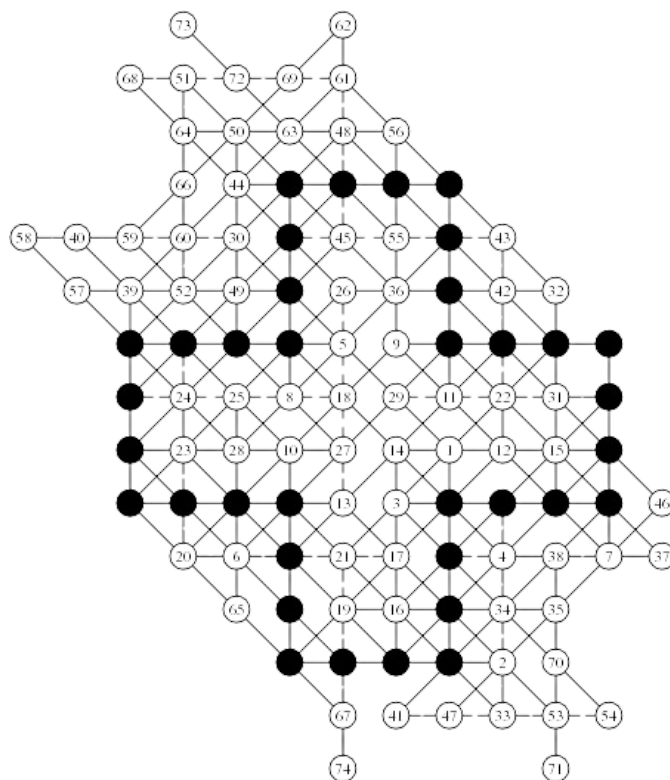


Rapport d'implémentation : Morpion Solitaire en Java

par Paul Malet et Jeanne-Emma Lefèvre



Sommaire

Sommaire	1
I. Le diagramme de classes	2
II. Structure du projet et architecture Modèle Vue Contrôleur	5
A. Le MVC	5
1. Modèle	5
2. La Vue	6
3. Le Contrôleur	7
B. Les algorithmes de recherche	8
1. Méthode de recherche de solution automatique aléatoire	8
2. Méthode de recherche de solutions avec l'algorithme NMCS	8
C. Les tests	9
III. Stack technique	9
A. Maven	9
B. Stockage des données	10
C. Interface	11
Conclusion	12
Annexes	13
Diagrammes de classes par packages	13
1. app	13
2. controler	14
3. model	15
4. nmcs	15
5. view	16
6. modelTests	17
7. NMCSTests	17

I. Le diagramme de classes

Le diagramme de classes présenté ci-dessous modélise l'architecture de notre Morpion Solitaire, développé en Java avec l'utilisation de JavaFX et Maven. Cette conception met en œuvre divers concepts de programmation orientée objet étudié dans le cadre de notre cours de programmation objet avancée tel que des interfaces, des classes abstraites, des héritages et des agrégations.

Interfaces :

- L'interface *Serializable* est implémentée par les classes clés telles que *MorpionSolitaireController*, *MorpionSolitaireModel*, et *MorpionSolitaireView*, indiquant qu'elles peuvent être sérialisées, ce qui est essentiel pour la sauvegarde et le chargement de l'état du jeu.
- L'interface *Mode* représente les différents modes de jeu, avec des implémentations spécifiques telles que *FT* (Five Tuple) et *FD* (Five Diamond).

Classes abstraites :

- La classe abstraite *Record* fournit une base pour les enregistrements de grille sauvegardés, favorisant la modularité et l'extensibilité du système.

Héritages :

- Les classes principales telles que *MorpionSolitaireController*, *MorpionSolitaireModel*, et *MorpionSolitaireView* héritent de l'interface *Serializable*, assurant ainsi une gestion appropriée de la sérialisation pour ces composants essentiels du jeu.

Agrégation et composition :

- *MorpionSolitaireController* contient des instances de *MorpionSolitaireModel* et *MorpionSolitaireView*, illustrant un modèle d'aggrégation/composition pour la gestion du modèle et de la vue du jeu.
- *MorpionSolitaireModel* contient une instance de *Grid* et d'autres attributs nécessaires à la gestion de l'état du jeu.
- *MorpionSolitaireView* intègre un composant *Canvas* pour afficher graphiquement le jeu.

Dépendance :

- Une dépendance est établie entre *MorpionSolitaireView* et *Theme*, où la vue dépend du thème pour des aspects visuels spécifiques.

Association :

- Les classes *MorpionSolitaireModel* sont associées à des observateurs, implémentant les interfaces *PlayObserver* et *ScoreObserver*, pour réagir aux changements dans le jeu et dans le score respectivement.

Héritage Multiple :

- La classe *SauvegardeGrille* illustre un exemple d'héritage multiple, en implémentant l'interface *Serializable* et en héritant de la classe abstraite *Record*.

Utilisation des énumérations :

- Les enums *Mode* et *Direction* sont utilisés pour représenter respectivement les modes de jeu et les directions possibles pour les lignes.

Utilisation de Listes et Collections :

- La classe *Grid* utilise des collections pour gérer les points joués et les lignes créées sur le plateau.

Utilisation de Stack :

- La classe *App* utilise une structure de données de type *Stack* pour gérer l'historique des pages dans l'interface utilisateur, offrant une navigation efficace.

Encapsulation :

- Les classes encapsulent leurs attributs en fournissant des méthodes d'accès et de modification, assurant un contrôle approprié sur l'accès aux données internes.

Gestion des utilisateurs :

- La classe *UserManager* centralise la gestion des utilisateurs, offrant des fonctionnalités telles que la création, l'authentification, et la sauvegarde des utilisateurs, contribuant ainsi à la sécurité et à la personnalisation du jeu.

En conclusion, notre diagramme de classes illustre une structure robuste et modulaire pour notre jeu Morpion Solitaire. Pour plus de lisibilité vous pouvez retrouver les diagrammes de classes par *package* en annexes.

II. Structure du projet et architecture Modèle Vue Contrôleur

Notre projet Morpion Solitaire adopte une architecture couramment utilisée dans la conception logicielle, mettant en avant le patron de conception modèle-vue-contrôleur (MVC). Cette structure organisée se décline en packages distincts pour chaque composant essentiel : le modèle, la vue, le contrôleur, l'application, et la recherche du meilleur état du jeu. En parallèle de ces packages dans notre dossier source (*src*), nous avons dédié un répertoire aux ressources, utilisé pour stocker des fichiers et des données qui ne font pas partie du code source, mais qui sont nécessaires à l'exécution de l'application ce qui inclut nos images et fichiers *.xml*, ainsi qu'un dossier spécifique pour nos tests unitaires.

A. Le MVC

1. Modèle

Notre architecture de modèle comprend au total 14 classes, dont la totalité ne sera pas explicitée dans ce rapport. Vous pouvez consulter la documentation générée par JavaDoc pour une compréhension détaillée du fonctionnement de nos méthodes. Néanmoins, nous allons approfondir certains points que nous considérons comme essentiels en complément de la documentation fournie. Les classes incluses dans le modèle sont les suivantes :

- *BlockedDirectionPoint.java*
- *Direction.java*
- *Grid.java*
- *Line.java*
- *Mode.java*
- *MorpionSolitaireModel.java*
- *PlayObserver.java*
- *Point.java*
- *SauvegardeGrille.java*
- *Score.java*
- *ScoreObserver.java*
- *ScoreSauvegarde.java*
- *User.java*
- *UserManager.java*

Déroulement du jeu et classe de modèle

Les classes *Grid*, *Line* et *Point* constituent le cœur du jeu en gérant la logique associée. Ces objets sont ensuite réutilisés dans notre classe *MorpionSolitaireModel*, qui encapsule la logique métier du jeu Morpion Solitaire et agit comme une interface entre la représentation interne du jeu (la grille, les lignes, etc.) et les composants externes tels que l'interface utilisateur et le contrôleur de connexion.

Sauvegarde et gestion des données

L'aspect utilisateur est pris en charge par la classe *UserManager*, qui gère les utilisateurs et l'authentification, stockant ces informations dans un fichier *JSON*. Cela simplifie la gestion des mots de passe et des utilisateurs.

Nous avons choisi d'utiliser des fichiers *.sav* pour la sauvegarde des parties. Ces fichiers sont employés pour la sérialisation des données, offrant ainsi une solution robuste pour la sauvegarde. Cette approche a été adoptée après avoir rencontré des défis lors de la sérialisation/désérialisation des données et dans la manipulation de l'interface *Serializable*. De nombreuses classes de notre code implémentent cette interface, permettant ainsi la conversion des instances de ces classes en flux d'octets, facilitant leur stockage et leur transfert.

Annotation `@SuppressWarnings("exports")`

Certaines de nos méthodes utilisent l'annotation `@SuppressWarnings("exports")`. Elle est employée pour éliminer les avertissements du compilateur liés à l'utilisation de l'annotation `exports` dans le module. Cela contribue à une gestion efficace des dépendances et à la propreté du code. Cette annotation supprime les avertissements associés à l'exportation de packages dans le module, assurant ainsi une meilleure gestion des dépendances et des relations entre les différentes parties du projet.

Patron de conception : *PlayObserver* et *Observateur*

Notre code repose sur plusieurs patrons de conception, parmi lesquels *PlayObserver*, un observateur qui permet à la vue (*MorpionSolitaireView*) de détecter les changements dans le modèle (*Grid*) et de mettre à jour l'interface graphique en conséquence. Un observateur facilite la communication entre le modèle et la vue de manière modulaire, permettant à un objet d'être informé des changements d'état d'un autre objet et de réagir en conséquence. Les patrons de conception offrent des solutions éprouvées à des problèmes communs de conception logicielle, améliorant ainsi la clarté, la maintenabilité et la réutilisabilité du code.

2. La Vue

La classe *MorpionSolitaireView*, créée en utilisant *JavaFX*, assume le rôle de la vue dans notre architecture. En observant les changements dans le modèle grâce à l'implémentation de l'interface *PlayObserver*, elle suit le modèle d'observation. Les nuances de couleurs et le style graphique sont définis dans la classe *Theme*. L'emploi du modèle d'observation (*Observer*) permet à la vue de réagir de manière modulaire et efficiente aux changements dans le modèle.

Le type *Record* pour le Thème

La classe *Theme* tire parti du nouveau type de données *Record* de Java pour simplifier la définition et la gestion des données liées au thème graphique. Le type *Record*, une fonctionnalité de Java, simplifie la création de classes immuables en combinant

automatiquement la déclaration des champs avec les méthodes *equals()*, *hashCode()*, et *toString()*. Cette approche améliore la lisibilité du code.

FXML et Scene Builder

Les fichiers *FXML* sont utilisés pour définir l'interface graphique, simplifiant le processus de création des vues avec Scene Builder. Cette approche facilite la maintenance et l'organisation du code. Nos fichiers *.fxml* sont regroupés dans le dossier de ressources. Les fichiers FXML décrivent l'interface utilisateur de manière déclarative, permettant la séparation de la structure de l'interface graphique du code Java. Scene Builder simplifie la conception de l'interface utilisateur en autorisant la création visuelle des fichiers FXML, rendant ainsi la conception plus intuitive et simplifiant la maintenance du code.

GraphicContext

La classe *GraphicContext* de JavaFX offre la possibilité de dessiner sur un canevas. Dans *MorpionSolitaireView*, elle est utilisée pour manipuler les éléments graphiques, offrant un contrôle précis sur le rendu graphique. L'usage de *GraphicContext* permet une personnalisation puissante de l'interface utilisateur du jeu Morpion Solitaire, offrant la création d'une interface visuellement attrayante et personnalisée.

3. Le Contrôleur

Les quatre contrôleurs, que l'on retrouve dans notre package controller, suivent le principe de séparation des préoccupations, et rendent notre code modulaire. C'est-à-dire que chacun de nos contrôleurs joue un rôle spécifique au sein de l'application, il gère chacun les événements réalisés sur une interface réalisée (portant le même nom). Cette manière de faire permet de rendre notre code flexible et simplifiant la maintenance au fil du développement.

Le canevas de jeu

La variable *canvaJeu*, constitue une interface graphique dynamique au sein de l'application. En tant que composant visuel, le canevas offre une surface interactive permettant le rendu et la manipulation graphique. Dans notre jeu, le canevas, associé à des événements de souris, offre aux joueurs une plateforme immersive pour jouer et interagir avec les éléments visuels du jeu.

Annotation @FXML

L'annotation *@FXML* indique que le champ de classe correspondant doit être injecté depuis un fichier FXML. Cette annotation établit un lien entre les composants graphiques définis dans nos fichiers FXML et leur représentation dans notre code. En facilitant cette intégration, l'annotation *@FXML* simplifie la gestion des éléments de l'interface et assure une synchronisation efficace entre notre visuel et la logique fonctionnelle de l'application. Elle joue un rôle clé dans l'association dynamique entre la présentation visuelle et la manipulation de ces composants dans le code source Java.

B. Les algorithmes de recherche

1. Méthode de recherche de solution automatique aléatoire

Le premier algorithme de recherche implémenté est une recherche de solution aléatoire. La classe *MorpionSolitaireModel* qui représente une partie jouée possède comme variable *lignesPossible* qui comprend l'ensemble des coups possibles -c'est à dire lignes pouvant être tracées- étant donné l'état de la grille. L'algorithme de recherche de solution automatique aléatoire sélectionne un de ces coup possible de manière aléatoire et le joue.

Dans notre interface de jeu, il est possible de jouer un seul coup aléatoire ou de jouer des coups aléatoires jusqu'à la fin de la partie. Jouer un seul coup de manière aléatoire consiste donc à choisir une ligne dans les lignes possibles, la jouer et actualiser l'état du jeu. Si on choisit de jouer des coups aléatoires jusqu'à la fin de la partie, alors l'algorithme joue coup par coup, actualisant l'état de la partie à chaque itération, calculant l'ensemble des coups possibles et en choisissant l'un d'entre eux. L'algorithme s'arrête lorsqu'il n'y a plus de coup possible et donc que la partie est finie.

2. Méthode de recherche de solutions avec l'algorithme NMCS

Le second algorithme de recherche implémenté est le Nested Monte Carlo Search. Cet algorithme consiste à effectuer de nombreuses simulations aléatoires du jeu en partant d'un état initial. Une simulation consiste à jouer un coup possible à faire de l'état initial puis de jouer les coups suivants aléatoirement, jusqu'à aboutissent à un état final caractérisé par le score obtenu. Tous les coups possibles depuis l'état initial sont explorés dans une simulation. Le coup joué est celui de la simulation ayant obtenu le score le plus élevé.

L'implémentation de l'algorithme se fait à l'aide de deux classes : une décrivant l'état d'une simulation et l'autre faisant la recherche de la meilleure simulation. L'algorithme possède un hyperparamètre : le niveau. Le niveau caractérise sur quelle profondeur tous les coups possibles sont explorés. Un niveau 2 signifie faire une simulation pour chacun des coups possibles depuis l'état initial 'i', et pour chaque simulation créer une simulation explorant chaque coup possible depuis cet état 'i+1' jusqu'à un état final. Ici nous prenons 1 comme niveau car nous n'avons pas la puissance de calcul nécessaire et fait un code asynchrone pour faire plus.

Comme pour l'algorithme de recherche de solution automatique aléatoire, il est possible de jouer au choix un seul coup optimisé par l'algorithme NMCS ou une partie entière avec l'algorithme NMCS. Les scores obtenus en faisant une partie entière sont en moyenne de 59 pour les règles 5D et de 82 pour le 5T. Les meilleurs scores obtenus ont été 61 et 86 respectivement.

C. Les tests

Des tests unitaires ont été écrits pour les classes du modèle et non de la vue et du contrôleur. Nous avons fait ce choix non seulement par souci de temps, mais aussi car les tests d'interface sont une branche à part entière des tests logicielle, pour laquelle nous manquons encore de compétence, nous avons donc préféré soigné nos tests de modèle plutôt que de nous éparpiller.

Nous avons effectué les tests dans un ordre précis : tests des classes les plus plus fondamentales en premières.

En effet, faire les tests de méthodes appelant d'autres méthodes n'ayant pas été testées ne serait pas rigoureux. L'ordre a donc été le suivant :

1. *Point.java*
2. *BlockedDirectionPoint.java*
3. *Line.java*
4. *Grid.java*
5. *Score.java*
6. *SauvegardeGrille.java*
7. *MorpionSolitaireModel.java*

Les 5 premiers tests correspondent aux composants de base du jeu, chacun appelant les classes inférieures, montant en complexité. Les tests de la classe *SauvegardeGrille* sont un peu différents car n'est pas à proprement parler un moteur du jeu. La classe *MorpionSolitaireModel* est l'agrégation de tous les éléments du jeu préalablement testés et ne peut fonctionner que si ces derniers fonctionnent. Il était donc important d'avoir réalisé les tests des autres classes au préalable.

Dans les tests de *Grid*, *Score*, *SauvegardeGrille* et *MorpionSolitaireModel* nous réalisons des coups choisis de manière arbitraire, sans forcément mener une partie à son bout car cela rendrait les tests plus longs à exécuter et surtout n'apportant pas de robustesse en plus. Dans le test de sauvegarde, nous vérifions aussi que les noms soient bien conservés.

Des tests unitaires ont également été écrits pour l'algorithme de recherche aléatoire et NMCS. Ils consistent entre autres à réaliser une simulation d'une partie entière et de voir si le score obtenu est cohérent.

III. Stack technique

A. Maven

Pour gérer nos dépendances de manière efficace nous utilisons Maven, cela nous permet aussi de simplifier notre processus de construction. Maven agit en tant que système de gestion de projet, facilitant le cycle de développement du jeu.

Le fichier *POM* (Project Object Model) que nous avons configuré spécifie des informations telles que le groupe, l'identifiant, la version du projet, ainsi que son nom et sa description. Le *POM* consolide également les détails liés au processus de construction, avec des configurations pour les plugins Maven pertinents.

Dans notre cas, Maven orchestre la compilation du code source Java à travers le plugin *maven-compiler-plugin*, assurant ainsi la compatibilité avec la version 17 du langage. De plus, il gère la documentation de notre code source grâce au plugin *maven-javadoc-plugin*, générant des fichiers javadoc et simplifiant ainsi la compréhension de notre code.

Les dépendances de notre projet, spécifiées dans la section *<dependencies>* du *POM*, sont également gérées par Maven. Ces dépendances incluent des bibliothèques telles que BCrypt, jBCrypt, Gson, JavaFX, JUnit, OpenCSV, et Jackson Databind. Maven se charge de télécharger automatiquement ces bibliothèques depuis les référentiels Maven centralisés, garantissant ainsi la cohérence et l'intégrité des versions utilisées dans notre application.

B. Stockage des données

Le tableau ci-dessous compare les outils de stockage de données utilisés pour le projet à une autre solution que nous avons envisagé au début avant l'implémentation. Cela permet de mettre en avant notre décision de stocker nos données au sein de .csv et .json plutôt que dans une base de données plus lourde.

Caractéristique	CSV	JSON	SGBD
Facilité d'utilisation	Facile à comprendre et à manipuler, mais limité pour les données complexes	Structure hiérarchique, adapté pour des données semi-structurées	Idéal pour gérer de grandes quantités de données structurées
Compatibilité	Compatible avec la plupart des systèmes et des langages	Souple, mais certains systèmes peuvent nécessiter des configurations spécifiques	Requiert un serveur de base de données, compatible avec de nombreux langages via des pilotes JDBC
Performance	Convient pour des volumes de données modestes	Convient pour des données de taille moyenne, peut être moins performant que CSV	Optimisé pour des performances élevées, surtout avec de grandes bases de données

Simplicité de mise en œuvre	Pas de configuration nécessaire, facile à implémenter	Configurations minimales, mais nécessite une certaine compréhension de la structure JSON	Configuration initiale requise, mais des outils facilitent la gestion
------------------------------------	---	--	---

C. Interface

Avant l'implémentation complète du jeu nous avons réfléchi au framework à utiliser pour l'interface graphique, pour cela nous avons donc comparé JavaFX avec Swing dont vous trouverez les arguments dans le tableau suivant. Nous avons opté pour JavaFX qui nous semblait mieux répondre à nos besoins pour l'implémentation de notre jeu.

Caractéristique	JavaFX	Swing
Design Esthétique	Permet des interfaces utilisateur modernes et esthétiques	Interface utilisateur plus traditionnelle
Fonctionnalités avancées	Offre des fonctionnalités avancées telles que les animations, les effets, les graphiques, etc.	Moins de fonctionnalités graphiques avancées
Intégration CSS	Supporte l'intégration CSS pour le stylage de l'interface utilisateur	N'a pas de support CSS intégré
Scène graphique	Utilise une scène graphique pour la gestion des éléments d'interface	Utilise des conteneurs et composants pour construire l'interface
Intégration avec FXML	Intègre le langage FXML pour la conception déclarative des interfaces utilisateur	Utilise des classes Java pour construire l'interface de manière impérative
Interopérabilité native	Meilleure interopérabilité avec des technologies plus récentes telles que Java 11+	Ancienne technologie, moins de support pour les nouveautés Java

Conclusion

La réalisation du projet que nous venons de décrire a été à la fois complexe et enrichissante. L'importance du détail a été à la fois pour le moteur du jeu mais aussi pour son interface nous a vraiment poussé à comprendre le domaine du développement logiciel en java. Tout au long de l'implémentation, nous avons fait face à des défis variés, mais avons également réussi à surmonter ces obstacles pour aboutir à une application fonctionnelle et bien structurée.

L'élaboration du diagramme de classes a été un aspect crucial du processus de conception. Il s'est constamment affiné au fil du temps pour refléter au mieux la structure logique du jeu. Cela a nécessité une analyse minutieuse des interactions entre les différentes composantes du système, garantissant ainsi une architecture robuste et flexible.

Parmi les défis majeurs rencontrés, l'implémentation de l'algorithme NMCS a été particulièrement exigeante. Son optimisation a demandé énormément de temps et de recherche annexes, ce qui nous a permis de mettre en place un algorithme robuste et à la fin de réellement comprendre son fonctionnement.

Les tests et la gestion des fichiers de sauvegarde ont également posé des difficultés substantielles. La nécessité d'assurer la fiabilité des données et la cohérence du jeu a exigé une attention particulière dans la conception des cas de test, ainsi qu'une validation minutieuse des mécanismes de sauvegarde et de chargement.

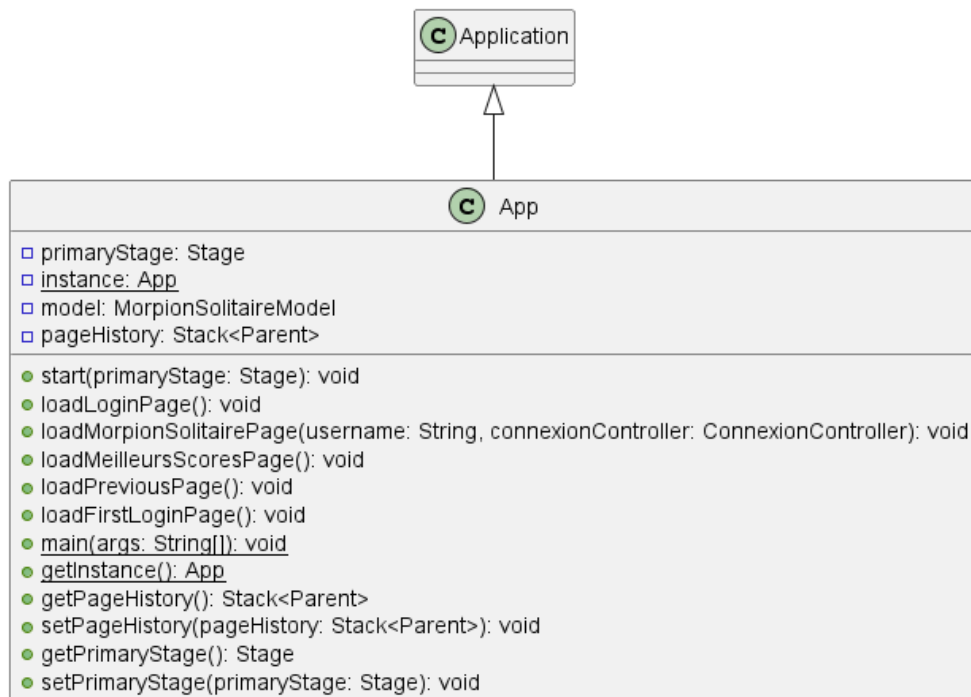
Bien que le projet ait atteint ses objectifs initiaux, nous reconnaissons qu'il reste des possibilités d'amélioration. Des itérations futures pourraient se concentrer sur l'optimisation des performances, l'ajout de fonctionnalités supplémentaires et l'amélioration de l'expérience utilisateur.

Finalement, le code source complet du projet Morpion Solitaire est disponible sur notre dépôt GitHub : <https://github.com/Jajouuuuu/Morpion>.


Annexes

Diagrammes de classes par packages

1. app




2. controler

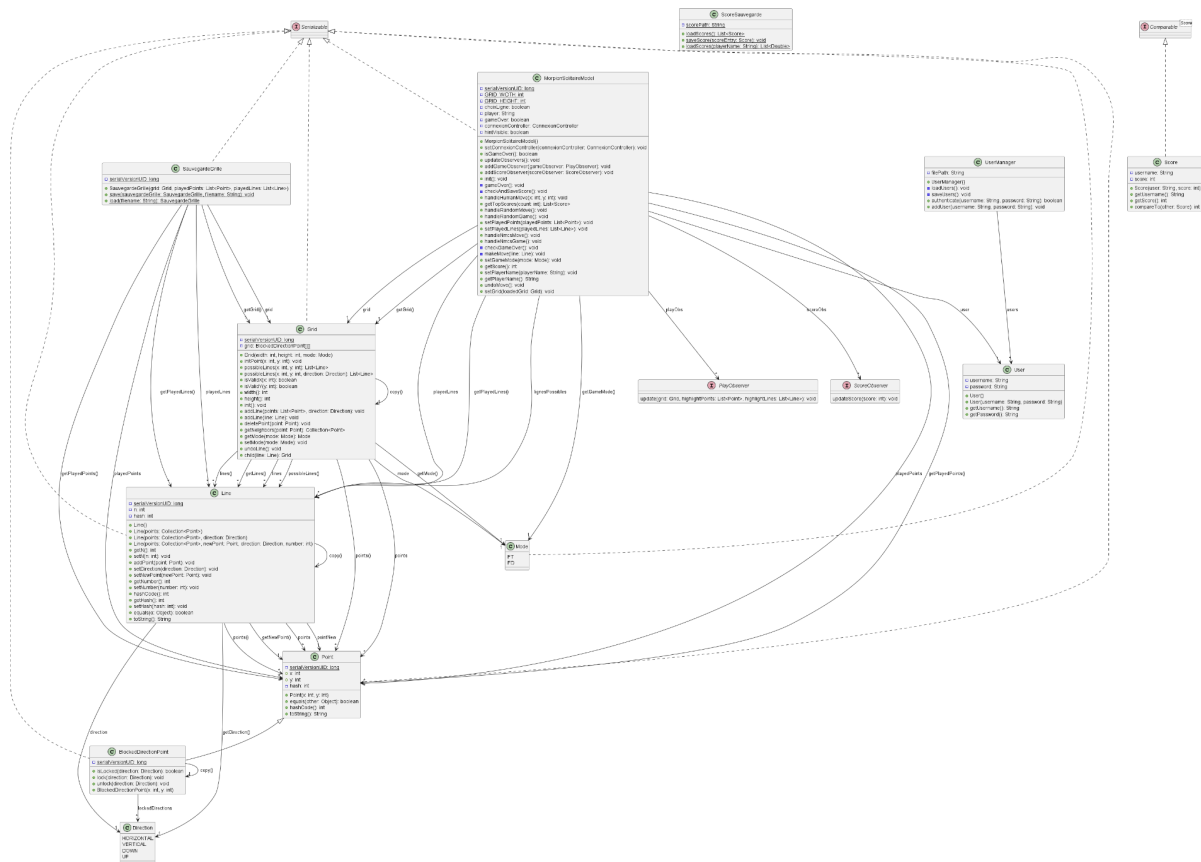
 MeilleursScoresController
<ul style="list-style-type: none"> □ scoresTable: TableView<Score> □ usernameColumn: TableColumn<Score,String> □ scoreColumn: TableColumn<Score,Integer> □ model: MorpionSolitaireModel
<ul style="list-style-type: none"> ● initialize(): void ■ handleRetourButton(): void ● setModelAndInitialize(model: MorpionSolitaireModel): void

 MorpionSolitaireController
<ul style="list-style-type: none"> □ canvaJeu: Canvas □ view: MorpionSolitaireView □ model: MorpionSolitaireModel □ app: App □ user: TextField □ mode: ComboBox<String> □ isInitialization: boolean
<ul style="list-style-type: none"> ■ initialize(): void ● setApp(app: App): void ■ canvasMousePressed(me: MouseEvent): void ● start(): void ■ setupOptions(): void ■ reset(): void ■ undoMove(): void ■ gameModeChanged(): void ■ deconnexion(): void ■ sauvegarderPartie(): void ■ chargerPartie(): void ■ mesMeilleursScores(): void ■ consulterInfos(): void ■ changerInfos(): void ● setModel(model: MorpionSolitaireModel): void ● setModelAndView(model: MorpionSolitaireModel, view: MorpionSolitaireView): void ■ checkGameOver(): void ● getCanvas(): Canvas ■ handleNmcsCoupButton(): void ■ handleNmcsPartieButton(): void ■ handleRandomCoupButton(): void ■ handleRandomPartieButton(): void

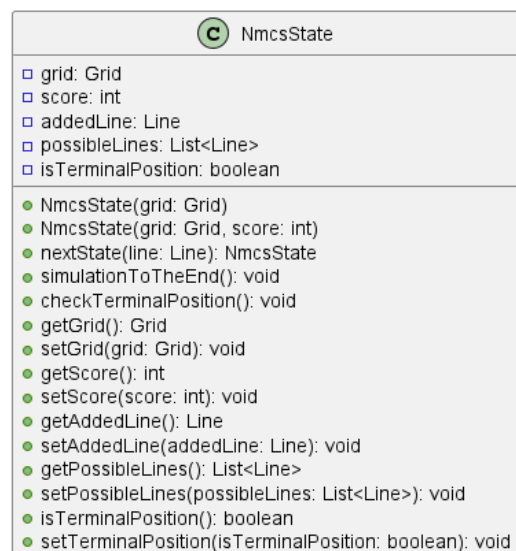
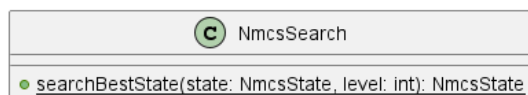
 ConnexionController
<ul style="list-style-type: none"> □ usernameField: TextField □ passwordField: PasswordField □ userManager: UserManager □ mainApp: App □ currentUsername: String □ loginButton: Button □ onLoginSuccess: Runnable □ onFirstLoginRequest: Runnable
<ul style="list-style-type: none"> ● setOnLoginSuccess(onLoginSuccess: Runnable): void ● setOnFirstLoginRequest(onFirstLoginRequest: Runnable): void ● getCurrentUsername(): String ■ handleLoginButtonAction(event: ActionEvent): void ■ handleFirstLoginButtonAction(event: ActionEvent): void ■ authenticateUser(username: String, password: String): boolean ■ handleSuccessfulLogin(): void ■ handleFailedLogin(): void ■ loginButtonAction(event: ActionEvent): void ■ firstLoginButtonAction(): void ● setMainApp(mainApp: App): void ■ showError(message: String): void

 FirstLoginController
<ul style="list-style-type: none"> □ newUsernameField: TextField □ newPasswordField: PasswordField □ userManager: UserManager □ mainApp: App
<ul style="list-style-type: none"> ● setMainApp(mainApp: App): void ■ createUserButtonAction(): void ■ showError(message: String): void ■ showInfo(message: String): void

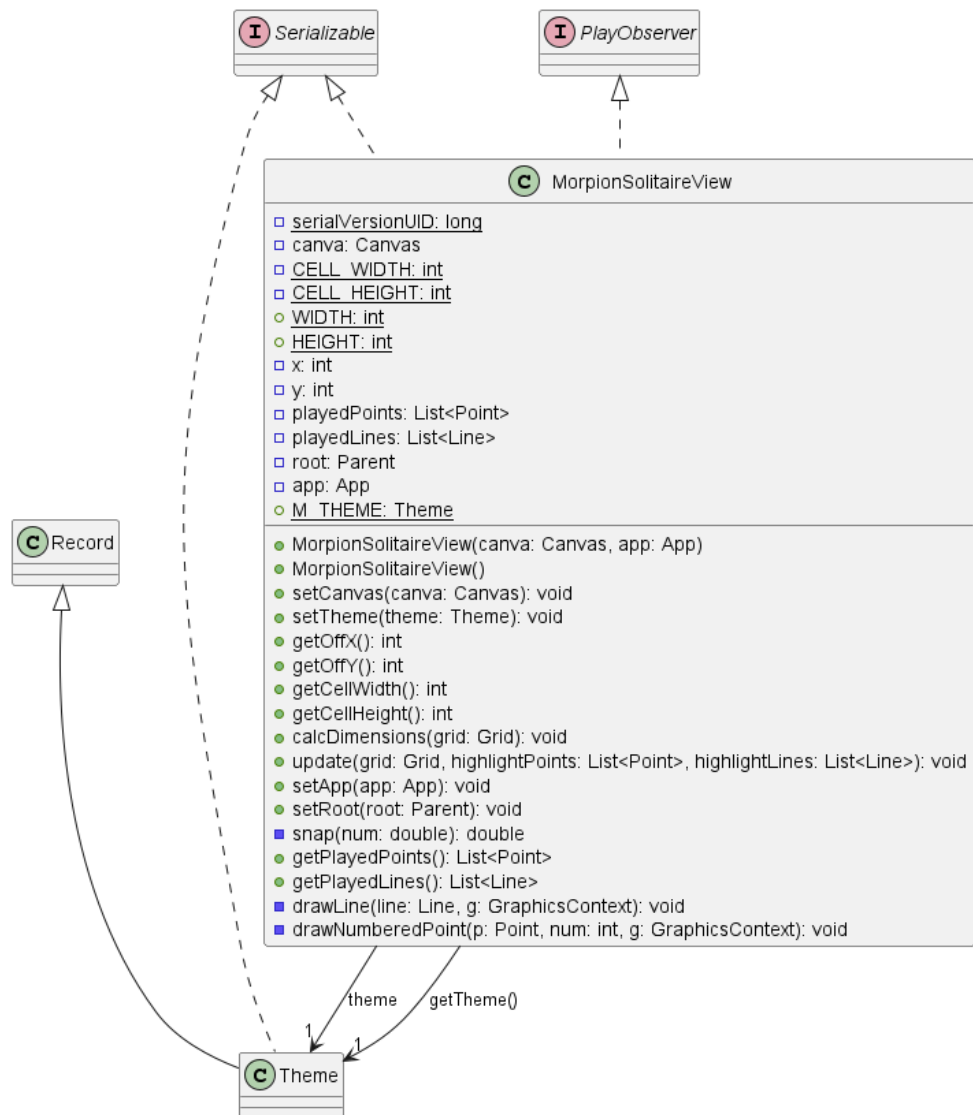
3. model



4. nmcs



5. view



6. modelTests

GridTest
<ul style="list-style-type: none"> grid: Grid setUp(): void testInit(): void testPossibleLines(): void testAddLine(): void testCopy(): void testUndoLine(): void testChild(): void

BlockedDirectionPointTest
<ul style="list-style-type: none"> testIsLocked(): void testLockAndUnlock(): void testCopy(): void

PointTest
<ul style="list-style-type: none"> testPointInitialization(): void testPointToString(): void testPointEquality(): void testPointHashCode(): void

LineTest
<ul style="list-style-type: none"> testLineInitialization(): void testLineWithPointsAndDirection(): void testLineWithPointsNewPointDirectionAndNumber(): void testAddPoint(): void testToString(): void testSetDirection(): void testCopy(): void testEqualsAndHashCode(): void

ScoreTest
<ul style="list-style-type: none"> testScoreInitialization(): void

SauvegardeGrilleTest
<ul style="list-style-type: none"> testGrid: Grid testPoints: List<Point> testLines: List<Line> testSauvegardeGrille: SauvegardeGrille testFilename: String setUp(): void testSaveAndLoad(): void

MorpionSolitaireModelTest
<ul style="list-style-type: none"> morpionSolitaireModel: MorpionSolitaireModel setUp(): void testInitialization(): void testInit(): void testHandleHumanMove(): void testHandleRandomMove(): void testHandleRandomGame(): void testHandleNmcsMove(): void testHandleNmcsGame(): void testUndoMove(): void

7. NMCSTests

NmcsSearchTest
<ul style="list-style-type: none"> testSearchBestState(): void

NmcsStateTest
<ul style="list-style-type: none"> testInitialState(): void testCustomScore(): void testNextState(): void testSimulationToTheEnd(): void