Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2020 – 2021
DEV2
mcd

DEV2 – Laboratoire Java

Projet - Lucky Numbers

Partie 2

Table des matières

Version du document
1er avril 2021

Modalités pratiques

Échéances

Nous sommes strict · es sur les échéances. Prenez la précaution de vérifier auprès de votre enseignant · e des modalités spécifiques de remises.

Date limite de remise : le vendredi 30 avril à 18h

Dépôt git

Nous vous rappelons que l'utilisation de git tout au long du développement du projet est obligatoire. Vous devez faire des **commits** réguliers et au minimum **à chaque fois que c'est demandé** dans l'énoncé faute de quoi votre travail ne sera pas évalué.

Pondération

Cette seconde partie compte pour la moitié dans la cote du projet. Toutefois, une cote définitive ne vous sera attribuée qu'après la défense individuelle du projet.

Amélioration de l'itération 1

Dès qu'il aura terminé de corriger votre itération 1, votre professeur vous communiquera ses remarques sur votre travail. On attend de vous que vous corrigiez votre itération 1 en fonction de ses remarques et que vous suiviez ses conseils pour ne plus reproduire les mêmes erreurs dans l'itération 2.

Ceci interviendra en partie dans la cote de l'itération 2.

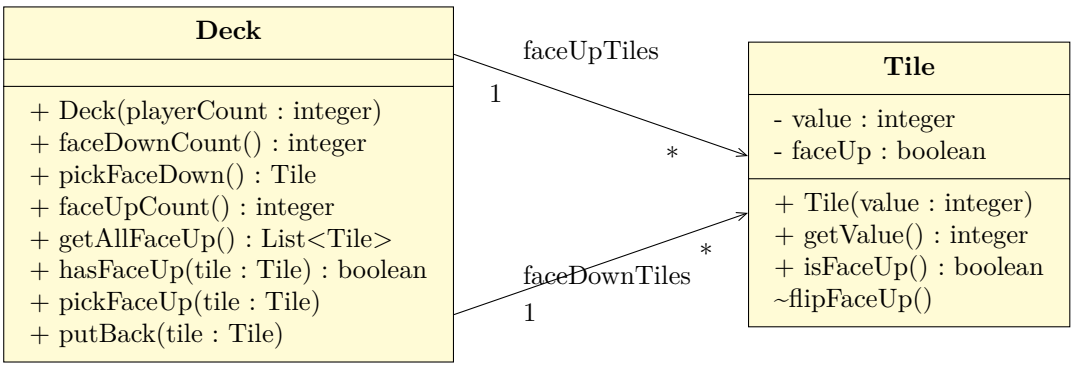
Itération 2

Pour cette seconde itération, nous allons introduire les règles du jeu qui avaient été laissées de coté dans l'itération 1.

1. Introduction d'une pioche avec toutes les tuiles au départ, face cachée.
2. Si une tuile est remplacée par une autre, elle revient au milieu de la table, face visible.
3. Le joueur peut décider de ne pas poser une tuile prise dans la pioche des tuiles non visibles. Elle retourne alors au milieu de la table, face visible.
4. Le joueur peut prendre une tuile disponible face visible à la place d'une tuile face cachée.
5. Prise en compte des règles complètes pour la fin de la partie.
6. Prise en compte du début de partie (tuiles dans la diagonale).

1 v1.1 – La pioche

Commençons par introduire la pioche qui va comprendre toutes les tuiles disponibles qu'elles soient face visible ou non.



1.1 Visibilité des tuiles

Modifions la classe tuile pour retenir si elle est face visible ou face cachée.

État.

- ▷ L'attribut `faceUp` retient si elle est face visible ou pas. Au départ, une tuile n'est pas visible.

Comportement.

- ▷ `isFaceUp` est l'accesseur du nouvel attribut.
- ▷ `flipFaceUp` rend une tuile visible. Rien ne se passe si elle est déjà visible. Comme indiqué dans le diagramme UML elle peut être de visibilité package car elle ne doit pas être utilisée par la vue.

Tests. Écrivez deux tests unitaires pour la méthode `flipFaceUp()`.

Il est temps de faire un commit (`v1.1 - tile's visibility`) et d'envoyer sur le dépôt.

1.2 Le `Deck`

Cette classe contient toutes les tuiles disponibles au centre de la table, qu'elles soient face visible ou pas.

État. Nous vous proposons de retenir les tuiles dans 2 listes différentes [1] : une première pour les tuiles face cachée, une seconde pour les tuiles face visible.

Comportement.

- ▷ Le constructeur crée une pioche initiale comme indiqué dans les règles. On y trouve toutes les tuiles de valeur 1 à 20 en autant d'exemplaires que le nombre de joueurs, toutes face cachée. Vous pouvez décider de les mélanger ici **ou** d'écrire la méthode `pickFaceDown()` de telle sorte qu'elle en prenne une au hasard.
- ▷ `faceDownCount` : donne le nombre de cartes face cachée dans la pioche.
- ▷ `pickFaceDown` : retire une tuile face cachée et la retourne. Si la liste des tuiles cachées a été mélangée dans le constructeur vous pouvez retourner la dernière, sinon vous devez en choisir une au hasard.
- ▷ `faceUpCount` : donne le nombre de cartes face visible dans la pioche.
- ▷ `getAllFaceUp` : retourne la liste des tuiles face visible.
- ▷ `hasFaceUp` : vérifie si la tuile en paramètre existe et est visible dans la pioche. Elle pourra vous être utile dans votre interaction avec le joueur.
- ▷ `pickFaceUp` : retire de la pioche la tuile indiquée.
- ▷ `putBack` : replace dans la pioche, face visible, la tuile donnée.

Pas de vérification des paramètres

Comme lors de l'itération 1, nous ne vous demandons pas de vérifier les paramètres ici ; ce sera fait dans la classe `Game`.

Tests unitaires. On vous demande de tester correctement cette classe. Pour vous aider, vous pouvez ajouter une méthode package qui donne la liste des tuiles cachées.

Il est temps de faire un commit (`v1.1 - add deck`) et d'envoyer sur le dépôt.

2 v1.2 – Intégration de la pioche dans le jeu

Nous allons maintenant intégrer cette pioche au jeu.

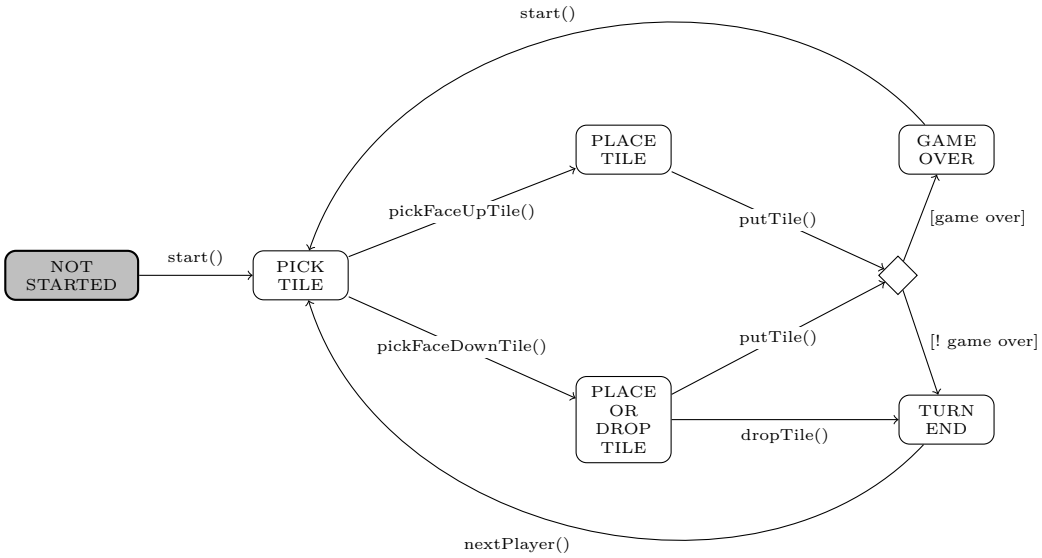
- ▷ Le joueur pourra choisir de prendre une tuile face cachée au hasard ou bien une des tuiles face visible.

1. En procédant ainsi, l'attribut de visibilité d'une tuile n'est pas absolument nécessaire mais gardons-le si vous le permettez. Alternativement, vous pouvez aussi décider de stocker toutes les tuiles dans **une seule** liste et écrire les méthodes en conséquence.

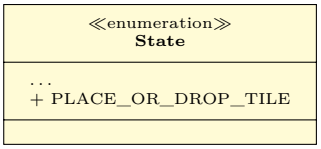
▷ S'il a pris une tuile face cachée, il peut décider de la reposer dans la pioche, face visible plutôt que de la poser sur son plateau.

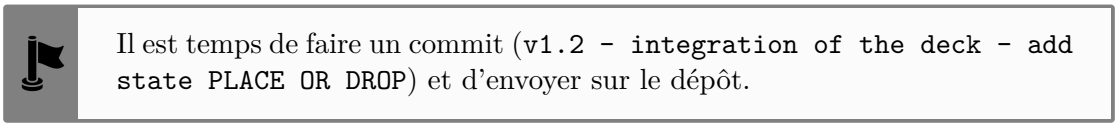
2.1 Un nouvel état

Nous devons introduire un nouvel état qui retient si l'utilisateur a choisi une tuile face cachée ou face visible car ses actions futures en dépendent. Voici le nouveau diagramme d'état.



Voici donc la nouvelle énumération où les ... indiquent ce qui est repris tel quel de l'itération 1.

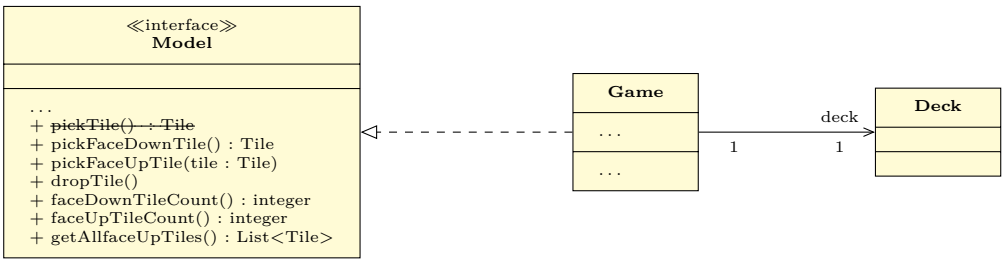




Il est temps de faire un commit (`v1.2 - integration of the deck - add state PLACE OR DROP`) et d'envoyer sur le dépôt.

2.2 Intégration dans le jeu

Il faut à présent intégrer ces nouveautés au modèle.



Le modèle

Dans l'interface `Model` :

- ▷ `pickTile` disparait au profit de `pickFaceDownTile` et `pickFaceDownTile`.
- ▷ `pickFaceDownTile` choisit une tuile face cachée au hasard.
- ▷ `pickFaceUpTile` choisit la tuile face visible donnée.
- ▷ `dropTile` replace la tuile précédemment choisie dans la pioche, face visible.
- ▷ `faceDownTileCount` donne le nombre de tuiles face cachée.
- ▷ `faceUpTileCount` donne le nombre de tuiles face visible.
- ▷ `getAllfaceUpTiles` donne la liste des tuiles face visible.

On vous demande bien sûr de soigner la documentation de ces nouvelles méthodes et d'adapter, si nécessaire, la documentation des anciennes méthodes.

Le jeu

Dans la classe `Game` vous devez :

- ▷ Introduire un nouvel attribut correspondant au deck.
- ▷ Implémenter les nouvelles méthodes du modèle. Elles devront bien sûr faire toutes les vérifications nécessaires sur les paramètres et sur l'état du jeu.
- ▷ Pour `getAllfaceUpTiles` faites attention. Si vous vous contentez de retourner l'attribut, vous donnez accès à un attribut privé. Un code externe pourrait (par inadvertance ou par malice) modifier cette liste. On vous suggère de regarder ce que la classe `Collections` vous propose pour éviter ce problème.
- ▷ Vous serez amené à modifier des méthodes existantes. Par exemple, lorsqu'on pose une tuile sur une case occupée, la tuile qui s'y trouvait n'est plus écrasée mais placée face visible sur la table.

On vous demande d'écrire les tests unitaires correspondants (et d'adapter les précédents si nécessaire).

Il est temps de faire un commit (`v1.2 - update Game`) et d'envoyer sur le dépôt.

2.3 Adaptation de la vue

Au niveau de la vue :

- ▷ Adaptez l'affichage du jeu pour que les tuiles disponibles soient affichées lorsque vient le moment d'en choisir une. Comme dans la version électronique du jeu sur BGA, on vous propose d'afficher le nombre de tuiles restantes face cachée et le détail des tuiles visibles.
- ▷ Vous introduirez également de quoi permettre au joueur d'indiquer s'il veut piocher une tuile face cachée ou face visible (et laquelle dans ce dernier cas).

Il est temps de faire un commit (`v1.2 - update View`) et d'envoyer sur le dépôt.

2.4 Adaptation du contrôleur

Vous pouvez maintenant adapter le contrôleur pour qu'il prenne en compte les nouveaux éléments introduits.

Arrivé à ce stade, votre jeu doit être tout-à-fait fonctionnel.

Il est temps de faire un commit (`v1.2 - update Controller`) et d'envoyer sur le dépôt.

3 v1.3 – La fin du jeu

Pour le moment, nous considérons que la partie est finie lorsqu'un joueur remplit sa grille ; il est alors déclaré vainqueur.

La règle officielle stipule également un autre cas de fin de partie. On vous laisse l'implémenter sans vous guider. On pense que vous êtes à présent capable de le faire seul et proprement.

Modification du modèle

Puisqu'il peut maintenant y avoir plus d'un vainqueur, nous vous proposons d'ajouter un `s` à la méthode `getWinner` du modèle. C'est normalement le seul changement à apporter à cette interface.
Dans l'implémentation, vous serez probablement amener à ajouter des méthodes publiques dans les classes utilisées par `Game`.

Il est temps de faire un commit (`v1.3 - full rules for end of the game`) et d'envoyer sur le dépôt.

4 v1.4 – Le début du jeu

Nous n'avons toujours pas intégré la particularité du début de partie, à savoir la pose des tuiles sur la diagonale. On vous propose d'intégrer cet aspect en choisissant **une** de ces propositions. Nous les avons classées dans l'ordre de difficulté d'implémentation (la plus simple d'abord).

1. **Version en ligne sur Board Game Arena** : Au début du jeu, chacun reçoit 4 tuiles qui sont placées automatiquement sur la diagonale dans l'ordre croissant.
2. **Variante officielle dite « Mise en place de Michael »**.
3. **Version officielle** : Au début, chacun reçoit 4 tuiles qu'il place manuellement.

On vous demande bien sûr d'écrire les tests unitaires correspondants. Vous serez d'ailleurs probablement amené à modifier certains tests existants.

Expliciter vos choix

Pour pouvoir évaluer correctement votre travail, nous devons savoir quelle variante vous avez codé. Nous vous demandons d'ajouter un fichier `README.md` à la racine de votre dépôt avec :

- ▷ La variante choisie.
- ▷ Quelques lignes d'explications sur comment vous vous y êtes pris, sur les modifications que vous avez du apporter au modèle...

⚠ Édition du README.md directement sur gitlab

Il est possible de créer et de modifier directement ce fichier via le site GitLab for ESI. Attention toutefois que ceci crée un commit. Si vous le faites :

1. Veillez à bien commiter et pusher votre projet avant.
2. Après avoir édité le fichier sur GitLab, il est important de faire un `pull` en local afin de récupérer le commit contenant la nouvelle version du `README.md`.

Il est temps de faire un commit et d'envoyer sur le dépôt.

5 v1.5 - Et j'en fais un beau `jar`

Dès lors qu'un programme est terminé, c'est mieux de pouvoir le proposer aux utilisateurs et utilisatrices sans leur demander d'installer Netbeans — ou un autre IDE —.

Nous voudrions leur fournir un fichier *jar* qu'il suffit d'exécuter [2].

`jar` *file*

Un fichier *jar* est une archive au format *zip* — vous pouvez le *dézipper* et regarder son contenu — à l'intérieur de laquelle se trouve tout ce qui est nécessaire à l'éxécution de votre programme ; les fichiers `class` et un *manifeste*.

Le *manifeste* est le fichier `MANIFEST.MF` qui est le point d'entrée de votre archive `jar`. Il contient, par exemple, le nom de la classe principale. Celle qui doit être exécutée lors du lancement du programme.
Un tel fichier peut avoir cette allure :

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: mcd
Build-Jdk: 11.0.6
Main-Class: mcd.luckynumbers.LuckyNumbers
```

Nous voulons que se trouvent dans le fichier `jar` :

- ▷ les fichiers `class` de notre application ;
- ▷ un *manifeste* précisant quelle est la classe principale.

2. Il faudra bien sûr une machine virtuelle Java (*jvm*).

La configuration par défaut de Maven sous Netbeans ajoute les fichiers `class` à l'archive `jar` ce qui règle le premier point.

Si Netbeans peut retenir quelle est la classe principale à lancer lorsque l'on clique « sur la petite icône en forme de triangle vert », il ne transmet pas cette information à Maven.

Pour régler ce dernier point, il faut *mettre les mains dans son `POM`*.

POM, *project object model*

Le *pom* est un fichier `xml` représentant la structure de projet. Ce fichier est lu par Maven pour construire les différentes cibles ; compilation, construction (*build*)...

Vous pouvez trouver le fichier `pom.xml` sous Netbeans dans la partie `Projet Files`.

Pour mettre à jour le manifeste et y ajouter le nom de la classe principale à exécuter, il faut y mettre ceci [3] :

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-jar-plugin</artifactId>
         <version>3.1.0</version>
         <configuration>
            <archive>
               <manifest>
                  <addClasspath>true</addClasspath>
                  <classpathPrefix>lib/</classpathPrefix>
                  <mainClass>le nom qualifié de la classe principale</mainClass>
               </manifest>
            </archive>
         </configuration>
      </plugin>
   </plugins>
</build>
```

Avec cette configuration, un *clean and build* du projet devrait donner un `jar` dans le répertoire `target`. Ce jar pourrait être « distribué ». Pour vous en convaincre, copiez uniquement ce `jar` sur une autre machine ou dans un autre répertoire et exécutez-le avec la commande :

```
java -jar <nom-du-jar>.jar
```

6 v2.0 - La touche finale

Ceci termine votre projet. Félicitations !

Avant la remise définitive, Repassez sur tout le code et vérifiez qu'il soit correctement documenté et le plus lisible possible. Revérifiez aussi une dernière fois que tous les tests passent toujours !

Lorsque vous êtes satisfait de votre code, faites un dernier *commit* avec un message explicite suivi d'un *push*.

3. En adaptant bien sûr le nom qualifié (avec le package) de la classe principale.