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

2019 – 2020
DEV2
pbt

DEV2 – Laboratoire Java

Projet - Partie 1

Humbug

Table des matières

Version du document
10 mars 2020

Présentation du projet

Nous vous proposons de développer une version simplifiée et console du jeu **Humbug** disponible via le site des développeurs [1].

Le projet se compose de 3 phases :

1. l'**itération 1** qui est une première version du jeu avec des fonctionnalités limitées et pour laquelle nous vous guidons ;
2. l'**itération 2** ajoute des fonctionnalités à l'application et nous vous laissons plus libres dans les choix d'implémentation ;
3. la **défense** du projet sans laquelle le projet n'est pas évalué.

1 Modalités pratiques et méthodologie

Inscription

Avant toute chose, il est nécessaire de vous inscrire auprès de votre professeur de laboratoire qui va vous créer un projet sur le serveur git [2] de l'école. C'est avec **ce** dépôt que vous travaillerez et remettrez votre travail à l'issue des différentes échéances.

Échéances et pondération

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

	Don de l'énoncé	Remise	Pondération
Itération 1	semaine du 16 mars	semaine du 23 mars	8/20
Itération 2	semaine du 6 avril	semaine du 4 mai	12/20
Défense		semaine du 11 mai	coefficient

Défense

La défense du projet consiste en une demande de modifications du projet par l'ajout ou la modification de fonctionnalités, la défense orale et la démonstration que le code est fonctionnel. Nous supposons que le travail se fait en utilisant NetBeans.

La défense est obligatoire.

Méthodologie

Dans ce projet, nous mettrons en œuvre différentes techniques de développement : l'approche itérative [3], le *refactoring* (réusinage en français) de code [4] et le développement pilotés par les test (*tests driven*) [5].

Chaque personne développant son projet sera donc invitée à :

- ▷ écrire des tests JUnit ;
- ▷ *refactoriser* son code ;
- ▷ sauvegarder et identifier les différentes versions de son code dans son gestionnaire de version *aka* Git ;
- ▷ travailler sous NetBeans (en échange, nous parsèmerons cet énoncé d'*astuces*)

1. https://www.dunderbit.com/ *consulté le 30 janvier 2020*
2. https://git.esi-bru.be *consulté le 30 janvier 2020*
3. https://fr.wikipedia.org/wiki/Cycle_de_développement_(logiciel) *consulté le 30 janvier 2020*
4. https://fr.wikipedia.org/wiki/Réusinage_de_code *consulté le 30 janvier 2020*
5. https://fr.wikipedia.org/wiki/Test_driven_development *consulté le 30 janvier 2020*

Gestionnaire de version Git

Utilisation de git tout au long du développement du projet :

- ▷ vérifier l'accès au dépôt *git-esi*[a] créé par l'enseignant ou l'enseignante ;
- ▷ faire des *commits* réguliers et au minimum après chaque fonctionnalité ;
- ▷ maintenir la synchronisation avec le serveur en faisant des *push* et des *pull* régulièrement.

a. Dans la suite *git-esi* désigne le serveur gitlab de l'école à l'adresse `https://git.esi-bru.be`

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.

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 [6]).

Ce paradigme regroupe les fonctions nécessaires en trois catégories :

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

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.humbug.model` [7].

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.humbug.view.text`.

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

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

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

Itération 1

3 Le modèle *v0.1*

Pour commencer, décrivons les données que nous allons utiliser il y aura bien sûr un plateau de jeu (*board*) constitué de petites cases (*square*). Nous distinguerons deux types de cases (*square type*) selon que ce soit de l'herbe ou une case d'arrivée d'un animal.

Sur ce plateau de jeu se trouvent des animaux (*animal*) qui s'y déplacent.

Commençons par ça.

3.1 Énumération `SquareType`

Écrivons une énumération, avec deux valeurs :

- `GRASS` pour les cases contenant de l'**herbe** ;
- `STAR` pour les cases d'arrivée des animaux. Elles sont représentées par une **étoile**.

Pour cette première classe, voici le code :

```
package pbt.humbug.model;

/**
 * SquareType represents the type of a square on the board.
 * Square are grass or star when there represent arrival.
 *
 * @author Pierre Bettens (pbt) <pbettens@he2b.be>
 */
public enum SquareType {
   GRASS, STAR;
}
```

Quelques remarques sur ces lignes de code :

- le code est commenté ;
- le nom de la classe est en *CamelCase* ;
- les constantes sont en majuscules ;
- les valeurs sont séparées par une virgule. La virgule est suivie d'une espace.

> Cette fonctionnalité étant écrite et sa javadoc également, il reste à faire un *commit* (avec un message explicite) et un *push* sur *git-esi*.

3.2 Classe `Square`

Une case est une des cases du plateau de jeu. Une case n'a qu'un seul attribut qui est le type de case qu'elle représente.

Cette classe possède :

- un attribut de type `SquareType` ;
- un constructeur à un paramètre de type `SquareType` représentant le type de la case ;
- un *getter* pour son attribut.

Pour cette deuxième classe, voici le code. Promis, après, c'est vous qui codez.

```java
package pbt.humbug.model;

/**
 * Square on the board. A square has a type grass or star and it's all.
 * A square doesn't know where it is on the board.
 *
 * @author Pierre Bettens (pbt) <pbettens@he2b.be>
 */
public class Square {
   private SquareType type;

   /**
    * Constructor of Square on board.
    * @param type Square is grass or star
    */
   public Square(SquareType type) {
      this.type = type;
   }

   /**
    * Simple getter of type.
    * @return type of Square
    */
   public SquareType getType() {
      return type;
   }

}
```

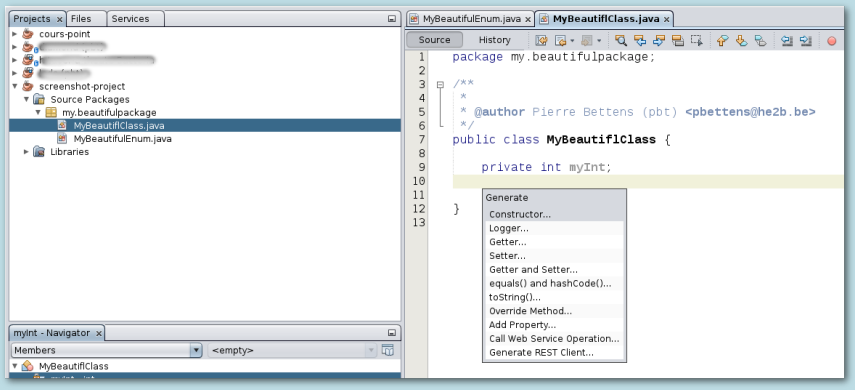
Quelques remarques sur ces lignes de code :

- ▷ le code est commenté ;
- ▷ le nom de la classe est en *CamelCase* ;
- ▷ le nom de l'attribut est représentatif de ce qu'il représente ;
- ▷ il y a un passage de ligne entre chaque méthode (pas deux).

Faites un *commit* avec un message explicite suivi d'un *push*.

Astuce Netbeans

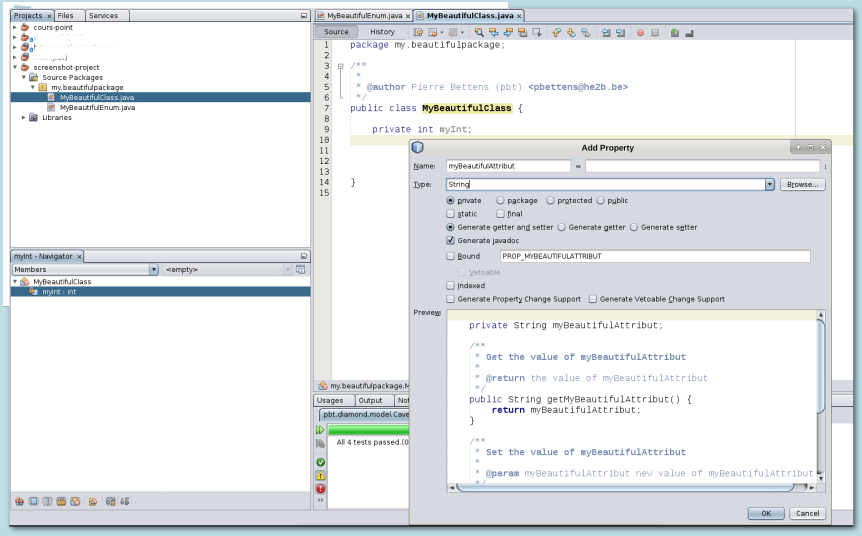
Netbeans a un raccourci incontournable ; [**Alt**][**Insert**]. Également accessible par un clic droit — c'est un menu contextuel — dans le *block* de la classe. Ce menu propose l'ajout — en fonction du contexte — d'un constructeur, de *getters* et *setters*...



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. Ce menu contextuel propose *Add property...*. Dès lors que le curseur est dans le *block* de la classe :

- [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.



3.3 Classe `Position`

Pour se repérer sur le 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*.

La classe `Position` possède :

- ▷ un attribut privé `row` (*ligne*) de type `int` ;
- ▷ un attribut privé `column` (*colonne*) de type `int` ;
- ▷ un constructeur à deux paramètres ; une ligne et une colonne ;
- ▷ un accesseur (*getter*) pour chaque attribut ;
- ▷ la possibilité de tester l'égalité de deux positions. Il faut pour ce faire réécrire les méthodes `equals` et `hashCode`.

Les deux attributs seront `final` car une position ne change pas. Une position est immuable (*immutable*). Pour rendre un objet immuable, il suffit que ses attributs soient finaux (`final`), privés et que la classe n'expose pas de mutateur (*setter*).

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

Remarque : `toString`

Nous ne préciserons jamais s'il est nécessaire ou non de réécrire la méthode `toString`. Nous vous laissons choisir.

Astuce Netbeans

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.

3.4 L'énumération `Direction`

Pour indiquer dans quelle direction les animaux vont se déplacer, nous utiliserons les quatre points cardinaux : le nord (`north`), le sud (`south`), l'est (`east`) et l'ouest (`west`).

Pour avoir la position au sud d'une position quelconque, il faut ajouter 1 à la valeur de la ligne de la position. Par exemple la position au sud de la position (1,2), est la position (2,2). Nous allons mémoriser dans l'énumération que pour aller au sud, il faut ajouter 1 à la valeur de la ligne. Le même raisonnement se fait pour les trois autres directions.

Pour mémoriser ces informations, nous allons ajouter deux attributs à cette classe : l'un représentant le *delta* pour les lignes et l'autre pour les colonnes. Ce qui donne ces valeurs :

Direction	delta ligne	delta colonne
NORTH	-1	0
SOUTH	1	0
EAST	0	1
WEST	0	-1

Cette classe possède donc :

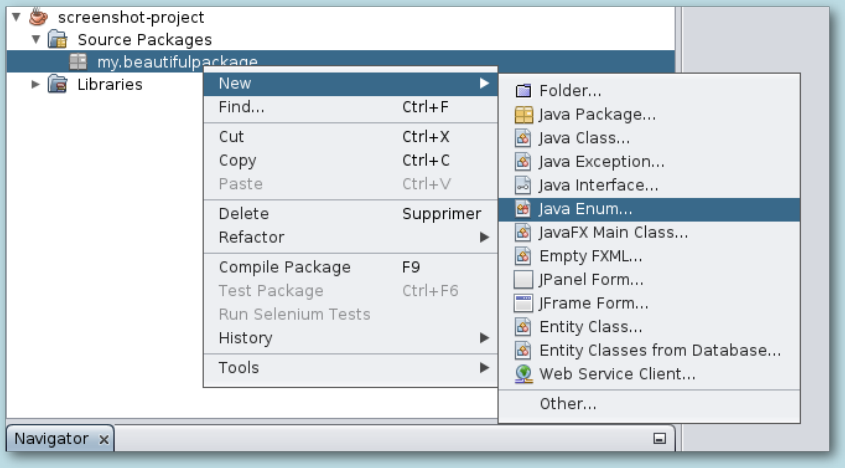
- ▷ les quatre valeurs mentionnées ;
- ▷ deux attributs privés `deltaRow` et `deltaColumn` de type `int` ;
- ▷ deux accesseurs pour ces attributs ;

Lorsque cette fonctionnalité est terminée et la javadoc écrite, faites un *commit* avec un message explicite suivi d'un *push*.

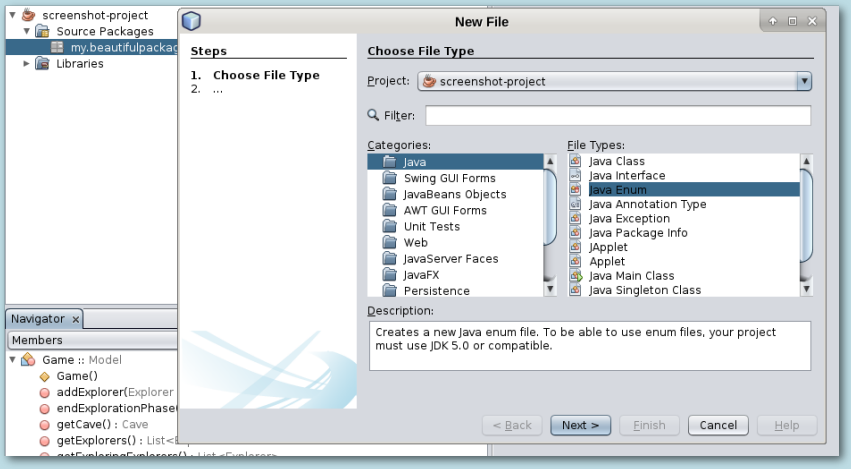
Astuce Netbeans

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* » et choisir dans le menu « Java Enum » et se laisser guider.



3.5 Les tests et la classe `Position`

Il est temps d'intégrer des tests à notre application. Les premiers tests ont été écrits pour vous et sont disponibles avec cet énoncé.

Nous n'avons pas écrit de tests pour les classes `SquareType`, `Square` et `Direction`. Ce n'est pas nécessaire. Intégrez les tests de la classe `Position` à votre projet. Il faudra adapter le nom du package. Les tests échouent car il manque une méthode.

Ajoutez la méthode *next* qui retourne la position juste à côté dans la direction passée en paramètre dont la signature est la suivante :

```
public Position next(Direction)
```

Exemple La position à l'est de la position (0,0) est la position (0,1). Nous pourrions écrire :

```
Position position = new Position(0,0);
System.out.println(position.next(Direction.EAST)); // (0,1)
```

Vérifiez que les tests réussissent et que la javadoc est écrite. Ensuite, faites un `commit` avec un message explicite et un `push`.

Les tests

Dans la suite, lorsque vous écrivez une classe, vérifiez si des tests sont fournis. Si c'est le cas, intégrez-les à votre projet sinon, demandez-vous s'il faut en écrire.

3.6 Classe `Board`

Le plateau de jeu (*board*) est constitué de cases (*squares*) lorsqu'il y en a une et de la valeur `null` s'il n'y en a pas. Le plateau de jeu n'a pas connaissance des différents animaux qui se baladent dessus [8].

Il est représenté par un **tableau à deux dimensions**.

Cette classe possède :

- un attribut privé `squares` de type `Square[][]` ;
- un constructeur à un paramètre de type `Square[][]`. Ce constructeur aura une visibilité *package*, il sera utilisé pour les tests et... par la méthode suivante ;
- une méthode statique retournant un `Board` dont la signature est la suivante :

  ```
  public static Board getInitialBoard()
  ```

  Cette méthode retourne le board du premier niveau de jeu présenté à la figure 1 page suivante.
- une méthode retournant un booléen précisant si une position donnée est sur le plateau de jeu. C'est-à-dire si la position fait référence à une case « herbe » ou une case « star ». La signature de cette méthode est la suivante :

  ```
  public boolean isInside(Position)
  ```

8. C'est un choix d'implémentation qui pourrait être discuté et qui devra être respecté. Il aura comme conséquence que certaines méthodes seront plus compliquées à écrire... et d'autres plus simples.

Cette méthode lance une exception de type `IllegalArgumentException` si le paramètre est `null`.

- une méthode retournant le type d'une case dont la position est passée en paramètre dont la signature est la suivante :

```
public SquareType getSquareType(Position)
```

Cette méthode lance une exception de type `IllegalArgumentException` si la position n'est pas sur le plateau de jeu.

- deux méthodes donnant le nombre de lignes et le nombre de colonnes du plateau de jeu dont les signatures sont les suivantes :

```
public int getNbRow()
public int getNbColumn()
```

Pour le premier niveau, le plateau est comme ci-contre ; il est composé de 3 lignes et de 3 colonnes. En première ligne, première colonne, une case « herbe », en première ligne, deuxième colonne, une case « herbe », en première ligne toujours et en troisième colonne, pas de case, etc.

Ce plateau (*board*) sera représenté par :

GRASS	GRASS	`null`
`null`	GRASS	GRASS
`null`	`null`	STAR

Vérifiez que votre classe passe les tests fournis.

FIGURE 1 – Premier niveau du jeu

Cette fonctionnalité est terminée, faites un *commit* avec un message explicite, suivi d'un *push*.

Astuce Netbeans

Netbeans permet d'écrire rapidement

```
MyObject mo = new MyObject();
```

Il suffit d'utiliser le raccourci **newo**. Taper *newo* suivi de [Tab] et se laisser guider c'est-à-dire entrer le nom de l'objet dans le rectangle rouge et, ensuite, entrer la touche [Enter].

Il existe beaucoup d'autres raccourcis que nous vous invitons à découvrir.

4 Diagramme de classes

Voici le diagramme de classes technique (voir figure 4) pour le modèle dans sa version *0.1*.

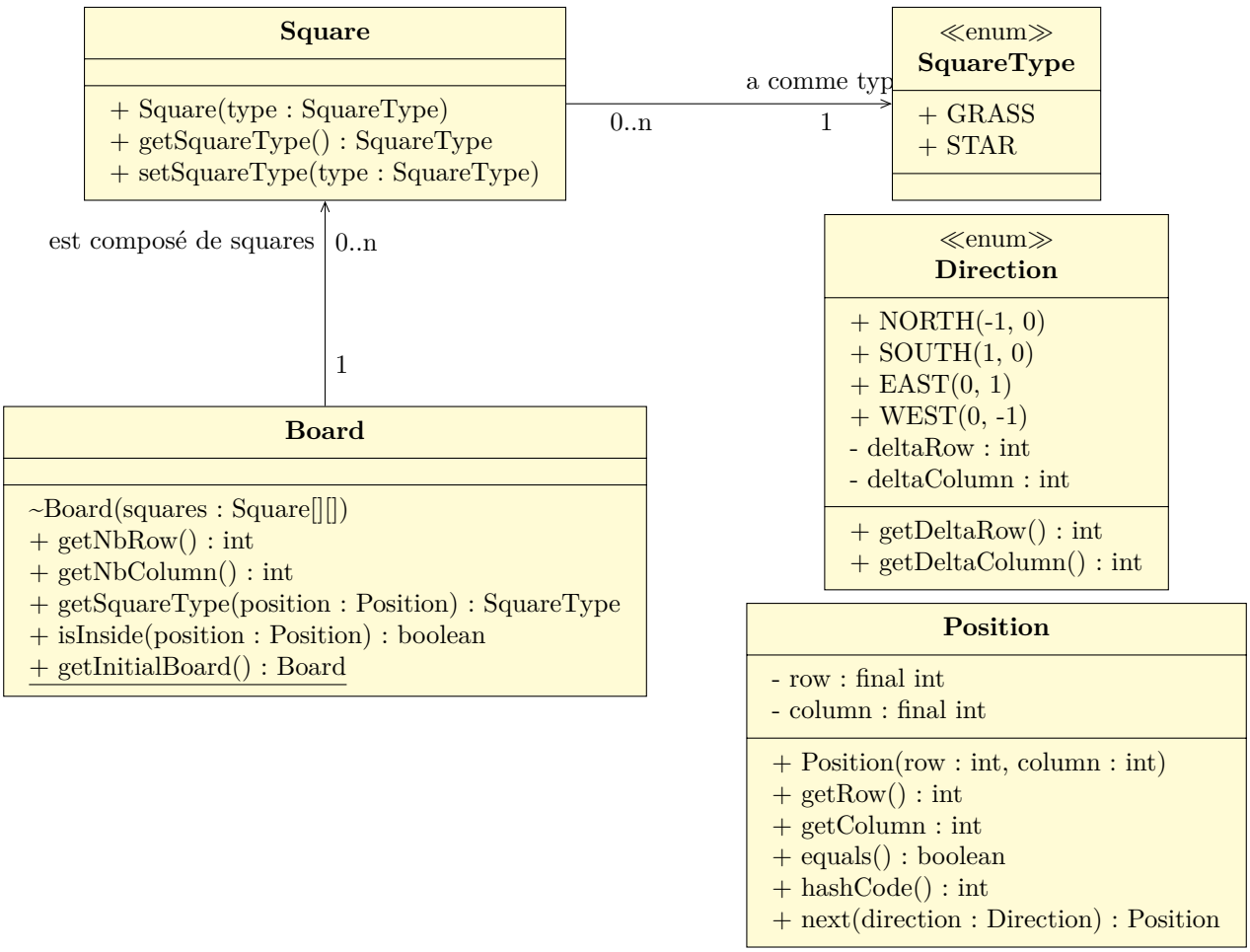


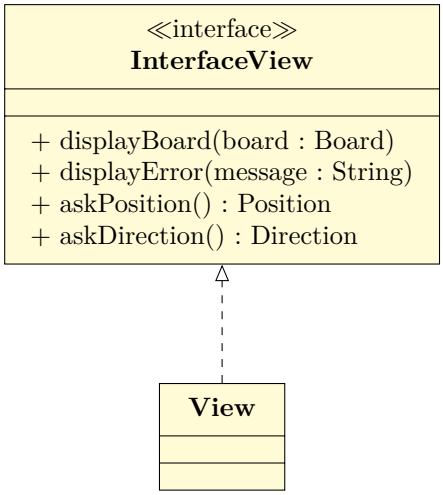
FIGURE 2 – Diagramme de classes technique `v0.1`

5 La vue

Consacrons un peu de temps à l'affichage du tableau de jeu et préparons les saisies de l'utilisateur ou de l'utilisatrice.

5.1 Classe `View`

Dans le package destiné à la vue — *aka* `g12345.humbug.view.text` — ajoutez une classe `View`. Cette classe possède :



- une méthode affichant le plateau de jeu dont la signature est la suivante :

```
public void displayBoard(Board board)
```

  Pour cette méthode nous vous conseillons de créer un tableau de `String` contenant les informations à afficher et d'afficher le tableau en fin de méthode plutôt que d'afficher les lignes au fur et à mesure.
  En effet, un peu plus tard, il faudra afficher également les animaux et ce sera plus simple si vous suivez ce conseil (voir la section 6.1 page 15).

- une méthode affichant un message d'erreur dont la signature est la suivante :

```
public void displayError(String message)
```

- une méthode demandant à l'utilisateur ou à l'utilisatrice d'encoder une position dont la signature est la suivante :

```
public Position askPosition()
```

- une méthode demandant à l'utilisateur ou à l'utilisatrice d'encoder une direction dont la signature est la suivante :

```
public Direction askDirection()
```

Les deux dernières méthodes seront robustes et donc résistantes aux mauvaises entrées de l'utilisateur ou de l'utilisatrice. La méthode `askPosition` ne vérifie bien sûr pas que la position est sur le plateau de jeu.

Pour voir que votre code est fonctionnel, vous pouvez écrire une méthode `main` dans votre classe qui crée un `Board` et l'affiche. Vous pouvez également tester vos saisies.

> Lorsque la documentation est écrite et que votre classe fonctionne, faites un *commit* avec un message explicite, suivi d'un *push*.

5.2 Organisation MVC des classes, première partie

Nous avons décrit dans la section 2 page 3 ce qu'était une architecture MVC.

Pour l'instant nos classes métier se trouvent dans le package `g12345.humbug.model` et c'est bien. La vue se trouve dans le package idoine et c'est bien aussi.

Pour que ce soit un peu plus propre, nous allons :

- ajouter une interface `InterfaceView` dans le package *view.text* qui précise quelles méthodes doivent se trouver dans la vue. Cette classe possède les signatures des méthodes suivantes :
  - `void displayBoard(Board)`
  - `Position askPosition`
  - `Direction askDirection`
  - `void displayError(String)`
- précisez que la classe `View` implémente l'interface `InterfaceView`

Une fois fait, faites un *commit* avec une description explicite suivi d'un *push*.

Astuce Netbeans

[Ctrl][Space] permet l'autocomplétion.

Cette autocomplétion est assez intelligente et dépend du contexte.

1. `int i = [Ctrl][Space]`
2. `List<Integer> is = new Ar[Ctrl][Space]`
3. `MyObject m[Ctrl][Space]`
4. `public class MyBeautifulClass[Ctrl][Space]`
5. `List<Integer> is = new ArrayList<>[Ctrl][Space]`

Les exemples précédents peuvent se compléter comme :

1. méthodes statiques et attributs disponibles. Par exemple, `int i = Integer.SIZE;`
2. constructeur. Par exemple, `List<Integer> is = new ArrayList<>();`
3. suggestion d'un identifieur. Par exemple, `MyObject mo;`
4. *keywords*. Dans ce cas `extends` ou `implements`. Par exemple, `public class MyBeautifulClass extends;`
5. suggestion de paramètres et réaffichage des paramètres attendus. Par exemple, `List<Integer> is = new ArrayList<>(otherList).`

6 Le modèle *v0.2*

Le moment est venu d'introduire les animaux dans notre projet. Nous vous conseillons de relire vos notes au sujet de l'*héritage* avant de continuer.

Voici les ajouts au diagramme technique de classes. Voir figure 6 page suivante.

6.1 Classe abstraite `Animal` et sous-classes `Snail` et `Spider`

Dans le jeu, des animaux se baladent sur le plateau de jeu, le but étant de les faire se déplacer jusqu'aux cases « étoiles ». Un animal **sait où il se trouve sur le plateau**

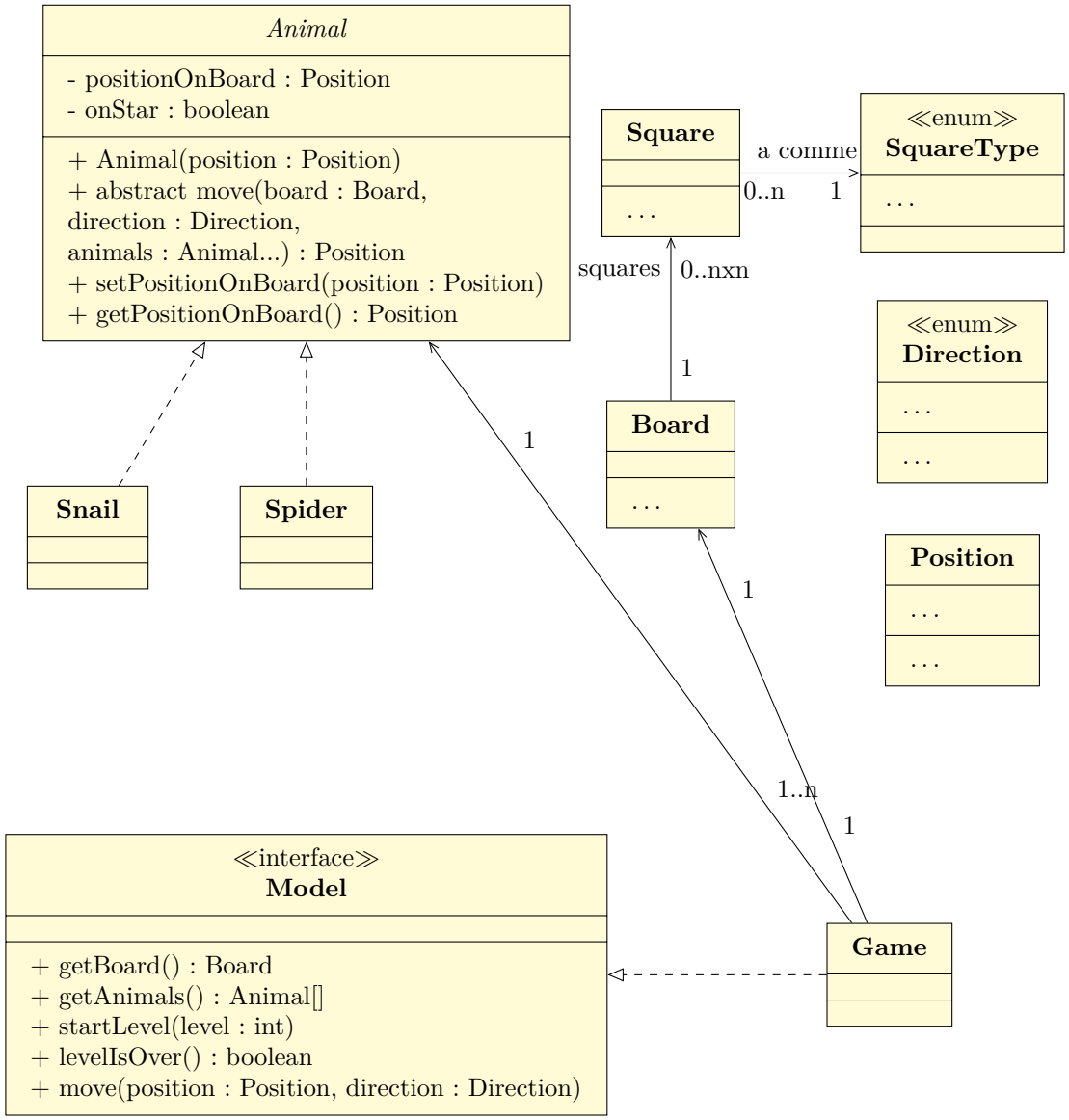


FIGURE 3 – Diagramme de classes technique `v0.2`

de jeu c'est le choix d'implémentation que nous avons fait comme nous l'avons dit à la section 3.6 page 9. Il ne sait par contre pas s'il est sur une case étoile. C'est le but de l'attribut `onStar`. Même s'ils savent où ils se trouvent, ils ne sont pas très prudents et tombent si on les dirige mal.

La classe `Animal` possède :

- ▷ un attribut privé `positionOnBoard` de type `Position` qui représente la position de l'animal sur le plateau de jeu et son accesseur ;
- ▷ un attribut privé `onStar` de type `boolean` qui précise si l'animal est sur une case « étoile » ou pas ;
- ▷ un constructeur avec un paramètre, la position. L'attribut `onStar` est initialisé à `false` ;
- ▷ les accesseurs et mutateurs ;

Les animaux ont la possibilité de se déplacer sur le plateau de jeu. Chaque animal a sa manière de se déplacer ; certains volent ou sautent tandis que d'autres rampent... Nous

n'allons pas écrire une méthode gérant le déplacement de tous les animaux dans cette classe. Nous allons déléguer cette tâche aux classes enfants.

Ajoutez une méthode **abstraite** à la classe `Animal` dont la signature est la suivante :

```
public abstract Position move(Board board,
    Direction direction, Animal... animals)
```

Cette méthode déplace effectivement l'animal — c'est-à-dire qu'elle change la position de l'animal — et retourne la nouvelle position. Si le déplacement ne peut pas se faire — par exemple parce qu'un autre animal bloque le passage —, la méthode retourne la position initiale de l'animal. Si l'animal **tombe**, sa position est placée à `null` et la méthode retourne également `null`.

Lors des déplacements, il faut tenir compte du fait qu'un animal sur une case étoile (*star*) est arrivé. Il disparait et ne bloque plus personne. Nous le laisserons dans la liste des animaux et son attribut `onStar` signale qu'il est arrivé [9]. La case du jeu correspondant à cette position devra devenir herbe (*grass*).

Pour arriver, l'animal doit s'arrêter exactement sur une case étoile. Il ne peut pas se contenter de *passer dessus*. Par exemple si une araignée travers le plateau de jeu pour s'arrêter à un obstacle en passant sur une case étoile, elle n'est pas arrivée.

Rappels

- ▷ Écrire une méthode abstraite dans une classe impose que la classe, elle aussi, soit abstraite et que toutes les classes enfants implémentent la méthode [10].
- ▷ La notation `...` est utilisée pour les **varargs**, nombre variable de paramètres. Il est peut-être aussi utile de relire la théorie à ce sujet.

Écrivez deux classes qui héritent de la classe `Animal` [11].

1. La classe `Snail` représente un **escargot**. L'escargot se déplace d'une case dans la direction indiquée à la condition que la case ne soit pas occupée par un autre animal.
2. La classe `Spider` représente une **araignée**. L'araignée se déplace dans la direction indiquée tant qu'elle ne rencontre pas d'obstacle.

> Des tests existent pour ces deux classes. Vérifiez bien que vos classes passent les tests avant de faire votre *push*.

Maintenant que les animaux sont rentrés dans le jeu, ils doivent apparaitre sur le plateau de jeu. Mettez à jour la méthode `View.displayBoard()` pour que les animaux soient représentés. Cette méthode prendra maintenant deux paramètres et sa signature est la suivante :

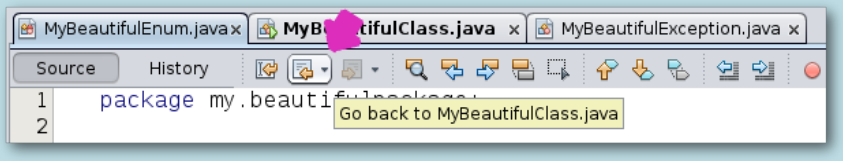
```
public void displayBoard(Board board, Animal... animals)
```

9. Vous pouvez, si vous le préférez, faire le choix de supprimer les animaux arrivés du tableau d'animaux. C'est un autre choix d'implémentation qui ne va pas à l'encontre de l'analyse. Vous devez simplement rester cohérent avec vos choix tout au long de l'implémentation de votre projet
10. Ou la déclarent abstraite à leur tour. Etc.
11. Dans la première itération nous n'aurons besoin que de l'escargot et pas de l'araignée mais nous prenons un peu d'avance ;-)

Astuce Netbeans

[Ctrl]<clic> sur un nom de méthode a pour effet de se rendre au code de cette méthode. Très pratique.

Pour revenir au point précédent cliquer là (voir *screenshot*).



Chez moi, l'affichage (pour le niveau 1) a cette allure — j'utilise la classe `TerminalColor` disponible sur git-esi [12]pour l'affichage en couleur — mais vous êtes libre de faire comme bon vous semble.

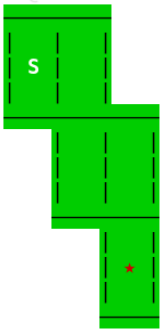


FIGURE 4 – Exemple d'un affichage du *board*

Lorsque vos classes sont documentées et passent les tests, faites un *commit* avec un message explicite suivi d'un *push*.

6.2 La classe `Game`

La classe `Game` rassemble les éléments nécessaires au jeu pour présenter une façade [13] à la vue. La vue interagit uniquement avec cette classe pour l'accès au modèle. L'interface `Model` définit les méthodes que doit implémenter la classe `Game`.

```
public interface Model {
    Board getBoard();
    Animal[] getAnimals();
    void startLevel(int level);
    boolean levelIsOver();
    void move(Position position, Direction direction);
}
```

Cette classe possède donc :

- ▷ un attribut `board` représentant le plateau de jeu et de type `Board` ;
- ▷ un attribut `animals` représentant les animaux présents sur le plateau et de type `Animal[]` ;
- ▷ des accesseurs pour ses attributs ;
- ▷ une méthode *startLevel* comme annoncée qui initialise le plateau de jeu et les animaux pour ce premier niveau (*level 1*) ;

12. https://git.esi-bru.be/pbt/terminalcolor *consulté le 31 janvier 2020*
13. https://fr.wikipedia.org/wiki/Façade_(patron_de_conception) *consulté le 31 janvier 2020*

- ▷ une méthode *levelIsOver* comme annoncée qui précise si le niveau est terminé. Un niveau est terminé si tous les animaux sont sur une case « étoile » ;
- ▷ une méthode *move* comme annoncée qui effectue le déplacement s'il est permis.

Comme les attributs de la classe ne sont pas créés par le constructeur mais par la méthode `startLevel`, il serait bon de vérifier dans les méthodes `move` et `levelIsOver` que ces attributs ne sont pas nuls avant toute chose. S'ils l'étaient, nous lancerions une exception. Vous n'êtes pas obligés de le faire dans cette première itération. Nous reviendrons en détail sur ce point dans l'itération 2.

6.3 Organisation MVC des classes, deuxième partie : le contrôleur

Le projet contient, une vue (texte) et un modèle. Pour implémenter MVC, il manque un contrôleur. Le contrôleur est responsable de la dynamique du jeu et de la mise à jour de la vue au fur et à mesure de l'avancement.

La classe `Controller` se trouve dans le *package* `g12345.humbug.controller` et elle possède :

- ▷ un attribut `game` de type `Model` qui représente le modèle ;
- ▷ un attribut `view` de type `InterfaceView` qui représente la vue ;
- ▷ un constructeur à deux paramètres ; la vue et le modèle ;
- ▷ une méthode `startGame` qui se charge de jouer. Jouer consiste en :
  - ▷ démarrer le modèle ;
  - ▷ tant que le niveau n'est pas fini et qu'aucun animal n'est tombé ;
    - ▷ afficher le plateau ;
    - ▷ demander une position et une direction ;
    - ▷ tenter le déplacement

Remarque Puisque l'on **tente** le déplacement, c'est que celui-ci peut échouer. C'est le bon endroit — et ce sera le seul dans cette première itération — pour utiliser l'instruction `try-catch`. Le bloc *try* fera l'appel au `move` tandis que le bloc `catch` verra apparaitre la méthode `displayError`.

Il reste à écrire une classe `Main` dans le package `g12345.humbug` qui crée le contrôleur et lance la partie.

```
public static void main(String[] args) {
    Controller controller = new Controller(new Game(), new View());
    controller.startGame();
}
```

Ceci termine la première itération.

Vérifiez que votre code fonctionne, passe les tests, est documenté et propre. Faites un *commit* avec un message explicite suivi d'un *push*.
Vous pouvez *taguer* [a] votre *commit* **v1.0**, c'est ce *commit* là qui sera lu par la personne qui évalue votre projet.

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

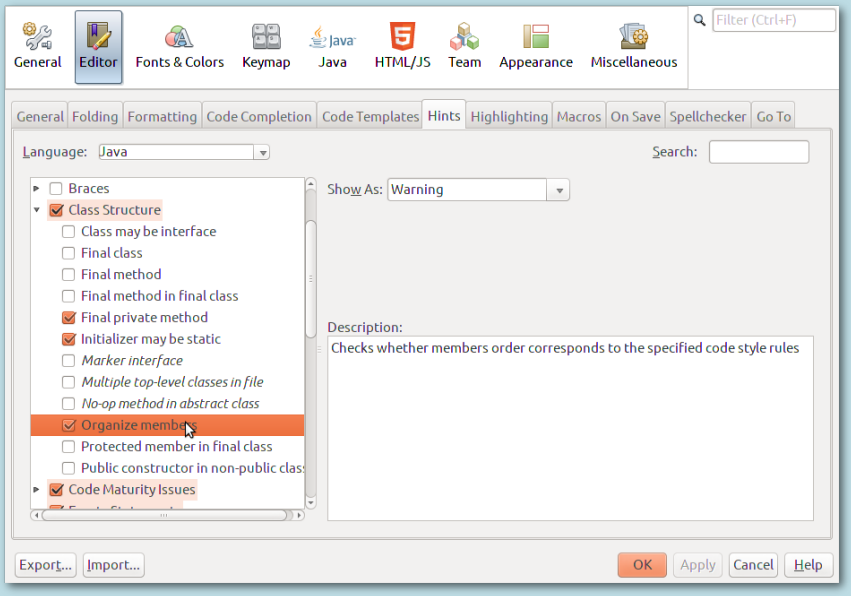
Astuce Netbeans

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.