# DEV

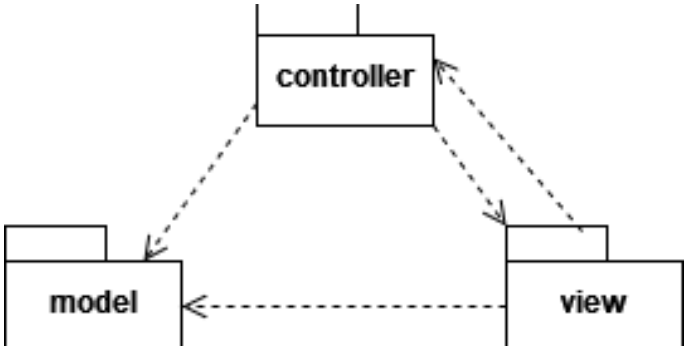
# DOCUMENTATION

| Morpion Solitaire | | Jeu terminé le |
|---|---|---|
| Batal Ilyess - Sengutuvan Arvinde | | 15 décembre 2023 : V0 |
| Jeu fait en Java 17 | | |

## Environnement de travail

Pour ce projet, nous avons utilisé Java 17 avec Maven. La mise en place d'une interface graphique est assurée par JavaFX. Pour les tests unitaires, nous utilisons JUnit 5.
De manière plus exotique, nous utilisons Microsoft Azure SQL Database (avec son Driver JDBC associé) pour le traitement de données persistantes.
Nous avons également importé par curiosité Spring Security, une bibliothèque qui fournit une implémentation d'un algorithme de cryptage robuste (BCrypt) utilisé pour l'authentification d'un utilisateur.

## Architecture MVC



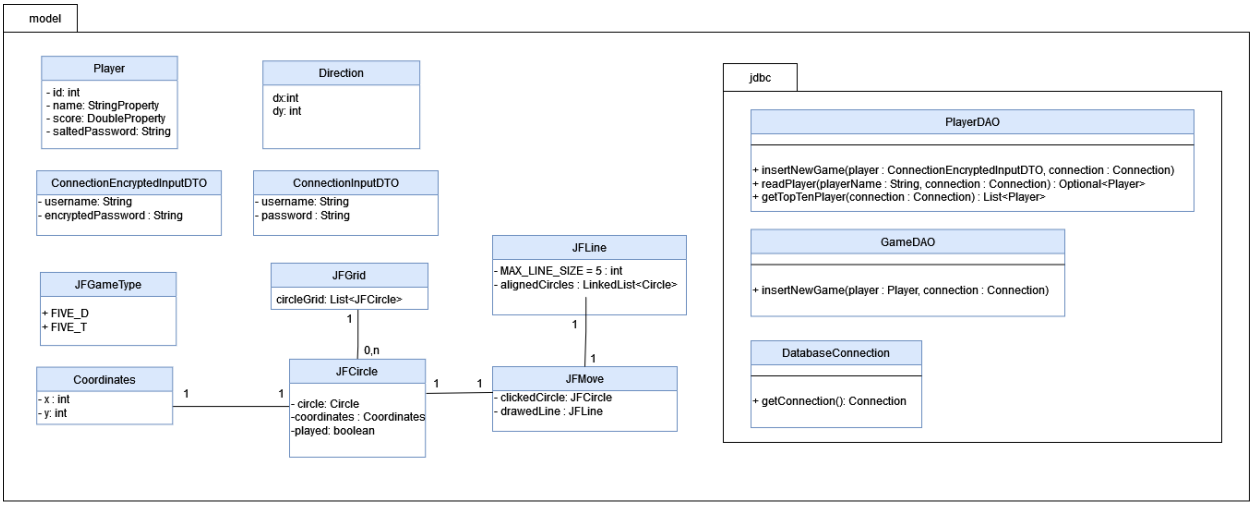

L'architecture MVC permet de réaliser un découpage du projet clair et simple d'un projet entre la partie contrôleur, vue et modèle. Chaque partie à une responsabilité définie, et interagit avec les autres:

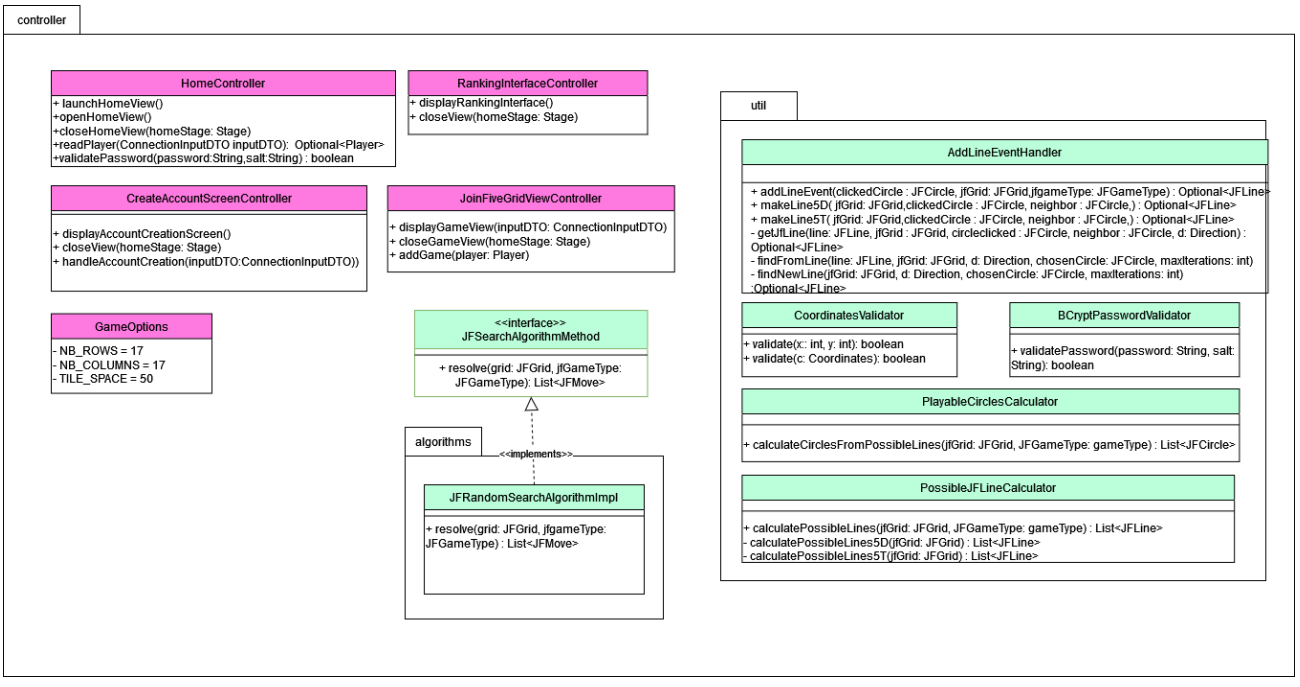
- Le modèle joue un rôle de représentation des données métier. Il à également la responsabilité de permettre l'accès et la modification de ces données de manière persistante (c'est-à-dire de pouvoir conserver une donnée même après avoir fermé l'application).

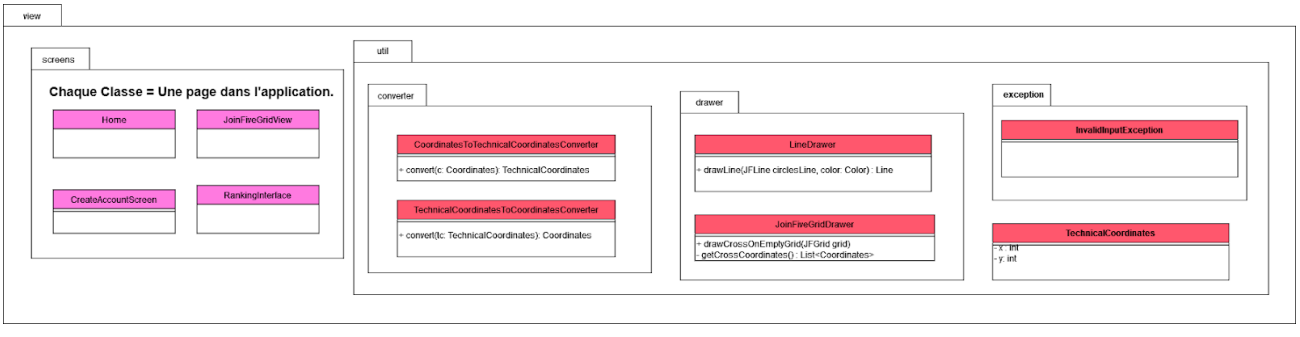
- La vue est la couche de présentation qui interagit directement avec l'utilisateur de l'application
- Le contrôleur est la couche qui implémente l'intelligence de l'application, avec notamment la logique du jeu.

Nous avons choisi cette architecture car elle était simple à mettre en place, et facilement compréhensible par d'autres développeurs (que ce soit pour vous lors de l'évaluation du projet ou bien pour d'éventuels futurs collaborateurs sur le projet...). En effet, nous avons vu qu'elle permet de découper la complexité de l'application en 3 parties distinctes, la rendant ainsi plus robuste, facilement testable, et maintenable.
Effectivement, si nous souhaitons adapter notre projet en webapp (ex: projet Spring Boot, Quarkus), nous aurons seulement à toucher la partie view du projet, avec de légères modifications sur la partie controller pour gérer la navigation dans l'application.
Inversement, si un jour nous estimons que notre base de données Azure devient trop chère, on pourra facilement la changer en modifiant la partie modèle. Dans le cas où on restait sur une base SQL, nous n'aurions en réalité pas besoin de modifier notre code  car on utilise une abstraction avec JDBC, installer de nouveaux drivers aurait été suffisant. Cela serait plus intéressant si on passait sur du NoSql par exemple, qui demanderait un effort de conception plus conséquent. Cela témoigne de la flexibilité apportée par l'architecture.

# Répartition du travail






Lors de la phase de conception du projet, nous avons établi ensemble un diagramme de classe, clarifiant ainsi les responsabilités de chaque classe.

Afin de mener à bien ce projet, nous avons divisé notre travail en plusieurs phases. Tout d'abord, nous avons entamé la phase de conception, établissant ainsi un diagramme de classe pour mieux comprendre ce que nous allions développer.

Par la suite, nous avons renforcé notre confiance dans la partie interface en commençant le développement d'une grille et d'un menu en JavaFX. Suite à la première réunion de suivi, nous avons pris en compte vos remarques sur notre interface (le menu et l'écran de jeu étaient originellement divisés en deux écrans distincts, ce qui n'était pas facile à utiliser pour l'utilisateur).

En conséquence, nous avons procédé à l'implémentation de toutes les fonctionnalités en termes de logique, pour pouvoir jouer au jeu. Pour terminer, nous avons intégré la base de données et les fonctionnalités qui lui sont associées.

Au commencement de notre projet, nous avons initié une division des responsabilités, attribuant à Ilyess la conception de l'interface graphique et à Arvinde le développement de la logique du jeu. Cependant, conscient de l'importance d'une coordination , nous avons rapidement décidé d'adopter une approche collaborative. Cette décision visait à travailler conjointement sur tous les aspects du projet simultanément et ensemble, favorisant ainsi une progression plus rapide et une synergie accrue entre l'interface visuelle et la mécanique du jeu. Cette stratégie a renforcé notre efficacité et a permis

une meilleure intégration des différents éléments, contribuant ainsi à la réussite globale de notre travail.

## Fonctionnalités réalisées

Le projet respecte les spécifications demandées:

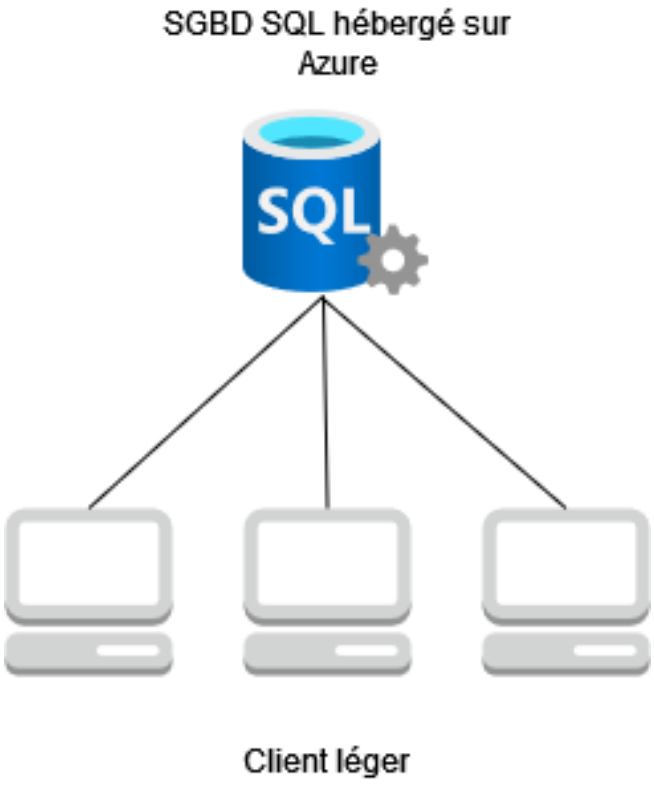
- Il est possible de jouer une partie dans son intégralité
- Appliquer un algorithme random, pour que l'ordinateur puisse finir une partie
- Obtenir une aide en cas de blocage (si il est possible de continuer)
- Recommencer une partie
- Annuler une action
- Comparer ses scores avec d'autres joueurs, avec l'affichage d'un tableau actualisé contenant le top 10 des joueurs.

L'ajout d'une base de données a permis la réalisation des fonctionnalités complémentaires suivantes:

- Connexion à l'application
- Inscription dans l'application
- Afficher une liste du top 10 actualisée des joueurs
- Sauvegarder les scores de parties des joueurs de manière persistante

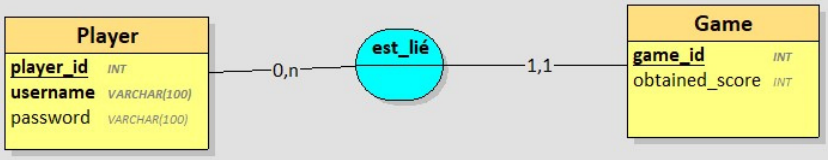
## Persistance des données : BD

Pour rappel, notre projet utilise une base de données SQL intégrée sur le cloud MS Azure. Les clients légers (ceux qui lancent l'application JavaFx) interagissent directement avec la base SQL avec des requêtes, grâce aux pilotes JDBC fournis par Microsoft.




**MCD:**

On peut représenter notre base de données avec le modèle conceptuel suivant:




**MLD:**

Le MLD utilisé pour la création des tables en SQL est le suivant, les champs soulignés représentent les clé primaires et ceux précédés par un # correspondent aux clés étrangères.
**Player(player_id, username, password);**

**Game(game_id, obtained_score, #player_id);**

## 6) Qualité du code

Sur la qualité du code, nous avons mis en place des bonnes pratiques de programmation, notamment en respectant SOLID. En effet, nos classes ont chacune une responsabilité à la fois limitée et propre,

ne dépendent pas de cas très concrets et peuvent être extensibles. Cela favorise la lisibilité et la maintenabilité de notre projet dans le futur.
Sur le code en lui-même, l'architecture MVC nous a permis de favoriser l'usage de la délégation pour pouvoir communiquer entre deux couches et séparer la responsabilité de chaque classe. Nous nous sommes également renseigné sur des mécanismes fournis par le langage Java comme Optional pour éviter les NullPointerException. De plus, nous avons mis en place des designs patterns comme Builder pour faciliter la création d'objets pouvant êtres instanciés de plusieurs manières différentes (ex: la classe Player).
Le code écrit est commenté et couvert par une multitude de cas de tests (notamment pour la partie controller). Pour la réalisation des tests, nous avons suivi une approche BDD (Behaviour Driven Development). Chaque test unitaire est divisé en 3 parties distinctes:

- On initialise les données pour le test
- En appliquant la méthode à tester sur les données...
- On s'attend à un résultat clair et défini

Cette approche permet d'améliorer la clarté des scénarios de tests, et de vérifier que notre produit répond bien aux exigences demandées.

## Difficultés principales du projet

Les difficultés principales du projet ont été de mettre en place de l'architecture de manière propre pour faire en sorte de garder un maximum de code stable (la partie modèle et contrôleur en particulier). Le travail sur l'interface graphique (la partie view du projet) a également nécessité un effort particulier, car nous n'étions pas expérimentés avec JavaFX et les possibilités offertes par le framework. Nous sommes fiers du travail accompli mais nous pensons qu'il a nécessité plus de temps qu'initialement prévu.