ISEN | école d'ingénieurs

ALL IS DIGITAL!

TP 5: Le château de Ravenduck

11 octobre 2022

Principe et objectif

L'objectif de ce TP est de contruire un petit jeu de A à Z. Il s'agit d'un jeu à deux joueurs de type "stop ou encore". Le jeu se joue en cinq manches. Chaque manche se divise en plusieurs tours. À chaque tour, chaque joueur·euse a le choix de continuer l'aventure ou de s'arrêter. À chaque tour, un certain nombre de changements va avoir lieu dans les ressources des joueuses·eurs. En particulier, elles·ils ont un certain nombre de points de vie, et une quantité d'or qu'elles·ils ont récupéré durant l'aventure. Si un·une joueuse·eur tombe à 0 point de vie, elle·il doit fuir en laissant tout l'or accumulé lors de cette manche, alors qu'elle·met en sécurité son or si elle·il s'arrête, mais elle·il perd alors toute opportunité de gagner plus d'or, alors que son·sa comparse peut continuer à en accumuler.

Interface graphique

L'interface graphique du jeu ressemble à cela :

Le but n'est pas nécessairement de reproduire cette fenêtre exactement, mais de réussir à :

— changer la police de caractère du titre (et sa taille),
— diviser l'écran en plusieurs parties,
— changer l'icône de l'application afin de mettre l'image du (terrible) Comte Ravenduck.

Les images dont vous aurez besoin sont sur Moodle.

Pour placer ces éléments, créez une classe `GameWindow`, héritant de `JFrame`. Vous placerez cette classe et toutes les classes liées à l'interface graphique dans un package `graphics`.

Pour mettre une image en arrière plan, au début du constructeur de la `GameWindow`, vous pouvez utiliser le code suivant (en remplaçant `path/to/img` par le chemin jusqu'à l'image) :

```
1  super();
2  try {
3      final Image backgroundImage = javax.imageio.ImageIO.read(
4          new File("path/to/img/Ravenloft_icon.png"));
5      setContentPane(new JPanel(new BorderLayout()) {
6          @Override public void paintComponent(Graphics g) {
7              g.drawImage(backgroundImage, 0, 0, null);
8          }
9      });
10 } catch (IOException e) {
11     throw new RuntimeException(e);
12 }
```

Nous laissons de côté les explications sur ce code. Il permet de créer une classe « à la volée » qui étende `JPanel` et qui change la fonction `paintComponent`.

⚠ Nota Bene

Pour organiser la fenêtre, prenez le temps de la diviser. N'oubliez pas qu'il est possible d'imbriquer des `Layout` dans des `Layout` en utilisant des `JPanel`.

Nous nous intéresserons un peu plus tard au comportement des composants de cette fenêtre, mais nous allons pour le moment laisser l'aspect graphique et nous intéresser à la logique de notre programme.

Logique

Nous allons maintenant nous intéresser à la logique du jeu. Pour cela, nous allons créer une nouvelle classe, `GameManager`, que nous placerons dans le dossier `logics`. Celle-ci sera en charge de gérer les différents processus du jeu. Il ne doit exister qu'une unique instance de cette classe, nous allons donc utiliser un singleton.

⚠ Nota Bene

Pour rappel, voici les modifications à effectuer pour créer un singleton :

1. rendre le constructeur `private` afin qu'il ne soit pas possible d'accéder au constructeur par défaut ;
2. créer une instance de la classe, qui sera la seule instance de cette classe. Cette instance doit être statique, afin qu'il soit possible d'y accéder dans les méthodes statiques de la classe `private GameManager instance;` ;
3. créer une méthode `public static GameManager getInstance();`. Cette méthode renverra l'instance créée précédemment. Il s'agit d'une méthode `static`, ce qui signifie qu'il est possible d'y accéder depuis l'extérieur de la classe en l'appelant directement sur la classe : `GameManager.getInstance();` ;
4. Dans cette application, l'initialisation ne se fera pas à travers la méthode `getInstance` (*lazy initialization*). On utilisera donc une méthode spécifique `public static void generateInstance()` à laquelle on donnera les arguments dont le constructeur aura besoin.

En plus de cette classe, on crée une classe `Player` qui permet de stocker les informations de chaque joueuse·eur. On a besoin des informations suivantes : son nombre de points de vie (`int`), la quantité d'or obtenur dans cette manche (`int`), la quantité d'or sauvegardée jusqu'à présent (`int`), la ma,che à laquelle on est (`int`) et enfin le nombre de PV maximal de la joueuse·eur (`int`). Cette classe sera placée dans le package `logics`.

À sa création, le `GameManager` doit créer deux `Player`. En plus de cela, c'est le `GameManager` qui va gérer les évènements qui vont arriver aux joueuses·eurs. Pour cela, on va représenter les évènements sous la forme d'une classe qui comporte 3 attributs : le nombre de points de vie modifiés (`int`) par l'événement, la quantité d'or modifée (`int`) et le texte affiché aux joueurs·euses (`String`). Pour cela, on crée une nouvelle classe. Pour éviter les confusions avec la classe `java.awt.Event`, on appelle cette classe `Story`. Cette classe sera elle aussi placée dans le package `logics`. L'ensemble des évènements possible doit être accessible à l'instance de `GameManager` et lui est passée à la création sous la forme d'une liste de `Story`. Nous verrons plus tard comment peupler cette liste.

Nous allons maintenant passer à la logique du programme. La principale fonction de la classe `GameManager` est la fonction `public void play()` cette fonction a pour rôle de tirer une `Story` aléatoirement depuis la liste des événements possibles, puis d'en appliquer la logiques aux joueuses·eurs. Elle addtionne le nombre de points de vie et d'or à la réserve de chacun·e.

⚠ Nota Bene

On arrive à un problème si l'un·e des joueuses·eurs est rentré, qu'elle·il ait dû fuir ou ait décidé de rentrer. Pour prendre en compte cet aspect, on va ajouter un attribut aux joueuses·eurs qui indiquera si elle·il est encore dans la partie. Puisque l'attribut ne peut prendre que deux valeurs, on utilisera un `boolean`.

La fonction `play` devra aussi vérifier si l'un ·e des deux joueuses·eurs arrive à 0 point de vie ou moins. Dans ce cas, il ou elle devient inactif·ve avant que l'acquisition de l'or ne soit fait.

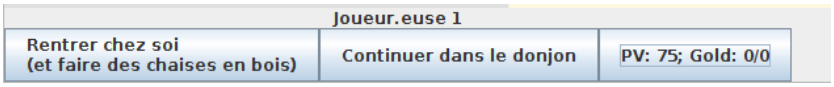
On souhaite que le tirage des histoires se fasse de manière à ce que chaque histoire ait été tirée une fois avant qu'une `Story` ayant déjà été tirée ne le soit une seconde fois. De plus, si les deux `Player` sont désactivé·e ·s, le `GameManager` doit être réinitialisé : on appelle pour cela une méthode particulière, `reinitialize`.

La méthode `reinitialize` permet de démarrer une nouvelle manche du jeu. On augmentera donc la manche à laquelle on est. Si un·e `Player` a encore des points de vie, on transfère son or vers l'or sauvegardé.

La partie logique n'est pas tout à fait terminée, mais nous la laissons de côté pour le moment, nous poursuivrons quand nous nous intéresserons à la liaison entre la partie graphique et la partie logique, dans la partie suivante.

Liaison entre graphique et logique : les listeners

La partie graphique contient deux fois trois boutons, permettant de faire trois actions pour chacun·e des `Player`. Nous allons donc définir trois classes héritantant de `ActionListener` pour chacune de ces actions. On leur donnera définira un attribut `private Player player` définissant quel `Player` est concerné par ce bouton. Les boutons permettent aux joueuses·eurs de choisir une action, à l'exception du dernier qui permet d'afficher le score d'un·e joueur·euse à la place du texte sur le bouton (voir la figure ci-dessous). En recliquant sur le bouton, on recachera le résultat.



Commencez par ce listener. Réfléchissez à ce dont il aura besoin : on doit être capable de savoir dans quel état est le bouton (activé ou non), il doit avoir accès au bon `Player`, afin d'afficher les bonnes informations. Enfin, il doit avoir accès au bouton pour en changer le label.

Passons maintenant aux deux autres boutons.Conceptuellement, le but est de déclencher la fonction `play` du `GameManager`, mais pour cela, il faut soit que tout le monde ait joué, soit qu'un·e `Player` soit inactif·ve. On va donc devoir garder en mémoire les actions faites jusqu'à ce que deux boutons de `Player` différents aient été cliqués. Pour gérer cela, on va ajouter un attribut à la classe `Player`. Cet attribut enregistrera le fait qu'on a cliqué sur un bouton pour ce·tte `Player` et quelle action a été choisie. Il doit donc pouvoir prendre trois valeurs : non choisi, continuer ou arrêter. Pour cela, on pourrait utiliser un `int`, mais cela ne correspond pas vraiment à notre besoin. On veut avoir un attribut qui ne peut prendre que trois valeurs. Pour ce genre de besoin, on utilise généralement une énumération, ou `enum`.

⚠ Nota Bene

Conceptuellement, un `enum` ressemble un peu à un singleton (il est d'ailleurs possible d'utiliser un `enum` avec une seule valeur pour implémenter un singleton), mais au lieu de n'avoir qu'une seule instance, on en a un nombre prédéfini. Une autre différence est que l'`enum` donne accès à ses instances *via* des attributs statiques, et non une méthode statique. Pour créer une `enum`, on procède comme pour la création d'une classe, mais on remplace le mot-clé `class` par le mot-clé `enum`. On donne ensuite, séparés par une virgule, le nom de chaque instance. Notez que chaque instance se voit aussi dotée d'un ordinal, un entier qui indique son ordre dans la liste. Chaque instance est unique, on peut donc utiliser directement `==`, il n'est pas nécessaire d'utiliser la méthode `equals` comme pour les `String` par exemple, car on compare bien le même objet.

Créez un `enum`, que vous nommerez `Action`, et qui sera dans le même package que `Player`. Cette énumération aura trois instances : `NOT_SELECTED`, `CONTINUE` et `STOP`. Créez ensuite pour chaque `Player` un attribut `Action action`. À l'initialisation de `Player`, `action` vaut `Action.NOT_SELECTED`. Quand un des boutons est cliqué, le listener doit maintenant vérifier l'état des `action` des players et déclencher la méthode `play`. Il faudra modifier cette dernière pour qu'elle gère ce nouvel attribut et agisse en conséquence. Une fois que la fonction a eu lieu ; faites en sorte que l'interface graphique se mette aussi à jour (vous devrez aussi modifier le `GameManager`).

Désormais, tout le jeu est fonctionnel, mais il manque un main, et l'accès aux `Story`...

La base de données et la fonction principale

Les `story` se trouvent dans une base de données. En utilisant ce qui a été fait dans le TP précédent, accédez à la base `10.10.57.5/ravenduck`. Les `Story` se trouvent dans la table `stories`. Chaque ligne est composée d'un `int`, qui ne sert que de clé primaire et que vous pouvez laisser de côté, puis d'une description sous forme de `String`, de la modification de points de vie sous forme d'un `int` et de la modification d'or sous forme d'un `int`.

Créez enfin le programme principal, qui s'occupera d'orchestrer la base de données, la création du GameManager et la création de la fenêtre.

Si vous êtes arrivé·e ici, vous avez révisé la majorité des concepts importants qui serviront pour le TP noté (prenez le temps de réviser aussi les exceptions, absentes de ce TP). Bravo à vous ! La suite du TP permet néanmoins d'aborder de nouvelles notions, qui pouyrront vous servir dans de futurs cours et dans votre vie professionnelle.

Les tests et JUnit

Nous n'avons pour le moment testé notre code qu'au travers d'exécutions. Sur de petites applications comme celles que nous avons créées, cela peut suffire, mais dès qu'on arrive sur des applications plus importantes, il est nécessaire de tester le code qu'on écrit de manière plus systématique. Pour cela, on va utiliser un `framework` de test unitaire, JUnit.

> **Nota Bene**
>
> Un test unitaire permet de tester une méthode dans une configuration contrôlée. Cela permet notamment de voir si la sortie est bien toujours la même après qu'on a modifié le code par exemple pour y ajouter une fonctionnalité.

Pratiquement, commencez par faire un click droit sur la racine de votre projet, puis faites "New", puis "directory". Nommez ce nouveau dossier `test`. Puis, faites un click droit sur ce nouveau dossier, "Mark directory as" "test source root". Le dossier s'affiche maintenant en vert.

Ajouter JUnit

Pour ajouter JUnit, procédez comme vous l'aviez fait pour ajouter mysql, mais recherchez `org.junit.jupiter:junit-jupiter:5.9.1` (ou une version supérieure). Récupérez aussi mockito, dont nous aurons besoin plus tard `org.mockito:mockito-all:2.0.2-beta`.

Votre premier test

Rendez-vous maintenant dans GameManager, et faites un alt+Entrée (ou Optn+Entrée sous Mac) sur la classe. La première option vous proposera d'écrire une classe test pour GameManager. Choisissez cette option. Cela génère une class `GameManagerTest` qui est placée directement dans le bon package, mais dans le dossier `test`. Choisissez de tester la méthode `play`. Nous allons devoir reproduire un `GameManager` minimal mais fonctionnel afin de tester la méthode `play`. Créez une liste de `Story` avec une unique story à l'intérieur. Inutile de récupérer les `Story` de la base de données. On cherche ici à tester la fonction `play`, le fait d'aller chercher en base risque de créer des interférences (en cas de test non valide, d'où vient le problème etc.). Un test ne doit tester d'une méthode à la fois, ainsi, en cas d'échec, on sait où investiguer. Générez ensuite une instance de `GameManager`, et faites en sorte qu'un·e `Player` continue et que l'autre arrête. En fonction de la Story que vous avez créée, utilisez les méthodes `Assertion.assertEquals` pour vérifier que les valeurs sont bien celles qui sont attendues pour chacun des deux `Player` après exécution du code. Au niveau du lancement du code, vous devez maintenant pouvoir lancer le test JUnit, faites-le...

Vous obtenez une erreur. Et pour cause, la fonction `play` appelle aussi de la mise à jour de l'interface graphique, et celle-ci n'a pas été définie. Comme pour la base de données, le but n'est pas ici de tester le composant graphique (on ne fait pas cela dans les tests unitaires). On va donc fournir au `GameManager` une fausse `GameWindow`, un `Mock`. Cette instance de `GameWindow` aura toutes les méthodes de GameWindow, mais ces méthodes seront simplement sans effet. Au début de votre méthode, ajoutez la ligne suivante :

```
1 GameWindow mockedWindow = mock(GameWindow.class);
```

en important la méthode statique `mock` depuis Mockito. Passez ce `mockedWindow` à votre GameManager comme dans le `main` et relancez le test. Celui-ci devrait passer. Observez ce qui se passe si vous modifiez une des valeurs attendues.

⚠ Nota Bene

Normalement, autant que faire se peut, tout code qui contient un peu de logique doit être testé (on ne teste généralement pas les getters et les setters, ni le constructeur). On appelle la proportion du code testé la couverture de test. Idéalement, toute condition doit être testée dans le positif et le négatif, afin de s'assurer que le code ne va pas lever une exception à son exécution. Certains outils permettent de calculer la couverture du code, comme *SonarQube*. Les tests peuvent aussi être exécutés de manière automatique quand on veut ajouter du code à l'application, empêchant l'ajout si cela casse un test (on appalle alors ces tests des tests de non régression). Ces outils (on parle de continuous integration) permettent un meilleur partage du travail entre collègues.
Dans certains cas, on commencera même par définir le contrat de la méthode, puis avant de l'écrire, on écrira des tests à laquelle la méthode doit se conformer, on complexifiera les tests peu à peu en même temps qu'on écrira la méthode, afin de s'assurer que tous les cas sont couverts. Cette méthodologie de développement s'intitule le *Test Driven Development*.

Si vous êtes arrivé·e jusqu'ici, c'est que vous êtes prêt·e à passer du côté obscur de la force…Jusqu'à présent, nous avons défini de nombreuses bonnes pratiques. Je vous propose maintenant d'aller lire un petit tutoriel sur l'introspection et le réflexion, deux concepts qui sont généralement considérées comme mauvaises pratiques lorsqu'on peut les éviter car elles passent outre les vérifications du compilateur, laissant de nombreux risques d'exceptions à l'exécution ; Il existe cependant certains cas où il est impossible de faire sans. Je n'ai pas eu le temps de faire un TP bonus dédié, mais je vous recommande le cours de Jean-Michel Doudoux sur Développez.