associées à cette direction(un objet de type Room)

L'un des principaux problèmes ici est l'utilisation d'attributs publics dans la classe **Room**. Cette classe expose dans son interface non seulement le fait qu'elle possède des sorties, mais aussi la manière dont les informations de sortie sont stockées. Elle viole un principe fondamental d'une bonne conception de classe : **l'encapsulation**

Nous avons donc plusieurs modifications à faire pour la classe **Room** : préserver l'encapsulation, permettre l'extensibilité (ajout de nouvelles sortie), prévoir des accesseurs et modificateurs pour une sortie (getExit, setExit)

1- Modifications à effectuer dans la classe **Room** :

- Supprimer les attributs public et les remplacer par un **HashMap** privé, de noms **exits**, tel que décrit plus haut et modifier le constructeur pour créer cet objet.
- Supprimer la méthode **setExits** Ajouter une méthode **getExit (String direction)** qui prend en paramètre une chaîne de caractères représentant la direction et qui retourne un objet

Room (la sortie dans cette direction).

- Ajouter une méthode **setExit(String direction, Room neighbor)** qui prend deux paramètres : une chaîne de caractères pour la direction et un objet de type **Room** (neighbor) qui est la pièce voisine dans cette direction et qui ajoute à l'objet exits de type **HashMap** une entrée.
- On veut que la classe Room soit responsable de préparer les informations concernant les sorties et qui sont ensuite imprimées dans la classe Game par la méthode printLocation. Donc on va définir une méthode dont la signature est : **public String getExitString()** qui renvoie une chaîne de caractères décrivant les sorties de la pièce, par exemple, "Sorties : nord ouest".

2 - Nous devons donc maintenant modifier la classe **Game :**

- Modifier la méthode **createRooms** pour qu'elle utilise le modificateur **setExit** pour initialiser les sorties des pièces.
- Modifier la méthode **printLocationInfo** pour qu'elle utilise la méthode getExitSring() définie dans Room.

Nous avons maintenant complètement supprimé la restriction concernant le stockage de quatre directions seulement pour la classe **Room**. Cette classe est en mesure de stocker les sorties haut et bas, ainsi que toute autre direction à laquelle vous pourriez penser (nord-ouest , etc.).

3- Implanter les modifications de direction dans votre projet.

4- Compiler vos classes et vérifier que l'exécution fonctionne toujours.

5- Eventuellement générer le nouveau diagramme de classe UML.

2.3 - Responsabilités et couplage

La conception et l'interface de la classe Room ont été grandement améliorées. Maintenant les informations concernant les pièces sont gérées par la pièce elle-même.

Nous allons encore améliorer un peu les classes Game et Room pour faciliter l'ajout d'objets dans les pièces. En effet, si nous voulons ajouter des monstres, des objets ou d'autres personnages tout cela doit figurer dans la description de la pièce.

Dans la classe **Room** :

- Améliorer le code de la méthode **getExitString** pour qu'au lieu d'une imbrication de conditionnelles, elle utilise une boucle **for each** sur la collection.

- Ajouter la méthode suivante :

```
public String getLongDescription()
    return "You are" + description+ this.getExitString() ;
}
```

Dans la classe **Game** :

Modifier la méthode **printLocation** pour qu'elle utilise maintenant la méthode **getLongDescription**

2.4 - Extension et couplage implicite

Le **couplage implicite** est une situation dans laquelle une classe dépend de l'information interne d'une autre sans que cette dépendance soit visible de manière évidente.

Supposons que l'on veuille ajouter une commande **look** dont le but est d'imprimer la description de la pièce avec ses sorties (pour savoir où on se trouve).

- Indiquer tous les endroits dans l'application où doivent être faites des modifications pour ajouter la commande **look**.
- Réaliser ces modifications et vérifier que votre application fonctionne correctement.
- Indiquer si vous avez rencontré un problème de couplage implicite : une modification oubliée qui n'empêche pas le programme de fonctionner mais qui cependant constitue un bug.

2.5 - Refactoring

L'interface utilisateur est liée très étroitement avec les noms de commandes écrits en anglais (ou français); par exemple dans la classe **CommandWords** où la liste des commande valides est stockée et dans la classe **Game** où la méthode **processCommand** compare explicitement chaque nom de commande.

Si on veut par exemple permettre de changer de langue il faut trouver tous les endroits dans le code source où des noms de commande sont utilisés. C'est un autre exemple de couplage implicite.

Idéalement il faut trouver une place unique dans le code source où le texte des noms est stocké et faire partout ailleurs des références à des commandes indépendantes de la langue.

Les types énumérés sont une caractéristique des langages de programmation qui rend cela possible : https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Les modifications à faire :

- Créer une nouvelle Enumeration appelée **CommandWord** qui définit simplement les noms des commandes en incluant un nom pour une commande inconnnue (par exemple UNKNOWN)
- Faire les modifications nécessaires dans la classe **CommandWords**.
- Ajouter à la classe **CommandWords** une méthode **getCommandWord** qui prend en paramètre une chaîne de caractères (un nom de commande) et qui retourne un objet **CommandWord**. Pensez à tester le cas d'une commande inconnue.
- Réaliser les modifications nécessaires pour la classe **Game**.

- Modifier également dans la classe **Commmand** la méthode **isUnknown** pour qu'elle utilise `CommandWord.UNKNOWN`
- Compiler vos classes et vérifier que votre jeu fonctionne parfaitement.
- Refaire un diagramme de classes complet de votre application finale.

Maintenant changer de langue pour les noms de commande est plus facile !

Rendu du travail réalisé

1. Déposez sous forme d'un fichier PDF votre travail contenant les diagrammes de classes (si vous les avez faits avec un outil) et les explications sur votre jeu.
2. Déposez une archive contenant vos fichiers sources (.java).