



**UNIVERSITÉ
DE LORRAINE**



Patrons d'analyse et de conception

-

Dice Game

-

ARNOULD Maxime

CHEVRIER Jean-Christophe



Sommaire

1. Enoncé global	3
2. Choix.....	3
3. Fonctionnement	4
3.1. Fonctionnalités	4
3.2. Processus	5
3.2.1. Processus de « jouer »	5
3.2.2. Processus de « paramétrer son espace de stockage »	6
3.2.3. Processus de « consulter les meilleurs scores »	6
3.3. Scénarios de vie	7
3.3.1. Scénario de vie des dés	7
3.3.2. Scénario de vie du jeu	7
4. Architecture logicielle	8
4.1. Principes	8
4.2. Persistance des données	9
4.3. Système	12
4.4. Interface graphique	14
4.5. Global	16
5. Déploiement	17
5.1. Composants	17
5.2. Déploiement	17
6. Installation et exécution	18


1. Enoncé global

Dans le cadre de notre formation de Master 2 MIAGE SID, et pour le module de patrons d'analyse et de conception, nous avons reçu la mission d'implémenter une solution informatique devant répondre à un certain cahier des charges.

Le projet en question se nommait « **Dice Game** », et avait été fourni avec ce cahier des charges :

Dice Game

- Un jeu de dés
- Le joueur lance 10 x 2 dés
- Si le total fait 7, il marque 10 points à son score
- En fin de partie, son score est inscrit dans le tableau des scores.



Le principe était simple : la solution devait être un **jeu de dés**. Le jeu devait comprendre un joueur qui joue sur **10 tours**, en faisant **rouler à chaque tour 2 dés de 6 faces**. Le joueur gagnant a un tour **10 points si la somme des faces des 2 dés fait 7**. De plus, le joueur devait pouvoir **accéder à ses meilleurs scores** dans ce qui serait alors un tableau des scores.

En plus de ce cahier des charges, nous avons reçu d'autres d'instructions du maitre d'ouvrage M. BENALI :

- le joueur devait pouvoir **faire varier l'espace de stockage** où sont sauves ses scores ;
- en tant que maitre d'œuvre, nous devons proposer des **fonctionnalités supplémentaires** ajoutant encore de l'intérêt au jeu.

Nous verrons les fonctionnalités que nous avons ajoutés dans la partie 3. Fonctionnement.

2. Choix

Afin d'implémenter le projet qui nous avait été confié, nous avons fait certains choix de conception, ainsi que des choix techniques. Nous avons listé ces choix dans cette partie.

Tout d'abord, nous avons choisi de réaliser une conception orientée dès le départ en fonction de la stack / des technologies du projet. Pour expliciter, cela signifie que **notre conception n'est pas générique mais directement associée à la stack technique** que nous avons choisi.

Et justement, quelle stack technique avons-nous choisi ?

Nous avons choisi que le jeu soit en client lourd, donc pas sur le Web, pas en SaaS. Et dès lors, nous nous sommes dit que **Java 17 combinée avec Maven** serait une bonne stack pour faire un client lourd. Et pour développer un peu plus, nous avons choisi d'utiliser **Java Swing combinée à Java AWT**

pour faire l'interface graphique de la solution, car si bien manipulée ces librairies bien qu'anciennes font parfaitement l'affaire pour ce genre de petit projet.

Pour les espaces de stockage proposés au joueur, nous avons choisi de proposer ces derniers:

- **base de données PostgreSQL ;**
- **base de données MySQL ;**
- **base de données H2 ;**
- **sérialisation dans un fichier.**

Vous le verrez au fur et à mesure du rapport, nous avons fait la conception **en français pour les explications** dans les diagrammes, et **en anglais pour les composants, classes et objets** des diagrammes, car le code a été développée entièrement en anglais.

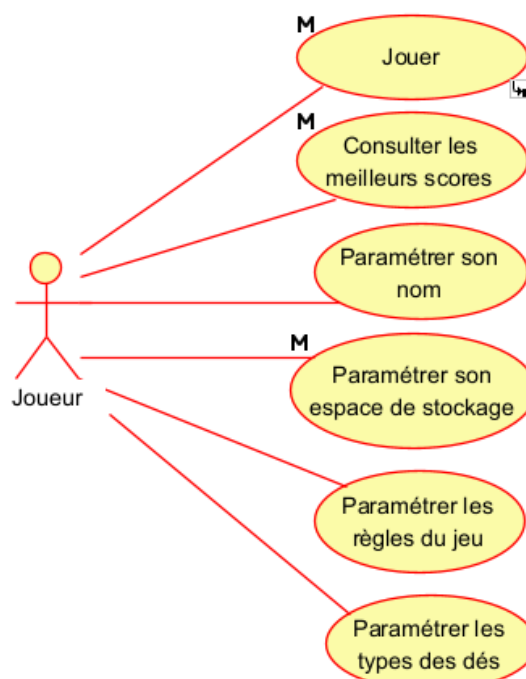
Enfin, dernier choix, le projet a été implémenté et documenté sur un **répertoire GitHub** : <https://github.com/jc-chevrier/dice-game>. Ce répertoire est public, vous pouvez y accéder sans problème, et nous vous le recommandons d'ailleurs.

3. Fonctionnement

Nous avons démarré la phase de conception en nous posant tout d'abord la question du fonctionnement global de la solution à faire.

3.1. Fonctionnalités

Nous avons dès lors commencer la conception, en se posant la question des fonctionnalités que devrait avoir notre solution. Les fonctionnalités étant la base du fonctionnement de tout logiciel. Nous avons de fait réalisé un diagramme de cas d'utilisation listant les fonctionnalités du projet. Ci-après le diagramme en question :



On peut voir qu'un seul rôle existera dans la solution : le joueur. Et on peut voir également les fonctionnalités que nous avons choisi d'ajouter, à savoir :

- **pouvoir paramétrer / changer les règles du jeu ;**
- **pouvoir paramétrer / changer les types des dés.**

Pour les règles du jeu, dans notre solution le joueur peut en effet choisir entre ces 3 types de règles :

- **10 points si somme des 2 dés égale à 7 ;**
- **10 points si somme des 2 dés inférieure à 7 ;**
- **10 points si mêmes faces pour les deux dés.**

Et pour les types de dés, le joueur peut choisir entre 3 types de dé, chaque type de dé correspondant à un niveau de difficulté :

- **dé à 6 faces : faces de 1 à 6, difficulté normale ;**
- **dé à 10 faces : faces de 0 à 9, difficulté difficile ;**
- **dé à 20 faces : faces de 1 à 20, difficulté expert.**



3.2. Processus

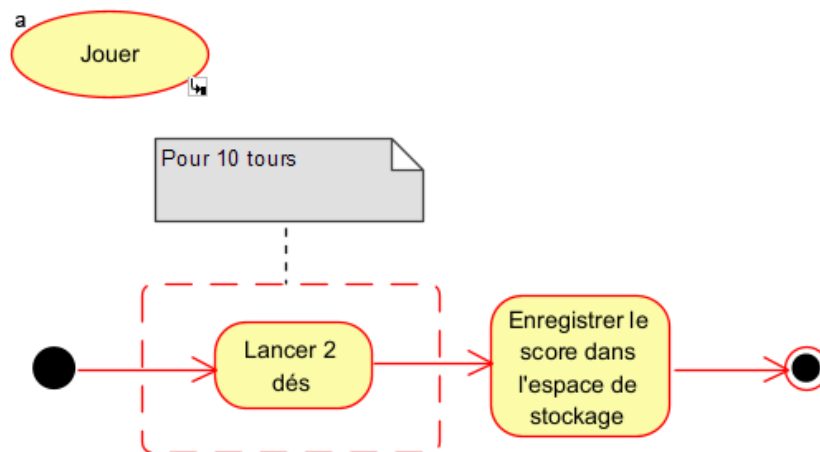
Une fois que nous avons planifié les fonctionnalités, nous avons réalisé des diagrammes d'activité. Et cela, afin de détailler les processus des fonctionnalités dont le fonctionnement n'est pas évident, ou est plus complexe qu'il ne le semble.

Ainsi, nous avons réalisé des diagrammes d'activité pour 3 des fonctionnalités parmi les 6 prévues dans la solution.

3.2.1. Processus de « jouer »

Tout d'abord, nous avons voulu décrire le fonctionnement de la fonctionnalité au centre du jeu, à savoir : « jouer ».

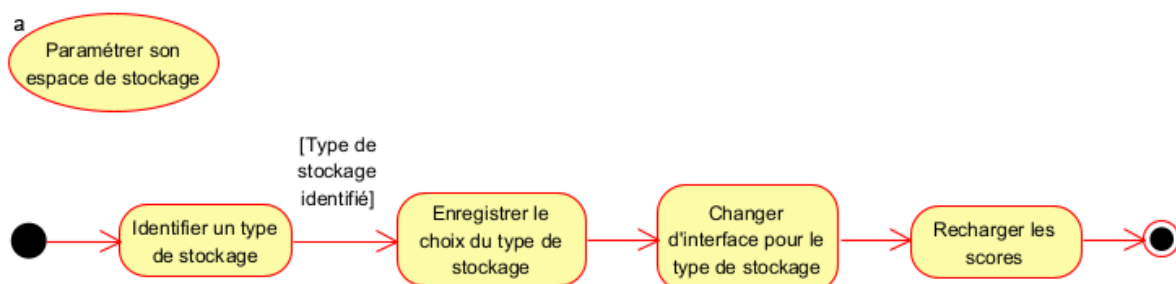
Ci-après le diagramme d'activité de « jouer » :



3.2.2. Processus de « paramétrer son espace de stockage »

Puis, il nous a semblé que « paramétrer son espace de stockage » était une fonctionnalité qui valait la peine d’être décrite plus dans le détail, car pas forcément évidente du point de vue de l’implémentation.

Ci-dessous donc le diagramme d’activité de « paramétrer son espace de stockage » (il peut être nécessaire de bien zoomer) :

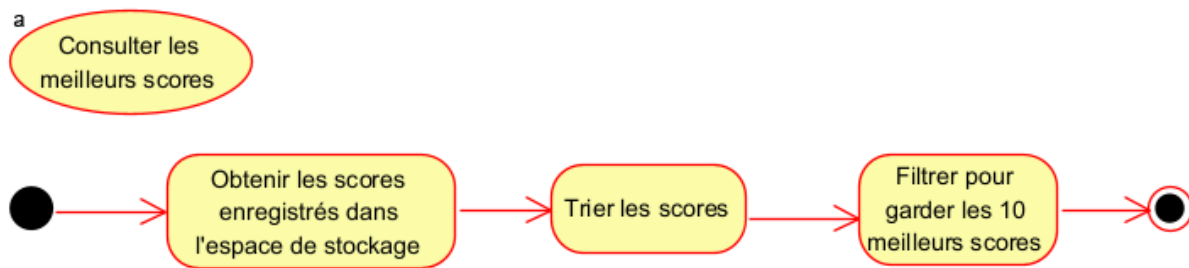


On peut remarquer une chose intéressante ici : la solution utilise des interfaces pour communiquer avec les différents espaces de stockage, et au changement de l’espace de stockage la solution change d’interface pour celle adéquate.

3.2.3. Processus de « consulter les meilleurs scores »

Enfin, « consulter les meilleurs scores » nous a semblé nécessiter aussi quelques précisions, nous avons donc également fait un diagramme d’activité pour.

A la page qui suit, vous pouvez voir le diagramme d’activité de « consulter les meilleurs scores ».



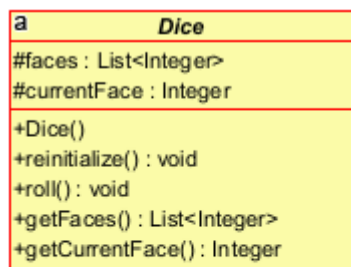
3.3. Scénarios de vie

Afin de toujours rendre moins abstrait le fonctionnement de la solution, nous réalisons des diagrammes pour détailler les scénarios de vie des principales « entités » de cette dernière.

Nous avons entre autres ici fait des diagrammes d'état-transition.

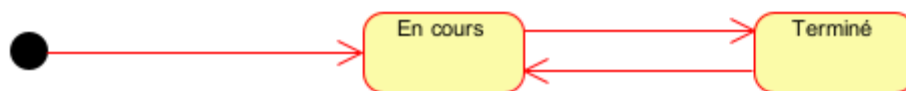
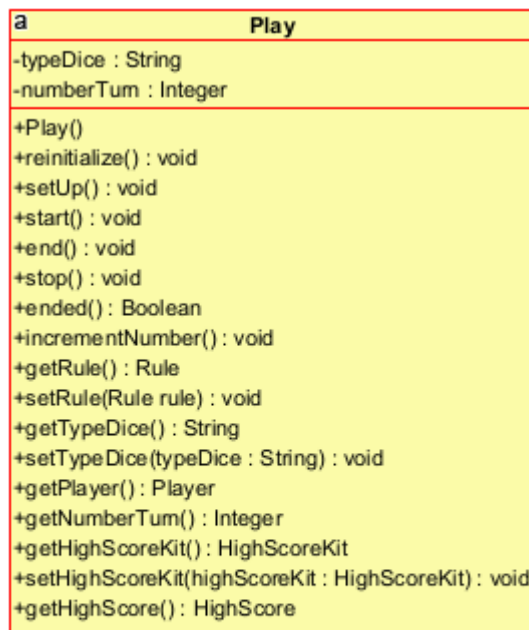
3.3.1. Scénario de vie des dés

Tout d'abord, nous avons fait un diagramme d'état-transition des dés afin de décrire le fonctionnement global des dés. Et il est assez basique comme vous vous en doutez. Voici le diagramme :



3.3.2. Scénario de vie du jeu

Ensuite, nous en avons fait de même pour le jeu, afin de décrire son fonctionnement très global, son scénario de vie donc. A la page qui suit vous pouvez voir le diagramme en question.

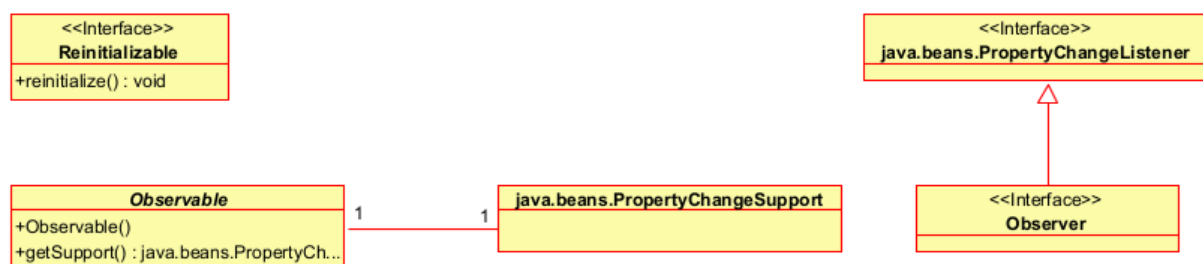


4. Architecture logicielle

Une fois que nous avons terminé de prévoir les fonctionnalités du jeu, nous nous sommes intéressé à l'**architecture logicielle** concrète qu'aurait l'implémentation de notre solution. De fait, nous avons réalisé des diagrammes de paquetage ainsi que des diagrammes de classes.

4.1. Principes

Tout d'abord, nous avons créé un package pour déclarer certains **principes de notre solution** dont des patrons de conception. Il s'agit du package « principe ». Voici un diagramme de classes le décrivant :



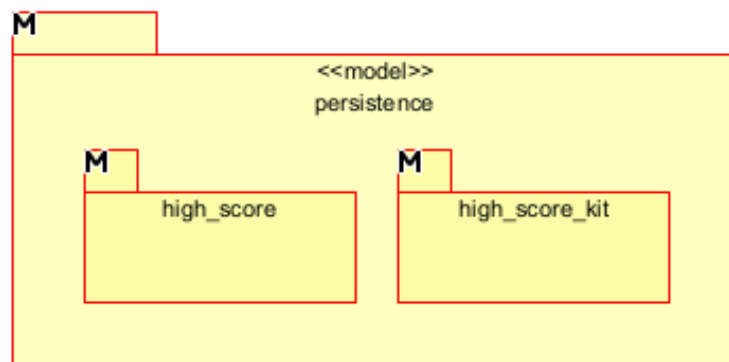
« Reinitializable » est une interface pour les **familles d'objets de la solution capable de se réinitialiser**.

« Observable » et « Observer » sont des classe abstraite et interface implémentant le patron de conception du même nom. Ces programmes Java sont en fait des **surcouches de « PropertyChangeSupport » et « PropertyChangeListener »**, comme vous pouvez le voir sur le diagramme.

4.2. Persistance des données

Afin de prévoir comment la solution **sauvegarde les données**, et également afin de prévoir comment la solution **s'adapte aux différents espaces de stockage**, nous avons prévu un package « persistence ». Comme son nom l'indique, il a la responsabilité d'assurer le **persistance des données** et entre autres des scores passés des joueurs.

Voici tout d'abord un diagramme de paquetage du package « persistence » :



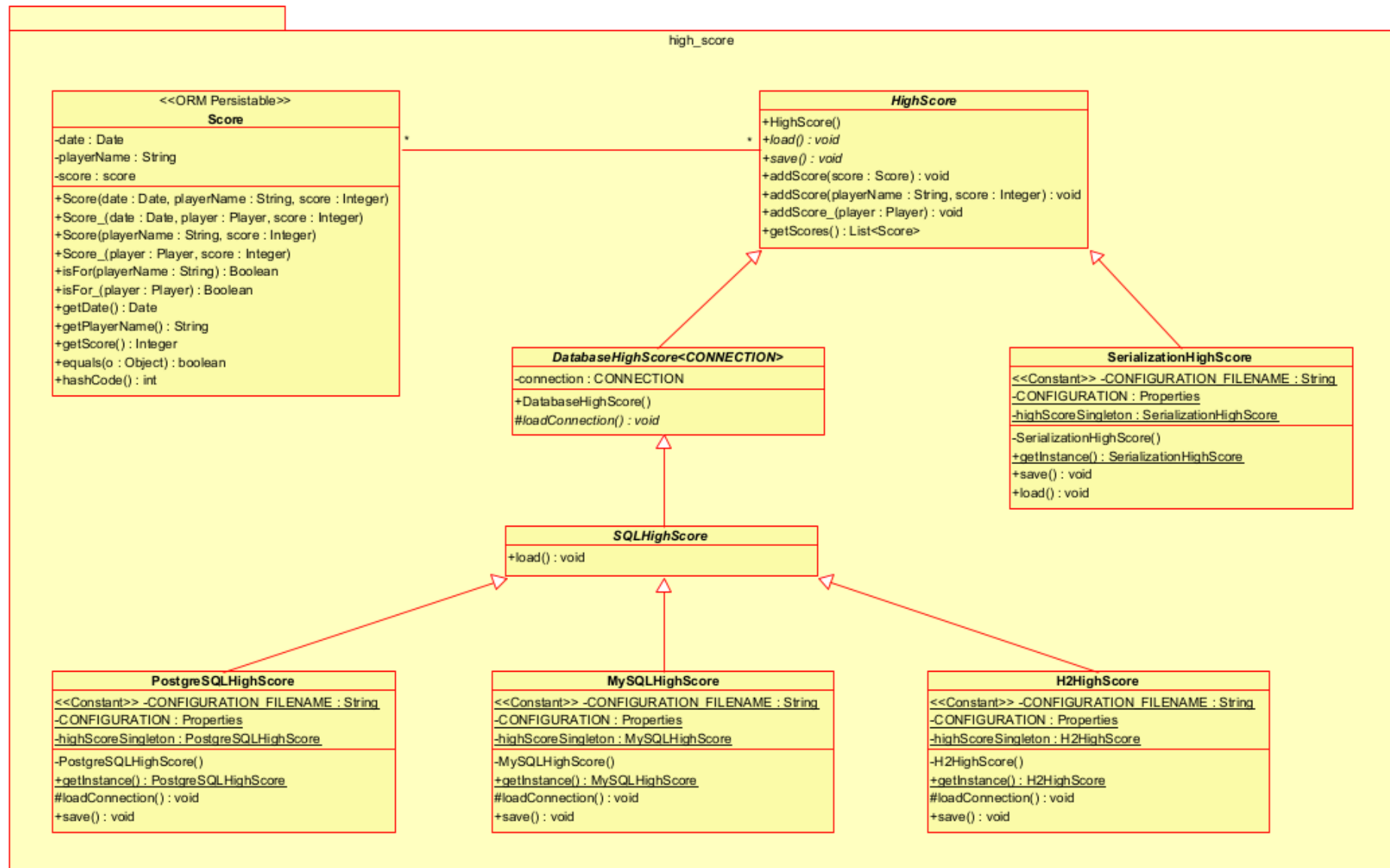
On peut voir que le package est lui-même composé de sous-package :

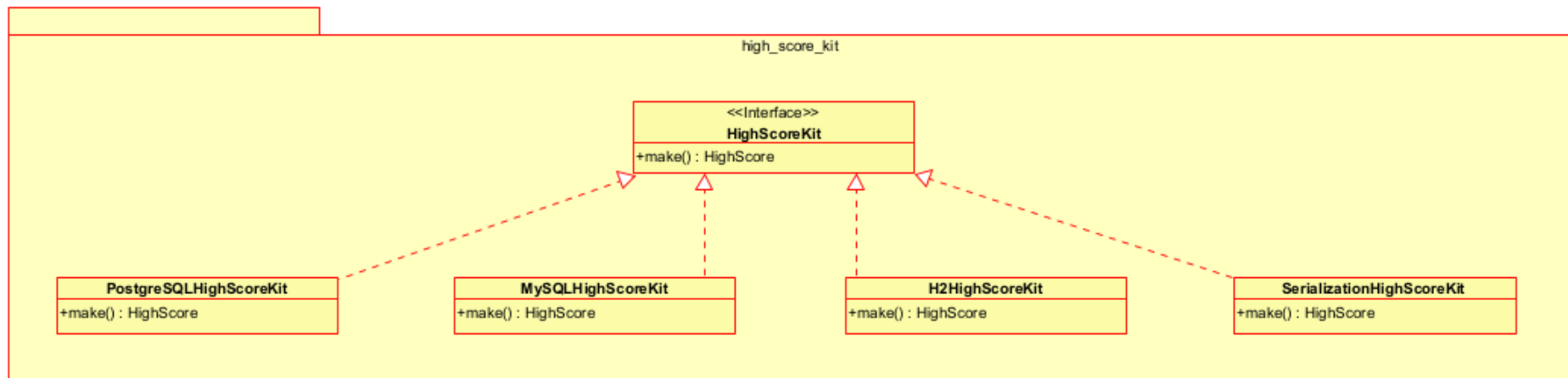
- Un sous-package « high_score » pour les classes de **représentation des données**, et pour les classes faisant les **interfaces avec les différents de espaces de stockage** (interface au sens interface entre système ici, pas interface Java).
- Un sous package « high_score_kit » pour varier entre espace de stockage en toute abstraction.

A la page qui suit, vous pouvez voir le diagramme de classes du package.

Entre autres, on peut voir que le package « high_score » contient des classes implémentant les **patrons de conception Factory et Singleton**. En effet, toutes les classes héritant de la classe « HighScore » implémentent ces deux patrons. Il y a dans ce package, autant de classes concrètes « [...]HighScore » que d'espace de stockage dans le jeu. En effet, chaque espace de stockage a une classe lui correspondant pour implémenter l'interface entre Java et les serveurs des bases de données / le système de fichiers pour la sérialisation.

En revanche, la classe « Score » est un **ORM Persistable**, c'est-à-dire une entité en reflet de la table « Score » des bases de données.

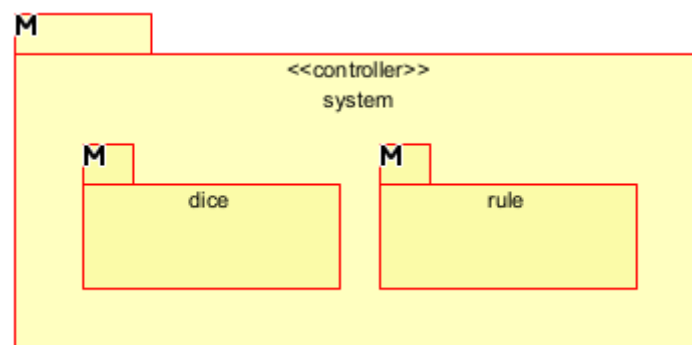




4.3. Système

Dans l'objectif de prévoir comment le système du jeu fonctionnera précisément, nous avons réalisé un package « system », constituant le **cœur de notre solution**. Par ailleurs, un autre nom pour ce package aurait pu être « core ». Le but de package est vraiment de contenir les **programmes qui dirigent le jeu**. La responsabilité de ce package est donc la **gestion des fonctionnalités en elles-mêmes du jeu**.

Ci-après, le diagramme de paquetage du package « system » :



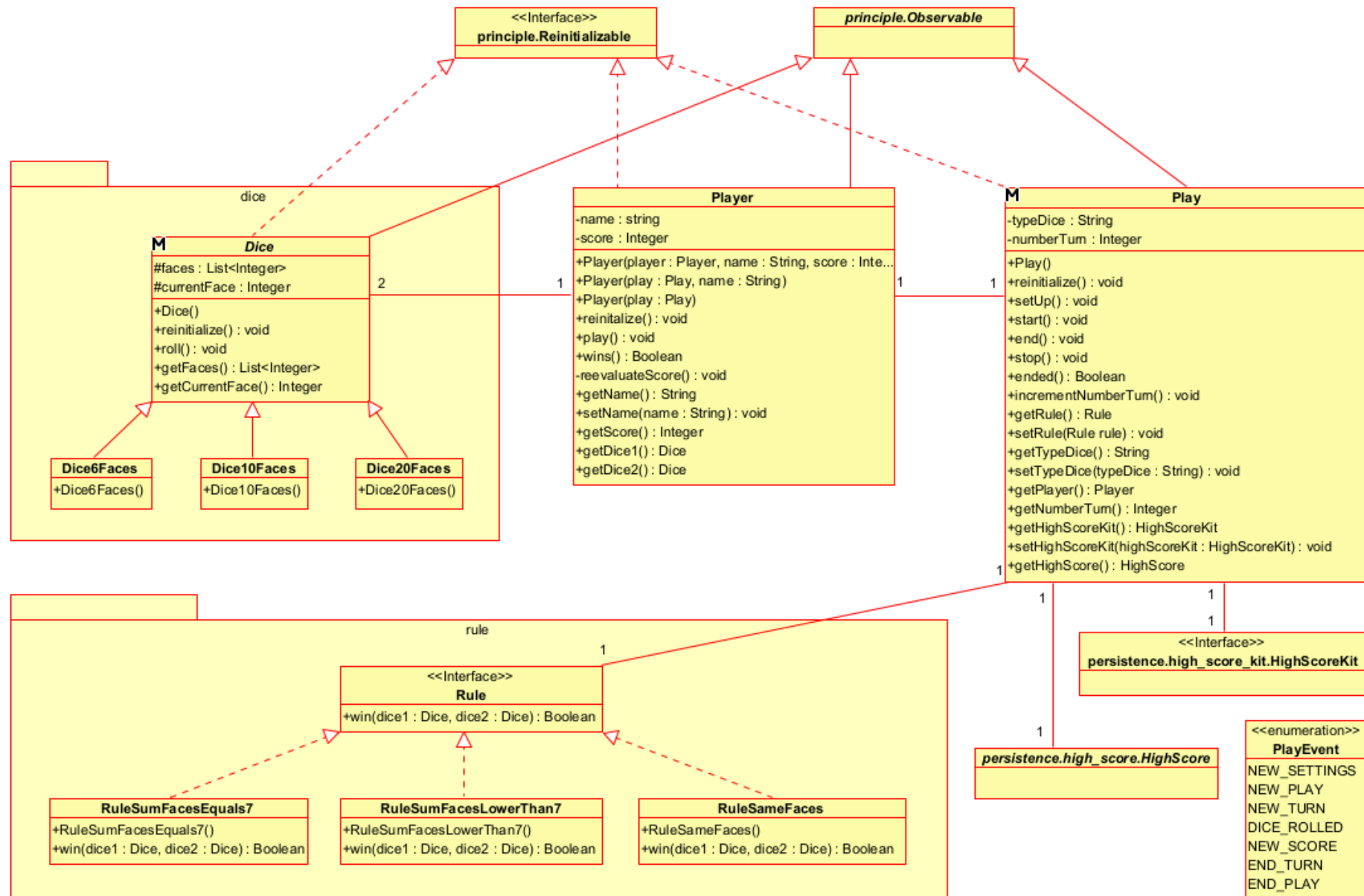
On peut voir que ce package est également composé de sous-package :

- Un sous-package « dice » pour la **famille des dés du jeu** : les dés à 6 faces, les dés à 10 faces et les dés 20 faces. Les classes de ce package utilisent le **patron de conception Strategy**.
- Un sous-package « rule » pour la **famille des règles du jeu** : règle de somme des faces égale à 7, règle de somme des faces inférieure à 7, règle des mêmes faces. Les classes de ce package utilisent également le **patron de conception Strategy**.

A la page suivante, vous pouvez voir le diagramme de classes du package « system ».

On peut voir, que « Play », « Player », et « Dice » sont des **classes réinitialisables et des observables**, le **patron de conception Observer-Observable** est utilisé ici.

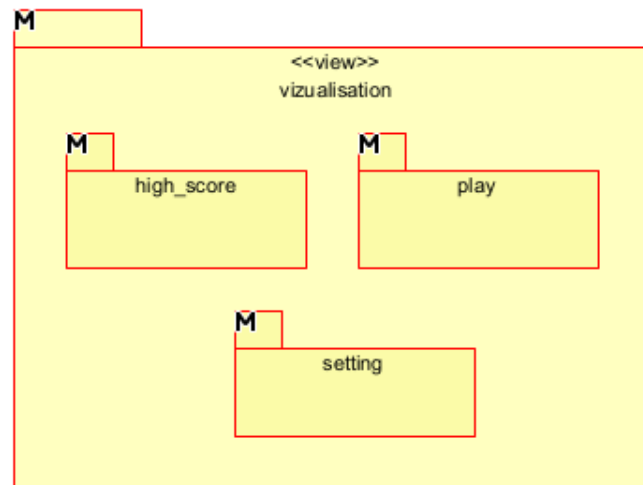
Autre chose intéressante, on peut voir une énumération appelée « PlayEvent ». Cette énumération est en fait l'objet utilisé par les observables de ce package, pour **différencier les événements qu'ils notifient à leurs vues** du package « visualisation ». Par exemple, un objet « Dice6Faces » notifie toujours ses vues avec cette valeur de l'énumération comme événement : « DICE_ROLLED ».



4.4. Interface graphique

Afin de prévoir l'interface graphique qu'aura notre jeu, nous avons réalisé un package « vizualisation », ayant donc la responsabilité du **visuel du jeu, de l'expérience UX / UI donné au joueur**.

Ci-après, le diagramme de paquetage du package « system » :



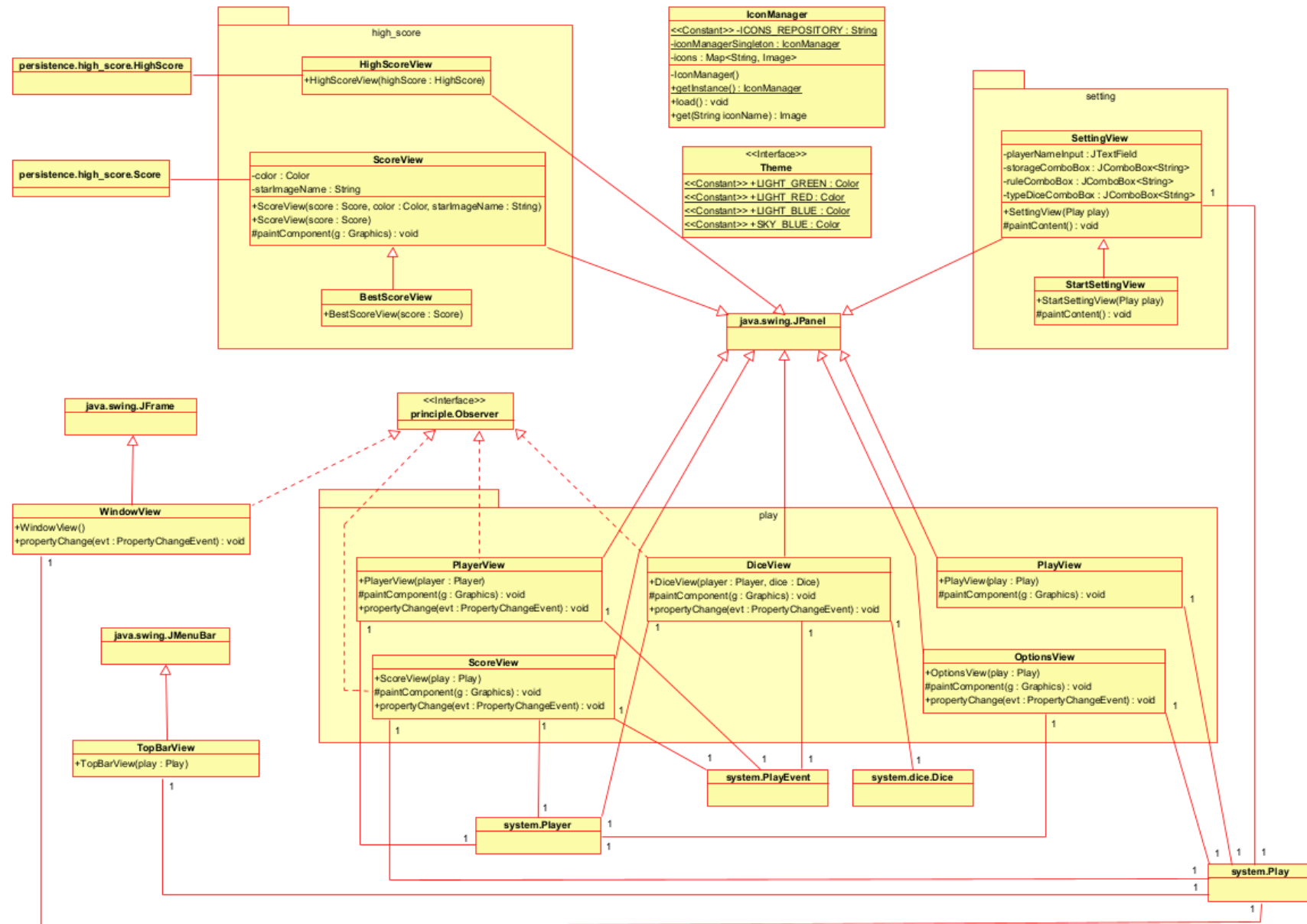
On peut voir que ce package est également composé de sous-package :

- Un sous-package « high_score » pour les classes de la **page des meilleurs scores** de l'interface graphique. Les classes de ce package implémentent un **patron de conception Decorator**.
- Un sous-package « play » pour les classes de la page du jeu en lui-même de l'interface graphique.
- Un sous-package « setting » pour les classes de la **page des paramètres** de l'interface graphique. Les classes de ce package implémentent également un **patron de conception Decorator**.

Toujours même principe, à la page qui suit, vous pouvez voir le diagramme de classes du package.

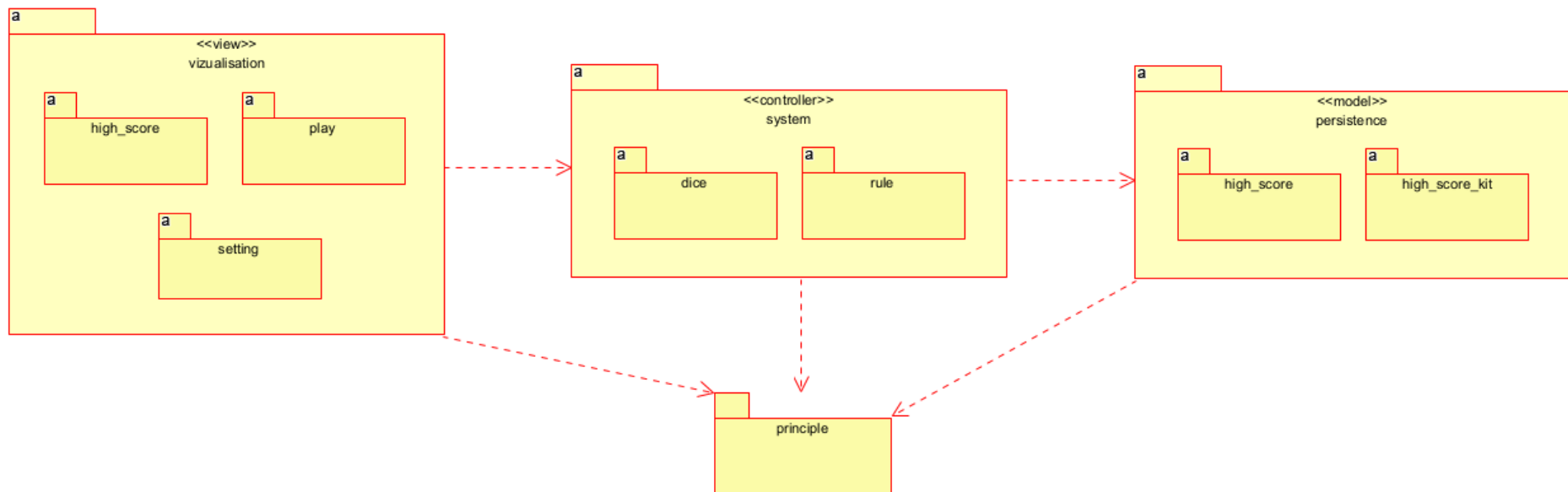
On peut voir que les classes « **WindowView** », « **PlayerView** », « **DiceView** », et « **ScoreView** » (« **play.ScoreView** ») sont des observateurs, le **patron de conception Observer-Observable** est utilisé ici.

On peut voir également que **la responsabilité de gestion et de chargement des images est alléguée à la classe « IconManager »**, qui par ailleurs implémente le **patron de conception Singleton**. Le chargement des images est fait une seule fois par cette classe, au lancement du jeu, et ce pour des raisons de performance.



4.5. Global

Ci-dessous, vous pouvez voir un diagramme de paquetage de l'ensemble de l'architecture logicielle, on peut voir que celle-ci implémente le **patron de conception MVC** :



Et ici, vous pouvez voir les 3 petites lignes qui démarre la solution, on peut voir que la classe principale « Main » emploie donc un **patron de conception Facade** pour lancer la solution :

```

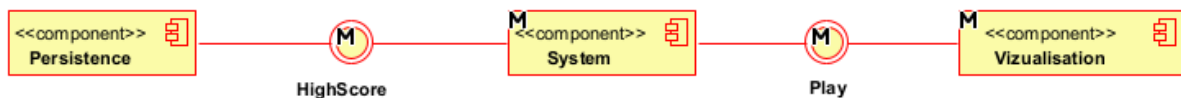
public class Main {
    public static void main(String[] args) {
        Play play = new Play();
        new Window(play);
        play.setUp();
    }
}
    
```


5. Déploiement

Une fois que nous avons conçu et planifié le fonctionnement et l'architecture de la solution, nous nous sommes posé la question du déploiement de cette dernière.

5.1. Composants

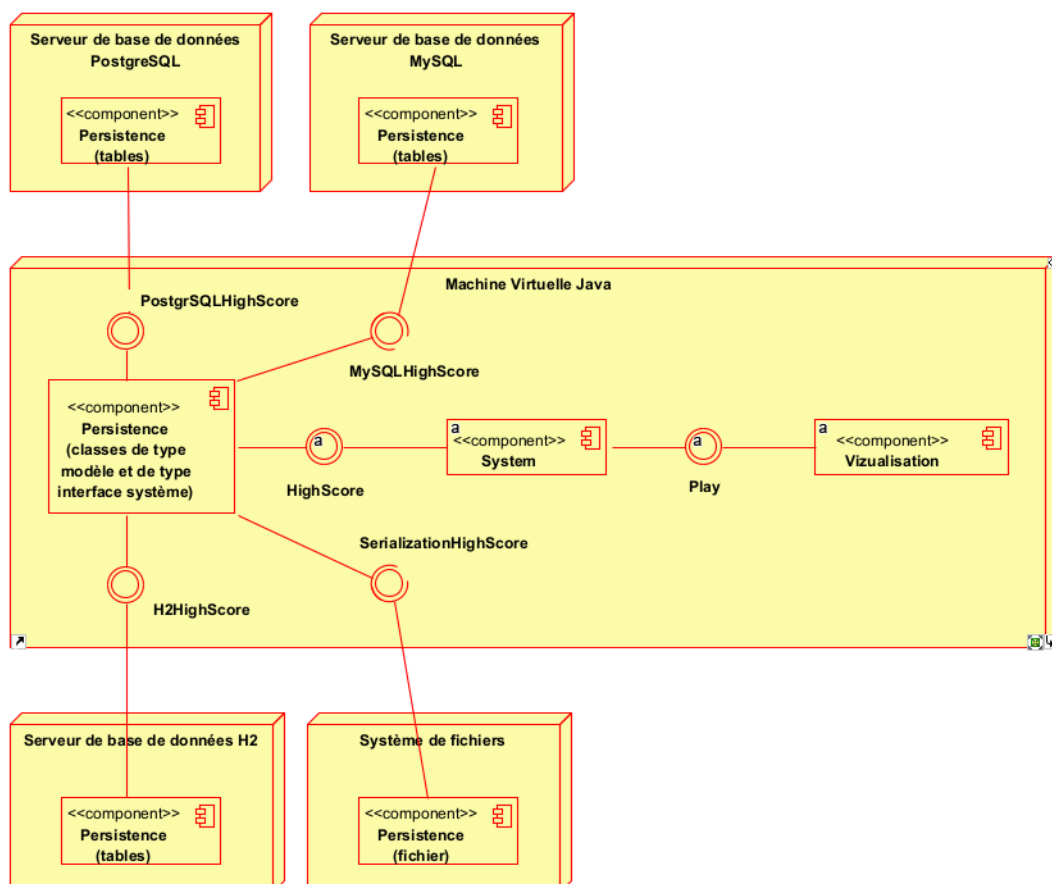
Avant tout, quels sont les composants de notre solution ? Ils sont en fait peu, et nous les avons prévu dans un diagramme de composants :



Il s'agit du découpage fonctionnel MVC assez classique, chaque composant correspondant à un package de la solution.

5.2. Déploiement

Par la suite, les choses se compliquent, lorsque l'on passe en effet des composants, au déploiement sur les plateformes et stacks techniques de la solution. Afin de planifier le déploiement, nous avons réalisé un diagramme de déploiement :



6. Installation et exécution

Si vous souhaitez utiliser notre solution, nous avons rédigé un « **mode d'emploi d'installation et d'exécution** » dans le readme du projet.

Ce readme est accessible sur le répertoire GitHub du projet, et cela depuis cet hyperlien : <https://github.com/jc-chevrier/dice-game/blob/main/readme.md>. Une fois sur la page Web en question, il faut se rendre à la section du readme : *Installation et exécution*.