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

2020 – 2021
DEV2
mcd

DEV2 – Laboratoire Java

Projet - Lucky Numbers

Partie 1

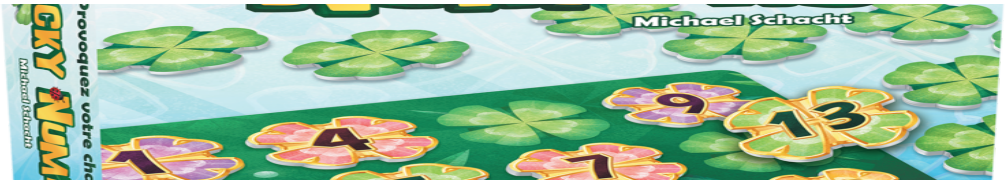


Table des matières

Ne paniquez pas ! Si ce document est si long c'est parce qu'on vous aide beaucoup ;)

Version du document
12 mars 2021

Présentation du projet

Avant de commencer à coder, veuillez prendre connaissance des informations suivantes.

1 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 2 avril à 18h

Dépôt git

Nous vous avons créé le dépôt `dev2-etd/2021/projet/lucky-numbers-xxxxx` pour votre projet. C'est avec **ce dépôt** que vous travaillerez et remettrez votre travail à l'issue des différentes échéances.

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.

Évaluation

Chaque personne développant son projet est invitée à :

- documenter tout son code ;
- écrire des tests JUnit lorsque c'est demandé ;
- utiliser les éléments du langage Java les plus adaptés à la situation (un *for-each* quand c'est possible par exemple) ;
- soigner la lisibilité.

Une méthode qui n'est pas documentée ou qui n'est pas accompagnée de tests unitaires alors que c'est explicitement demandé ne **sera pas évaluée**.

Tout problème de lisibilité (mauvais choix de noms, mauvaise indentation…) ou d'utilisation inadéquate des fonctionnalités du langage sera **sanctionné**.

Pondération

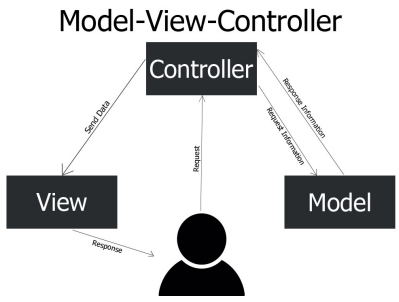
Cette première partie compte pour la moitié dans la cote du projet. Toutefois, une cote définitive ne vous sera attribuée qu'après la défense individuelle du projet.

⚠ Coopération vs tricherie

Le projet est un **travail individuel**. Il vous est interdit de copier en tout ou en partie le travail d'un autre étudiant. En cas de copie manifeste, le **copieur et le copié seront sanctionnés**. Toutefois, nous encourageons la collaboration : des échanges sur la compréhension de l'énoncé, sur les algorithmes à mettre en œuvre, la comparaison de pratiques, l'aide au débogage.

2 Architecture de l'application

Le patron de conception (*design pattern*) Modèle-Vue-Contrôleur (**MVC**) est une manière de structurer le code adaptée pour la programmation d'applications avec interaction utilisateur. On y distingue la partie métier (modèle et contrôleur) de la partie de présentation (vue) de l'application. Le patron modèle-vue-contrôleur (en abrégé MVC, de l'anglais *model-view-controller*) est un modèle destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants au sein de leur architecture respective (voir Wikipedia [1]).



Ce paradigme regroupe les fonctions nécessaires en trois catégories :

- ▷ un modèle : modèle de données et code métier ;
- ▷ une vue : présentation et interaction avec l'utilisateur ;
- ▷ un contrôleur : logique de contrôle, gestion des événements.

La partie modèle (*model*) contiendra pour nous les classes qui définissent les éléments ainsi que la logique principale de l'application. Ces classes seront regroupées dans un package spécifique : `g12345.luckynumbers.model` [2].

La partie vue (*view*) concerne les classes qui s'occupent de la présentation et de l'interaction avec l'utilisateur. Elles seront également regroupées dans leur propre package : `g12345.luckynumbers.view`.

La partie dynamique du jeu (*controller*) sera contenue dans le package : `g12345.luckynumbers.controller`.

Pour finir, **la classe principale** chargée de lancer le jeu sera contenue dans le package `g12345.luckynumbers`.

1. `http://fr.wikipedia.org/wiki/Modèle-vue-contrôleur` *consulté le 30 janvier 20020*
2. Durant ce travail, `g12345` représente (évidemment) votre matricule.

Itération 1

Pour cette première itération, nous allons coder une version simplifiée du jeu.

1. Il n'y a **pas de tas de tuiles-trèfles** sur la table. Lorsque c'est à son tour de jouer, le joueur reçoit une nouvelle tuile avec une valeur aléatoire entre 1 et 20.
2. Le joueur ne peut **que placer** la tuile sur son plateau. S'il la place sur une case occupée, elle remplace la précédente qui est perdue.
3. Nous ne gérons **pas les diagonales**. Au début du jeu, le plateau est vide et le joueur peut placer ses tuiles où il le veut (en respectant les règles d'ordre bien sûr).
4. Le premier à **remplir sa grille** est déclaré **vainqueur**.

3 v0.1 – Les éléments du jeu

Commençons par définir les différents éléments intervenant dans le jeu : la notion de tuile et de plateau de jeu. Toutes les classes de cette section font partie du modèle ; elles doivent donc être placées dans le package `...model`.

3.1 Les tuiles-trèfles

Tile
- value : integer
+ Tile(value : integer) + getValue() : integer

État

- ▷ Nous retiendrons la valeur de la tuile (1 à 20).
- ▷ Nous ne retiendrons pas pour l'instant (et peut-être jamais[3]) la couleur de la tuile.
- ▷ À ce stade de notre développement, il n'est pas nécessaire de savoir si la tuile est visible ou retournée car elles sont toutes visibles.

Comportement

- ▷ Nous n'avons besoin pour l'instant que d'un constructeur et de l'accesseur.
- ▷ Ne prévoyons pas de mutateur, la valeur d'une tuile ne change jamais.

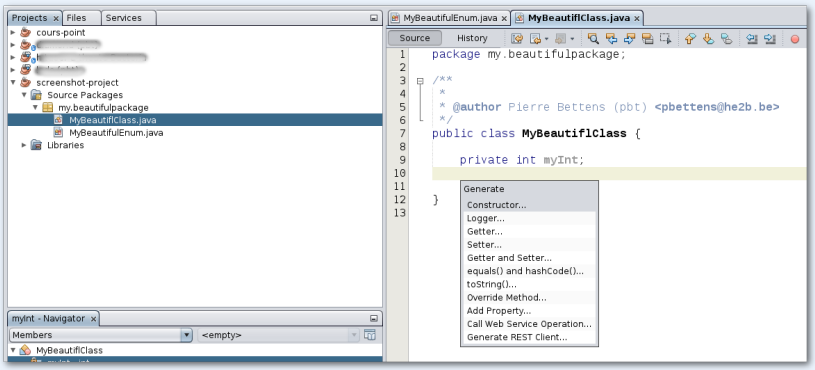
Redéfinition de méthodes

Dans toutes les classes du projet, nous vous laissons juger s'il est utile de redéfinir les méthodes de base comme `toString`, `equals` et `hashCode`.

3. Après tout, la règle stipule bien que : « Durant la partie, les couleurs des trèfles n'ont plus aucune importance. Les couleurs ne servent qu'à faciliter la préparation du jeu. »

Astuce Netbeans

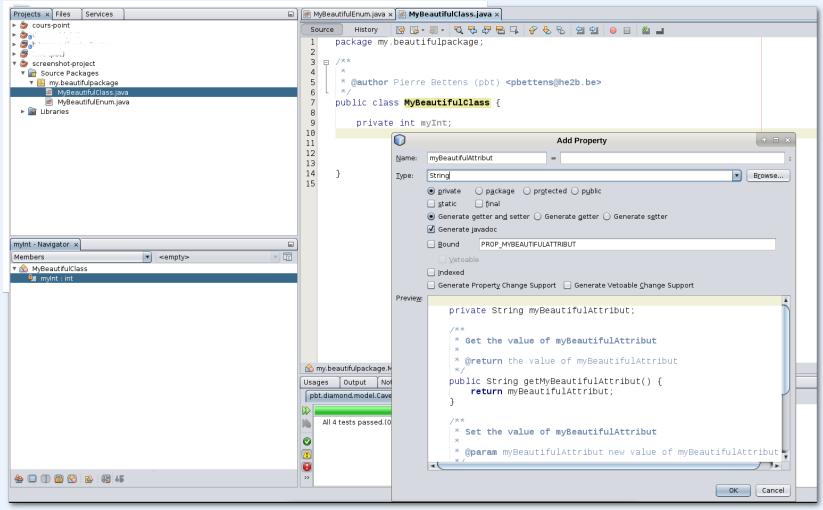
Netbeans a un raccourci incontournable : Alt - Insert . Également accessible par un clic droit — c'est un menu contextuel — dans le *block* de la classe. Ce menu propose l'ajout — en fonction du contexte — d'un constructeur, de *getters* et *setters*...



Pour peu qu'un attribut existe, ce menu contextuel peut ajouter un constructeur en un clic et les *getter* et *setter* en un autre clic.

Imaginons maintenant qu'aucun attribut n'existe et que l'on veuille ajouter un attribut, son getter, son setter. Dès lors que le curseur est dans le *block* de la classe, ce menu contextuel propose *Add property...*.

- Alt - Insert et choisir *Add property...* ;
- dans la fenêtre qui s'ouvre, indiquer
  - le nom de la propriété ;
  - son type ;
  - ce qu'il faut générer (par défaut, Netbeans génèrera ; *getter*, *setter*) ;
  - fixer la visibilité de l'attribut (par défaut, *private*)
- cliquer sur OK et tout est là.



Vérification des paramètes

Dans les classes de bas niveau comme celle-ci, nous ne demandons pas de gérer la validité des paramètres (comme par exemple que la valeur fournie au constructeur est bien entre 1 et 20). Nous ferons toutes les vérifications nécessaire dans la façade `Game` qui sera décrite plus loin.

Astuce Netbeans

Alt - Shift - F permet de mettre correctement le code en forme.

Si des lignes sont sélectionnées, c'est le bloc qui est reformaté sinon, c'est toute la classe. Plus d'excuse d'avoir un code mal indenté.

Lorsque cette classe est terminée, y compris sa javadoc, il reste à faire un *commit* avec un message explicite et un *push* sur *git-esi*. Par exemple :

```
Ajout de la classe Tile (section 3.1)
```

Rappel : Les commits demandés sont **obligatoires** sans quoi nous n'évaluerons pas votre travail.

3.2 Une position

Position
- row : integer - column : integer
+ Position(row : integer, column : integer) + getRow() : integer + getColumn() : integer

Pour indiquer une position sur un plateau de jeu, nous n'allons pas nous contenter de deux valeurs entières représentant la ligne et la colonne ; Nous allons les rassembler pour former une *position*.

État. Une position est caractérisée par une ligne et une colonne.

Comportement. On se contentera d'un constructeur et des accesseurs mais aucun mutateur.

Astuce Netbeans

Netbeans montre une erreur de compilation en soulignant la ligne en rouge et en ajoutant une icône rouge avec un point d'exclamation. En survolant l'icône, une information sur l'erreur apparait.

Netbeans montre un indice (*hint*) en soulignant la ligne en jaune et en ajoutant une icône jaune représentant une ampoule. Il cherche à attirer votre attention sur un code qui, bien que correct, est inutile ou pourrait être écrit différement. En survolant l'icône, une information apparait. Lorsque l'on clique sur cette icône, Netbeans propose une action.

S'il est nécessaire de corriger les erreurs de compilation, il n'est pas toujours judicieux d'accepter la modification proposée par les indices (*hint*). N'acceptez ces modifications que si vous les comprenez !

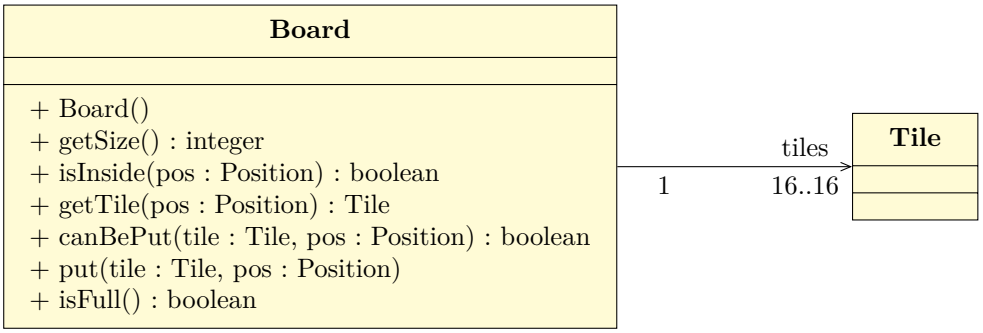
Pour paramétrer le comportement des indices de Netbeans,

- ▷ choisissez le menu *Tools/Options* ;
- ▷ cliquez sur *Editor* et choisissez l'onglet *Hints* ;
- ▷ sélectionnez les cases qui vous conviennent.

Lorsque cette fonctionnalité est terminée — la javadoc est également écrite — faites un *commit* avec un message explicite et un *push*.

3.3 Un plateau

Nous pouvons à présent coder un plateau, l'endroit sur lequel un joueur va placer ses tuiles-trèfles.



État. Nous allons utiliser un tableau `tiles` de 4×4 tuiles. Une case vide (sans tuile) sera indiquée par une valeur `null`.

Comportement.

- ▷ Le constructeur ne reçoit aucun paramètre. Il crée un plateau vide de tuiles.
- ▷ `getSize` retourne le nombre de lignes (et de colonnes, c'est la même chose) du plateau. Normalement c'est 4 mais le fait d'en faire un méthode permettrait de facilement introduire des variantes du jeu avec des plateaux de taille différente.
  Dans tout le reste de votre modèle, on vous demande de faire appel à cette méthode plutôt que d'écrire explicitement 4.
- ▷ `isInside` permet de savoir si une position donnée est bien une position du plateau (ligne et colonne entre 0 et 3, ou, plus exactement, `getSize()-1`).
- ▷ `getTile` donne la tuile à la position donnée (`null` si cette position est vide). On suppose que la position est bien dans le plateau.
- ▷ `canBePut` indique si la tuile donnée peut être placée à la position donnée en respectant les règles. On suppose que la position est bien dans le plateau. Elle peut être vide ou occupée.
- ▷ `put` place la tuile donnée à la position donnée. Aucune vérification n'est faite sur le respect des règles. On suppose que la position est bien dans le plateau. Si une tuile s'y trouvait déjà, elle est remplacée et perdue.
- ▷ `isFull` vérifie si le plateau est complètement rempli de tuiles.

⚠ S'écarter de l'énoncé

L'énoncé est fort dirigiste : nous imposons les classes, attributs et méthodes. Vous **devez vous y tenir**. L'objectif est de vous montrer, au travers d'un cas concret, les bonnes pratiques en ce qui concerne l'architecture d'un code. Si un écart vous parait justifié, vous devez **demander l'accord** de votre professeur de laboratoire.

Se rassurer. Afin de vérifier que votre code n'est pas complètement à coté de la plaque, on vous invite à écrire une méthode principale qui, par exemple :

- ▷ Crée un plateau ;
- ▷ Y place quelques tuiles ;
- ▷ Affiche le résultat (vous n'êtes pas obligé à ce stade de soigner l'affichage).

Décomposer son code

Il vous est interdit de créer d'autres méthodes publiques mais vous pouvez ajouter autant de méthodes privées que vous voulez.
Vous pouvez et vous êtes même encouragé à **décomposer** vos méthodes afin de : faciliter votre écriture, réutiliser du code et le rendre plus lisible.
Une bonne découpe fera partie de nos **critères d'évaluation** de votre projet.

Tester le code. La méthode principale que vous venez d'écrire a de quoi vous rassurer mais elle n'a pas permis de mettre en évidence tous les bugs qu'il pourrait rester dans votre code. Pour ça, il faut des tests unitaires complets.

Vous avez de la chance, nous vous les fournissons cette fois-ci – et uniquement cette fois ci ☺. Examinez-les attentivement pour vous en inspirer pour la suite. Par contre, il vous est interdit de les modifier.

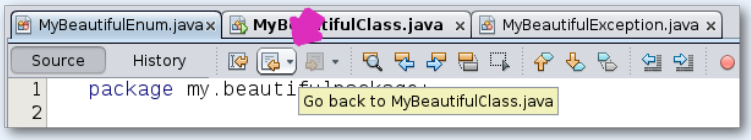
⚠ Aucun test ne se lance

Si le rapport de test vous dit qu'aucun test n'a été exécuté ou mentionne que tous les tests ont raté à cause d'un `NullPointerException` c'est que vous devez mettre à jour votre SUREFIRE – un composant MAVEN utilisé pour les tests.
Pour ce faire, votre `pom.xml` doit contenir :

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>2.22.2</version>
      </plugin>
   </plugins>
</build>
```

Astuce Netbeans

Ctrl - clic sur un nom de méthode a pour effet de se rendre au code de cette méthode ce qui est très pratique. Pour revenir au point précédent cliquer là (voir *screenshot*).

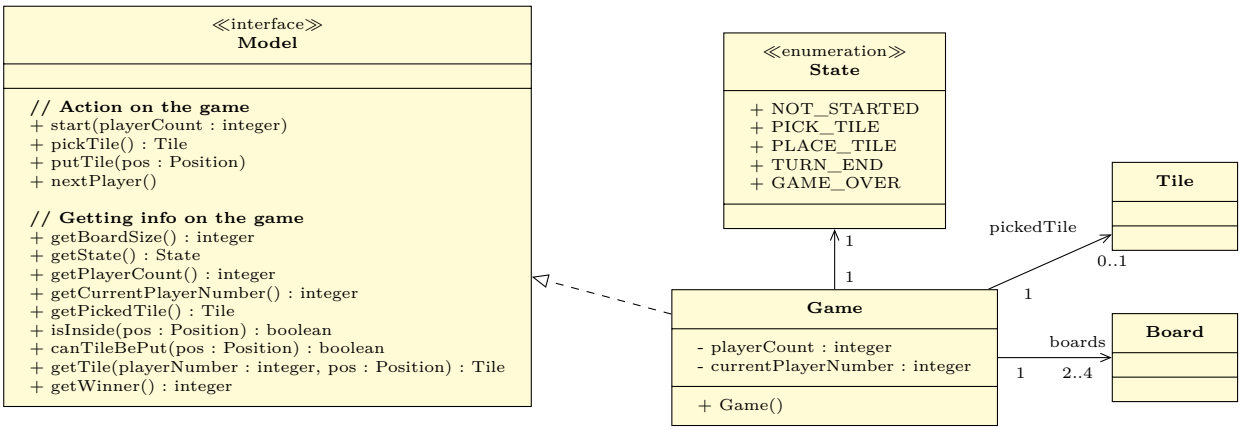


Il arrive, lorsque l'on édite son code, de cliquer sur d'autres onglets pour aller voir d'autres classes et d'autres méthodes. Ctrl - q vous ramène à l'endroit de dernière édition.

Lorsque tous les tests passent et que la classe est complètement documentée faites un *commit* avec un message explicite et un *push*.

4 v0.2 – Terminons le modèle

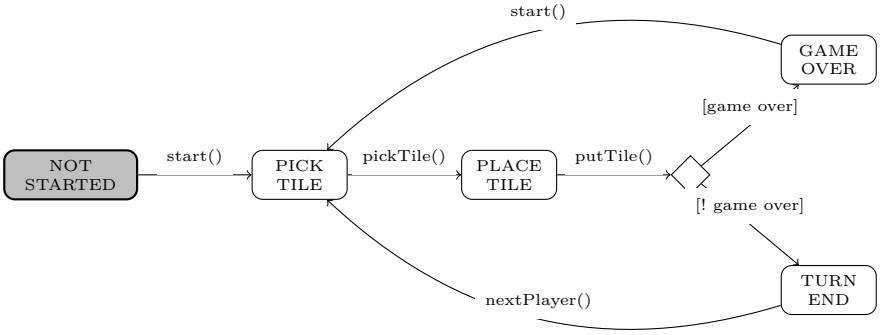
Dans cette partie, nous allons nous occuper de terminer la partie *modèle* de notre application. Essentiellement en introduisant la classe `Game` qui va rassembler les différents éléments du jeu et implémenter les différentes étapes d'une partie.



4.1 L'énumération `State` - l'état du jeu

Les différentes méthodes fournies par le `Model` devront être appelées dans un certain ordre. Ainsi, il serait illogique de tirer une tuile si la partie n'a pas encore démarré ou bien de placer une tuile si on n'en a pas encore tiré une.

Le **diagramme d'état** est un schéma UML standardisé pour décrire par quels états va passer un système et ce qui va déclencher les changements d'états.



Sur ce schéma, on peut lire par exemple que :

- Au début, on est dans l'état `NOT STARTED` et que la seule action possible sur le jeu sera d'appeler la méthode `start()`.
- Lorsque le joueur pose une tuile il y a deux possibilités
  - Si la partie est finie, la seule possibilité sera d'en recommencer une.
  - Si la partie n'est pas finie, on est à la fin du tour du joueur et la seule suite possible est de passer au joueur suivant.

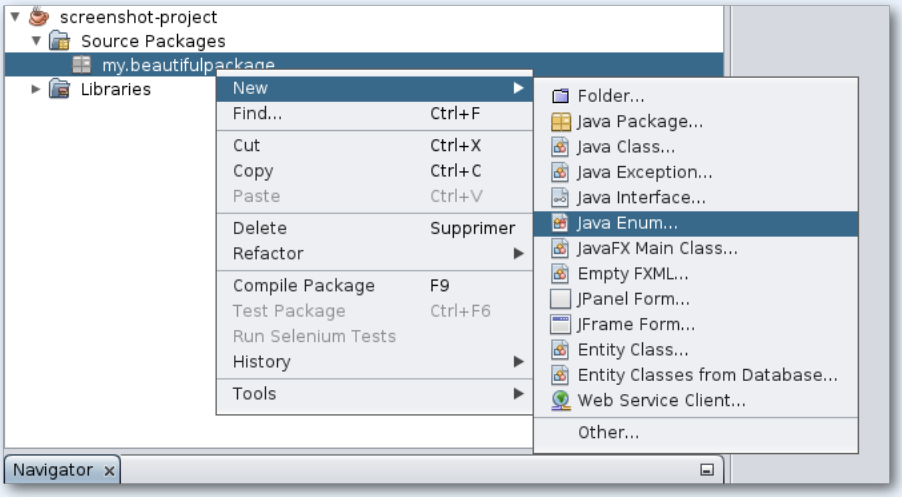
L'énumération `State` a pour but de retenir où on est en dans la partie (dans quel état du diagramme ci-dessus) afin de vérifier que chaque appel est licite à ce stade du jeu.

Il est donc temps maintenant d'écrire cette énumération. Chaque valeur doit être clairement documentée.

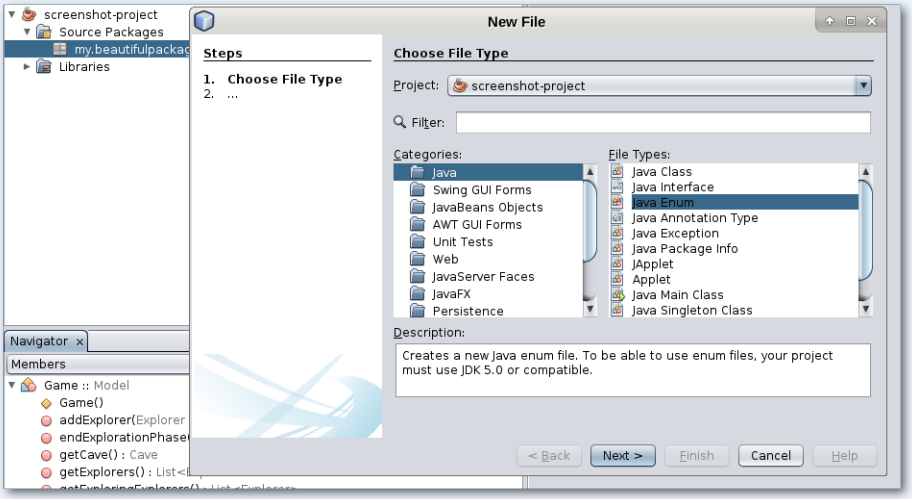
Astuce Netbeans

Pour créer une **énumération** avec Netbeans :

- ▷ clic droit sur le package et glisser la souris sur `new` ;
- ▷ si `Java Enum` apparait, cliquer dessus et se laisser guider ;



- ▷ sinon, cliquer sur `Other`, choisir dans le menu `Java Enum` et se laisser guider.



4.2 L'interface `Model`

La classe `Game` rassemble les éléments nécessaires au jeu et implémente les différentes étapes du jeu. Elle est le point d'accès privilégié pour la vue et le contrôleur.

Parce que c'est une bonne pratique (et aussi pour vous faire pratiquer cette notion ☺) nous introduisons une interface `Model` qui définit les méthodes que doit implémenter la classe `Game`. Certaines ont une action sur le jeu (elles le font avancer) ; d'autres interrogent le jeu sur son état.

Nous vous fournissons cette interface. Voyez la documentation pour comprendre précisément ce que doit faire chaque méthode. La section suivante vous y aidera aussi.

N'oubliez pas le *commit* et le *push*.

4.3 Plions le Game

Vous pouvez maintenant écrire la classe `Game` qui implémente le `Model`.

État.

- ▷ `state` retient à quel stade du jeu on en est.
- ▷ `playerCount` retient le nombre de joueurs pour cette partie (de 2 à 4).
- ▷ `currentPlayerNumber` retient le numéro (de 0 à `playerCount` − 1) du joueur courant.
- ▷ `boards` est un tableau qui contient les plateaux des joueurs. Sa taille est donc de `playerCount`.
- ▷ `pickedTile` est la tuile sélectionnée par le joueur.

Comportement.

- ▷ Le constructeur placera le jeu dans l'état `NOT_STARTED`. Il n'initialise aucun autre attribut ; ce sera le rôle de `start()`.
- ▷ Toutes les méthodes à écrire sont celles de l'interface. Faites bien attention à tester l'état et à la modifier comme indiqué dans la documentation afin de garder une cohérence avec le diagramme d'état introduit au point 4.1.

Héritage de javadoc

Les méthodes de `Game` sont déjà documentées dans l'interface. Si vous n'écrivez pas de javadoc pour une méthode héritée, elle sera reprise aussi de l'interface.

Astuce Netbeans

Si le curseur se trouve sur un nom de classe ou de méthode, Ctrl - Shift - Space affiche sa documentation (en allant la chercher dans un parent si nécessaire). Pratique quand on écrit une méthode dont la javadoc se trouve dans l'interface !

Se rassurer. Comme test rapide vous pouvez écrire une méthode principale qui :

1. crée un jeu à 2 joueurs ;
2. démarre la partie ;
3. choisit la tuile du 1er joueur ;
4. place la tuile à une position fixée ;
5. vérifie que la tuile occupe bien la position fixée ;
6. passe au joueur suivant ;
7. vérifie que le joueur courant est bien le second.

Déclarer une interface, instancier une classe

Les bonnes pratiques indiquent de toujours utiliser l'interface dans la déclaration et de n'utiliser la classe que lors de l'instanciation (le `new`). Par exemple, on écrira : `Model game = new Game();`.

Lorsque tout ceci à l'air de fonctionner, faites un *commit* suivi d'un *push*.

4.4 Tests de Game

Cette classe `Game` que vous venez d'écrire mérite d'être testée en profondeur. Mais il y a un problème : « Comment tester correctement quand une méthode contient un aspect aléatoire ? »

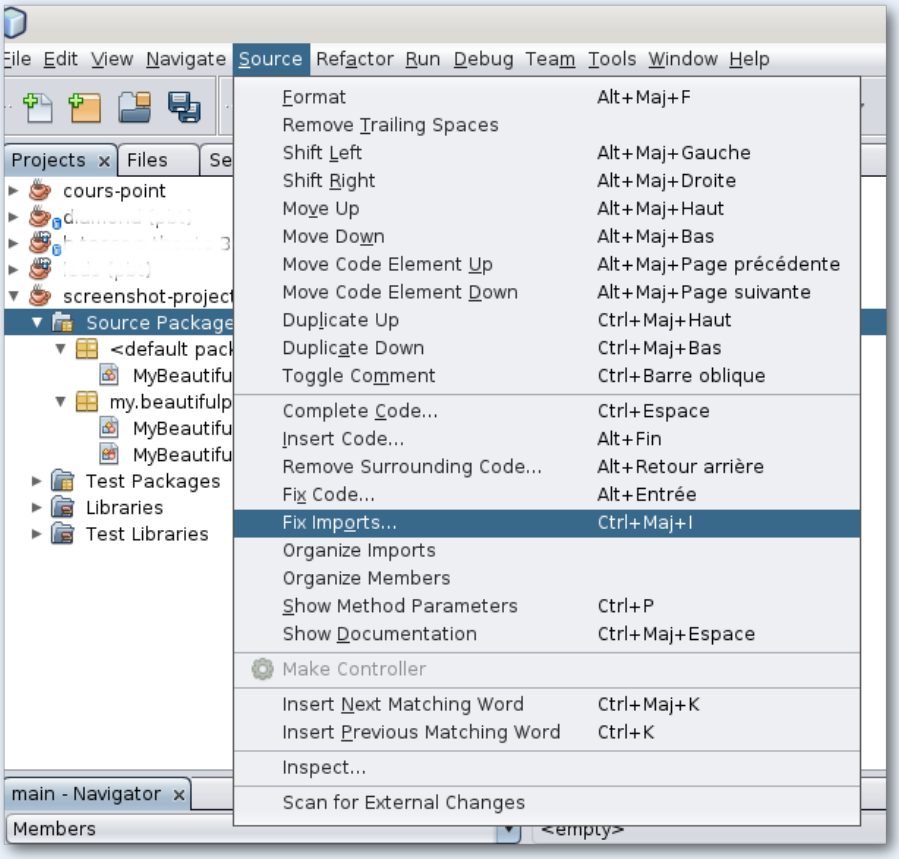
Pour pouvoir mener à bien nos tests, nous nous permettons d'ajouter à la classe `Game` une méthode de **visibilité package** qui va permettre de contrôler quelle tuile est prise.

```
/**
* Pick a tile with the given value. Should be used only for the JUnit tests.
* @return the picked tile.
*/
Tile pickTile(int value) {...}
```

Nous vous demandons d'écrire des tests unitaires complets. Pour vous aider, nous vous fournissons les tests de la méthode `start` (cf. poÉSI). Inspirez-vous en pour tester les autres méthodes.

Astuce Netbeans

Parfois il manque des **import**, parfois il y en a trop parce que l'on utilise une classe à un moment et, plus tard, on décide de ne plus l'utiliser.
Netbeans peut vérifier la liste des *import*, ajouter les manquants et supprimer les inutiles.
Ctrl Shift I ou `Source / Fix import` dans le menu.

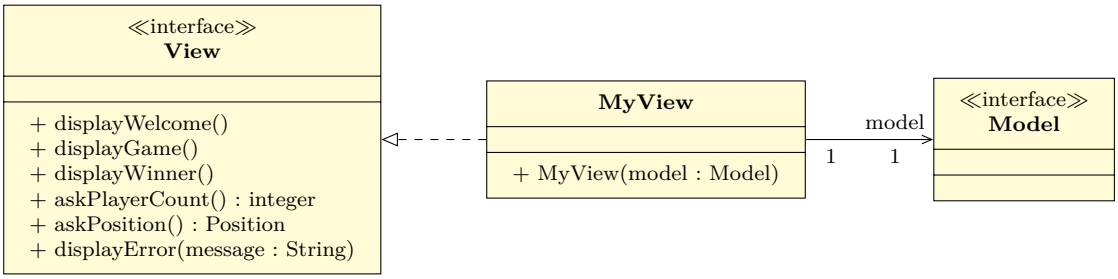


Lorsque tous les tests passent faites un *commit* avec un message explicite et un *push*.

5 v0.3 - Introduisons la vue

Pour le moment vous avez quelques affichages réalisés grâce à des méthodes `toString`. La bonne pratique demande à faire tous les affichages (et les lectures) dans une classe dédiée du package `view`.

Pour continuer dans les bonnes pratiques, on introduit une interface (mais c'est à vous de l'écrire ☺).



Comportement.

- `displayWelcome` s'occupe de l'affichage en début de partie : nom du jeu, auteur, version...
- `displayGame` s'occupe de l'affichage de l'état du jeu : joueur courant et son plateau ainsi que la tuile qu'il doit placer [4].
- `displayWinner` affiche le numéro du joueur gagnant.
- `displayError` affiche le message d'erreur en paramètre.
- `askPlayerCount` demande combien de joueurs participent à la partie (de 2 à 4).
- `askPosition` demande à l'utilisateur d'entrer un numéro de ligne et un numéro de colonne et les retourne sous forme d'une position. Elle s'assure que cette position est bien valide.

Nous vous laissons libres de choisir l'interaction avec l'utilisateur. Par exemple, l'affichage pourrait ressembler à [5] :

```
Joueur 1

    1  2  3  4
 -------------
1|  4  .  .  .
2|  .  . 13  .
3|  .  .  . 20
4|  .  .  .  .
 -------------
Tuile choisie : 10
```

S'adapter aux habitudes de l'utilisateur

Vous avez peut-être remarqué que nous numérotons les colonnes de 1 à 4 ! L'interface doit toujours penser à ce qui est le plus naturel pour l'utilisateur. Elle doit se charger de faire la conversion entre la représentation (où on commence à 0) et l'utilisateur (qui compte à partir de 1).

4. N'oubliez pas que vous ne pouvez pas utiliser explicitement 4 dans votre code mais que vous devez utiliser la méthode `getBoardSize`.
5. Ce n'est vraiment qu'un exemple ; trouvez votre propre style.

Décomposer son code

Nous l'avons déjà dit mais il est parfois bon de se répéter : vous êtes encouragé à **décomposer** vos méthodes. Par exemple, nous verrions d'un mauvais œil que la méthode `displayGame` ne le soit pas ☺.

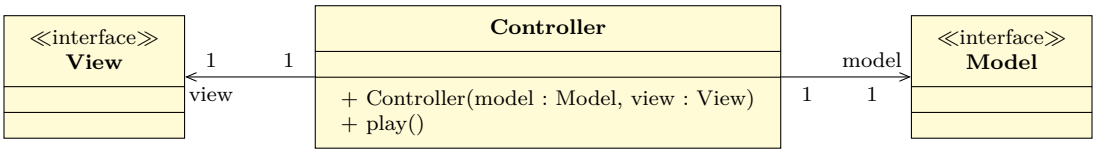
⚠ Tout est dans la vue

Maintenant que vous disposez d'une vue, tout, **absolument tout**, l'affichage doit se faire dans cette vue. Ainsi, il est interdit d'utiliser, même implicitement, les méthodes `toString` du modèle.
Cette bonne pratique permet de facilement changer de vue ; il suffit de remplacer les classes de la vue !

Lorsque la classe est terminée faites un *commit* avec un message explicite et un *push*.

6 v0.4 - Le contrôleur

À ce stade, vous avez une méthode principale qui simule quelques étapes du jeu mais on n'en est pas encore au stade d'un jeu complet. C'est ce que nous allons faire maintenant au travers de la classe `Controller` du package homonyme.



Comportement.

- Le constructeur initialise les 2 attributs.
- La méthode `play` gère un jeu du début à la fin. Elle est pilotée par l'état du jeu. En voici un extrait :

```
public void play() {
    view.displayWelcome();
    while (true) {
        switch (game.getState()) {
            case NOT_STARTED:
                int playerCount = view.askPlayerCount();
                game.start(playerCount);
                break;
            // À vous d'écrire les autres cas.
```

Décomposer, encore

On ne se lasse pas de vous le répéter : décomposer votre code. Le code associé à un cas du *switch* peut être mis directement s'il est court. Par contre, au delà de quelques lignes, vous gagnez à le placer dans une méthode dédiée, appelée dans le *switch*.

La méthode principale de la classe `g12345.luckynumbers.LuckyNumbers`, elle, se réduit maintenant au strict minimum.

```
public static void main(String[] args) {
    Model game = new Game();
    Controller controller = new Controller(game, new MyView(game));
    controller.play();
}
```

Vérifiez que votre code fonctionne, passe les tests, est documenté et propre. Faites un *commit* avec un message explicite suivi d'un *push*.

7 v1.0 - La touche finale

Ceci termine la première itération. Félicitations !

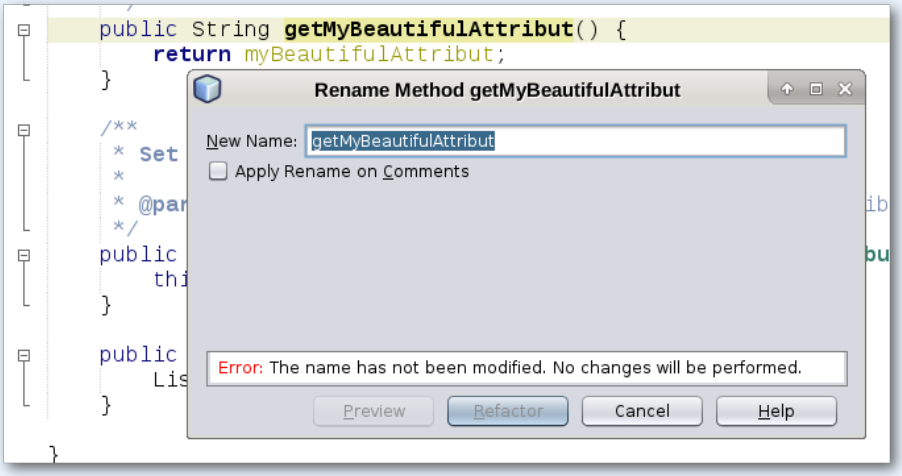
Avant la remise définitive, n'hésitez pas à repasser sur tout le code afin de vérifier qu'il soit correctement documenté et le plus lisible possible. Revérifiez aussi une dernière fois que tous les tests passent toujours !

Astuce Netbeans

Lorsque vous voulez renommer une variable avec Netbeans :

- placer votre curseur sur la variable ;
- entrer Ctrl r ou `Clic droit / Refactor / Rename` et le nom de la variable s'encadre en rouge ;
- renommer et ce renommage prendra effet partout.

Lorsque vous voulez renommer une méthode, c'est le même principe excepté qu'une fenêtre de dialogue s'ouvre.

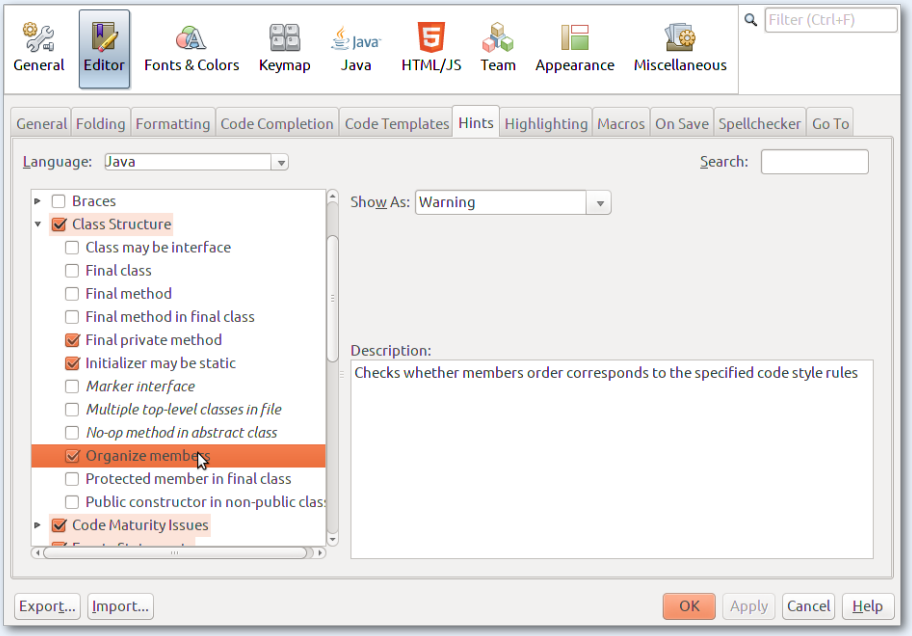


Vous n'avez aucune excuse pour ne pas repasser sur votre code et renommer des variables et/ou des méthodes privées afin d'améliorer sa lisibilité.

Astuce Netbeans

Java recommande un ordre bien précis pour les éléments d'une classe : les attributs puis les constructeurs et, enfin, les méthodes.
L'outil d'insertion de code de Netbeans aboutit régulièrement à un code qui ne respecte pas ces règles. Si on active la bonne option, Netbeans peut vous indiquer si l'ordre n'est pas respecté et réordonner les éléments pour vous.
Pour cela,

- ▷ choisissez le menu *Tools/Options* ;
- ▷ cliquez sur *Editor* et choisissez l'onglet *Hints* ;
- ▷ dans la section *Class Structure*, cochez l'option *Organize Members*.



Une fois cette option cochée, Netbeans affichera une petite ampoule devant le premier élément mal placé et vous proposera de réorganiser toute la classe.

Lorsque vous êtes satisfait de votre code, faites un dernier *commit* avec un message explicite suivi d'un *push*.
Vous pouvez *taguer* [a] votre *commit* **iteration1**.

a. https://git-scm.com/book/en/v2/Git-Basics-Tagging *consulté le 10 mars 2020*

8 Annexe - Diagramme de classe complet