



Le Mans Université
Licence Informatique
3ème année
Génie Logiciel 2 - Gr4
Cahier d'Analyse

M3ACNL

HashiParmentier

COGNARD Luka; GAUMONT Maël; LELANDAIS Clément;
MABIRE Aymeric; PESANTEZ Maëlig; PUREN Mewen;
TOUSSI Nassim;

27 mars 2025

Table des matières

I	Introduction.....	2
II	Choix Techniques.....	2
III	Integration.....	3
III.1	Conventions *****	3
III.2	GitHub Actions *****	3
III.3	Javadoc *****	4
III.4	Tests *****	5
IV	Organisation.....	6
IV.1	Répartition des tâches *****	6
IV.2	Diagramme de Gantt *****	7
V	Conception	8
V.1	Diagrammes de classes *****	8

I - Introduction

Ce projet vise à créer un jeu de Hashiwokakero interactif et intuitif, garantissant une expérience utilisateur fluide et agréable. Pour y parvenir, nous avons adopté des choix techniques pertinents, conçu une architecture logicielle optimisée, réparti efficacement les tâches au sein de l'équipe et élaboré une série de tests complète.

II - Choix Techniques

Afin de développer efficacement le jeu de Hashiwokakero, nous avons pris des décisions techniques adaptées aux exigences du projet, garantissant ainsi une implémentation robuste et performante.

Tout d'abord, nous avons opté pour GitHub comme plateforme de gestion de code et versionning, facilitant la collaboration, le suivi des modifications et la gestion des versions. Nous utilisons également des fonctionnalités telles que GitHub Actions, qui automatise le processus de production.

Pour assurer une uniformité du code et de la compilation sur les différents environnements de travail et systèmes d'exploitation, nous avons choisi Maven. Cet outil de gestion de projet et de compilation simplifie la gestion des dépendances, l'assemblage du code et la génération des artefacts, garantissant ainsi un développement plus fluide et une maintenance facilitée.

Afin de documenter clairement notre code source, nous avons adopté Javadoc, qui génère automatiquement une documentation à partir des commentaires Java. Cela améliore la compréhension et l'utilisation des classes et méthodes par l'équipe de développement.

L'intégration de Jupiter de JUnit et Mockito permet l'écriture et l'exécution de tests unitaires, assurant ainsi la fiabilité du code en détectant et en réduisant les erreurs potentielles.

Pour la manipulation des données au format JSON, nous avons opté pour la bibliothèque Jackson, qui permet la sérialisation et la désérialisation des objets Java en JSON, facilitant ainsi la gestion des sauvegardes et des ressources de l'application.

Enfin, nous avons utilisé Maven Shade pour la génération de l'application. Ce plugin permet d'intégrer les dépendances nécessaires directement dans le fichier JAR final, garantissant ainsi que les utilisateurs puissent exécuter le jeu sans avoir à installer manuellement des bibliothèques comme JavaFX, aussi connu sous le nom de FatJar.

III - Integration

III.1) Conventions

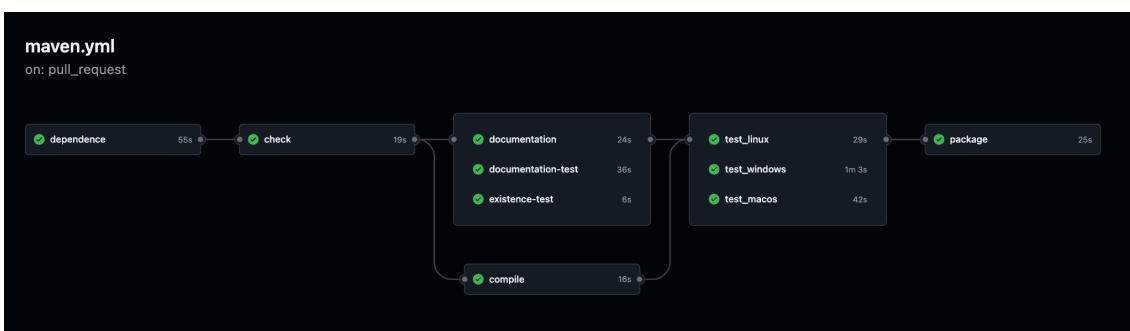
Afin de garantir une intégration fluide et une collaboration efficace entre les membres de l'équipe, nous avons défini des conventions de codage et de documentation à respecter tout au long du développement. Ces règles sont spécifiées dans un fichier `checkstyle.xml`, qui permet d'en vérifier l'application lors de la compilation du code. Cette validation est effectuée automatiquement par Maven à l'aide du plugin Checkstyle, et toute non-conformité entraîne un blocage de la compilation. Bien que certains membres de l'équipe aient rencontré des difficultés à s'adapter à ces exigences, l'usage de Checkstyle leur a permis de mieux appréhender l'importance de la qualité du code et de la documentation, tout en améliorant progressivement leurs pratiques.

Concernant l'utilisation de GitHub, nous avons mis en place des règles précises pour la gestion des branches, des commits et des pull requests. Le dépôt repose sur deux branches principales : `master` et `develop`. La branche `master` contient la version stable de l'application et correspond à la version de production, tandis que la branche `develop` regroupe les développements en cours, qui doivent rester aussi stables que possible. Toute nouvelle branche doit être créée à partir de `develop`. Chaque membre de l'équipe est tenu de créer une branche spécifique pour chaque nouvelle fonctionnalité et d'effectuer un commit à chaque modification significative.

Les messages de commit doivent être clairs et explicites, décrivant précisément les modifications apportées. Toute nouvelle branche doit faire l'objet d'une pull request, qui doit être revue et approuvée par un autre membre de l'équipe avant d'être fusionnée dans la branche principale. Ces bonnes pratiques ont permis de garantir une intégration continue et une collaboration efficace tout au long du projet.

III.2) GitHub Actions

Afin de garantir le bon fonctionnement du code présent sur la branche `develop` et, par extension, sur la branche `master`, il est essentiel de s'assurer que chaque nouvelle branche repose sur une base stable. Pour ce faire, nous avons décidé d'automatiser les différentes validations grâce à GitHub Actions, comme illustré dans l'exemple *ci-dessous*.



Comme on peut l'observer dans l'image *ci-dessus*, la validation s'effectue lors d'une `pull request` et se déroule en plusieurs étapes :

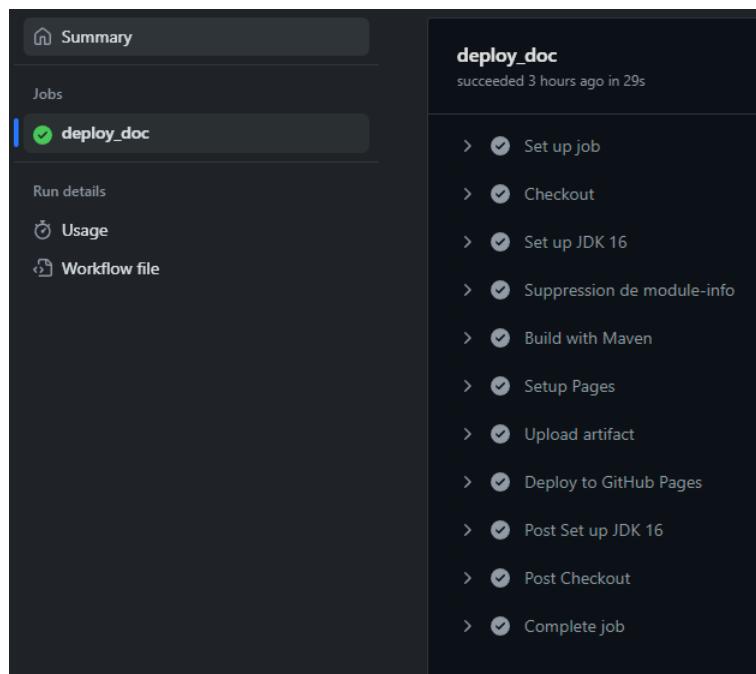
- **Dependence** : Télécharge toutes les dépendances Maven nécessaires aux étapes suivantes.
- **Check** : Vérifie la conformité du code aux règles définies par Checkstyle. Cette étape est indispensable pour toutes les étapes suivantes.
- **Compile** : Dépend de l'étape **Check** et s'assure que le code peut être compilé sans erreur.

- **Documentation** : Dépend de **Compile**. Vérifie la possibilité de générer la *Javadoc* pour les classes et bloque la compilation si des classes ou méthodes ne possèdent pas de commentaires correctement formatés.
- **Documentation-Test** : Dépend de **Compile**. Vérifie la génération de la *Javadoc* pour les classes de test et bloque en cas d'absence de commentaires formatés.
- **Test-Linux** : Dépend de **Compile**. Exécute l'ensemble des tests sous un environnement **Linux**. L'échec d'un seul test bloque la validation.
- **Test-Windows** : Dépend de **Compile**. Exécute l'ensemble des tests sous un environnement **Windows**. Tout test échoué entraîne un blocage.
- **Test-Macos** : Dépend de **Compile**. Exécute l'ensemble des tests sous un environnement **macOS**. Un seul échec empêche la validation.
- **Package** : Dépend de toutes les validations précédentes. Vérifie la capacité du code à générer un **JAR**, garantissant ainsi que le projet est livrable.

Si l'une de ces étapes échoue, la *pull request* est bloquée jusqu'à la correction du problème.

La gestion d'une GitHub Action se fait via un fichier **YML**, dans lequel les différentes tâches sont configurées sous forme de **jobs**, chacun contenant les étapes nécessaires à son exécution.

Nous avons également utilisé GitHub Actions pour automatiser la génération et la publication de la *Javadoc* lors d'un *push* sur la branche master.



III.3) Javadoc

La documentation *Javadoc* de notre projet est disponible en ligne sur GitHub. Afin de garantir son exhaustivité, nous avons mis en place des règles de compilation via Maven, rendant obligatoire l'ajout de commentaires au format *Javadoc* pour chaque classe, méthode et variable, qu'elle soit de classe ou d'instance.

Cette exigence, appliquée dès le début du projet, assure une documentation continue du code et présente plusieurs avantages :

- **Lisibilité et compréhension** : Les commentaires Javadoc expliquent le fonctionnement et l'utilisation du code, facilitant ainsi sa compréhension.
- **Amélioration du travail en équipe** : Une documentation claire favorise la communication entre les développeurs et simplifie la collaboration.
- **Maintenance et évolutivité** : Un code bien documenté est plus simple à maintenir et à faire évoluer au fil du temps.

- **Réutilisation du code** : Une documentation explicite encourage la réutilisation des éléments du projet en fournissant des indications précises sur leur usage.

L'intégration de la Javadoc dans notre flux de développement contribue ainsi à renforcer la qualité, la transparence et la fiabilité de notre projet.

III.4) Tests

Pour pouvoir s'assurer du bon fonctionnement des classes programmées, on a pensé judicieux d'implémenter des tests unitaires, pour cela nous avons utilisé Jupiter de JUnit et Mockito. Pour pouvoir mieux s'y retrouver dans les résultats des tests, on a modifié l'affichage en console comme ci-dessous.

```
===== DÉBUT SauvegardeManagerTest =====
[testInitialiseRépertoire()] OK
[testSaveCheminRépertoire()] OK
[testGetInstance()] OK
===== FIN SauvegardeManagerTest =====

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.094 s -- in fr.m3acnl.managers.SauvegardeManagerTest
[INFO] Running fr.m3acnl.managers.ProfileManagerTest

===== DÉBUT ProfileManagerTest =====
[testSauvegarder()] OK
[testGetProfileActif()] OK
[testDesactiverProfileActif()] OK
[testSupprimerProfileByName()] OK
[testSupprimerProfileByObject()] OK
[testListeProfils()] OK
[testCreerProfil()] OK
[testSetProfileActif()] OK
[testGetInstance()] OK
===== FIN ProfileManagerTest =====

[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.574 s -- in fr.m3acnl.managers.ProfileManagerTest
[INFO] Running fr.m3acnl.managers.OsmManagerTest
```

La valeur de retour des tests peut prendre 4 valeurs différentes avec les couleurs utilisées dans le terminal :

- **OK**
- **KO**
- **DÉSACTIVÉ**
- **AVORTÉ**

Pour pouvoir réaliser cet affichage, on a créé une classe mère de tout les tests qui est **Tests**. Cette classe étend **VerifierTest** qui exécute des fonctions selon le résultat du test, ce qui permet leur affichage.

Durant le développement, on est arrivé sur des cas où les tests unitaires étaient rédigés mais la classe testée n'était pas fonctionnelle, pour cela, on peut choisir de désactiver un test pour que lors de la compilation, il soit ignoré et donc ne bloque pas le processus.

Lors de la compilation, maven exécute les tests unitaires, et si par malheur un test échoue, la compilation est bloquée. Le fait d'obliger à un code fonctionnel à la compilation permet de réduire considérablement le nombre d'erreurs dans l'exécutable, ainsi que dans les branches du projet pour que les personnes reprenant le code n'aient pas à corriger les erreurs, et puissent utiliser le code de manière sereine.

IV - Organisation

IV.1) Répartition des tâches

Afin d'assurer le bon déroulement du développement du jeu, les tâches ont été distribuées parmi les membres de l'équipe en fonction de leurs compétences et préférences. Cette répartition a été supervisée par le chef de projet, qui a attribué les responsabilités de manière stratégique. Grâce à l'outil GitHub Project, nous avons pu suivre l'avancement de chaque membre, particulièrement lors du travail à distance. Les tâches ont été organisées en modules, chacun correspondant à une phase du projet. Chaque membre a pris en charge un ou plusieurs modules, permettant ainsi de travailler en parallèle sur différents aspects du jeu et d'accélérer son développement. La répartition des tâches a été équilibrée et équitable, assurant une contribution significative de chaque membre à la réalisation du projet.

PUREN Mewen : Chef de projet, développement du système de sauvegarde, gestion des utilisateurs, définition des conventions de codage et configuration de GitHub, répartition des tâches au sein de l'équipe, paramétrage de la compilation Maven.

PESANTEZ Maelig : Documentaliste, mise en place de la JavaDoc web, gestion des activités sur GitHub, création des assets graphiques, développement de l'affichage JavaFX (visualisation du jeu, tutoriel), élaboration des livrables (cahier des charges, cahier d'analyse).

COGNARD Luka : Développement de la logique du jeu, assistance pour les aides contextuelles, supervision des actions sur GitHub.

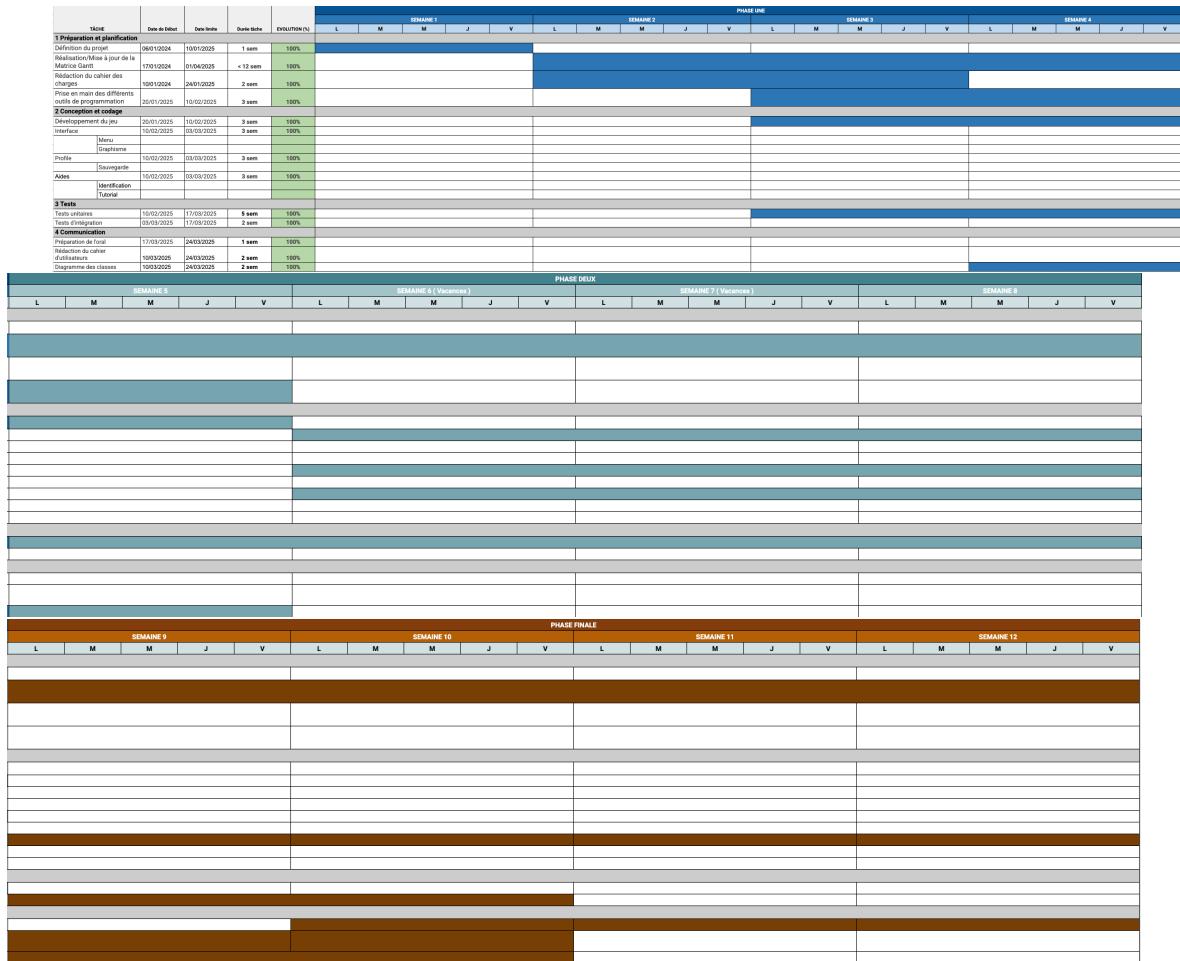
MABIRE Aymeric : Développement de la logique du jeu, développement de l'affichage JavaFX (affichage du jeu, gestion des assets, liaison des menus), rédaction du manuel utilisateur, création du diagramme de classes.

TOUSSI Nassim : Développement des menus, création du diagramme de GANTT.

LELANDAIS Clément : Développement des aides, relecture du cahier des charges.

GAUMONT Mael : Développement des aides.

IV.2) Diagramme de Gantt



V - Conception

V.1) Diagrammes de classes

Les diagrammes de classes suivants sont disponibles dans le dépôt des Livrables pour une meilleure qualité.

Diagramme de classe du package racine du projet.

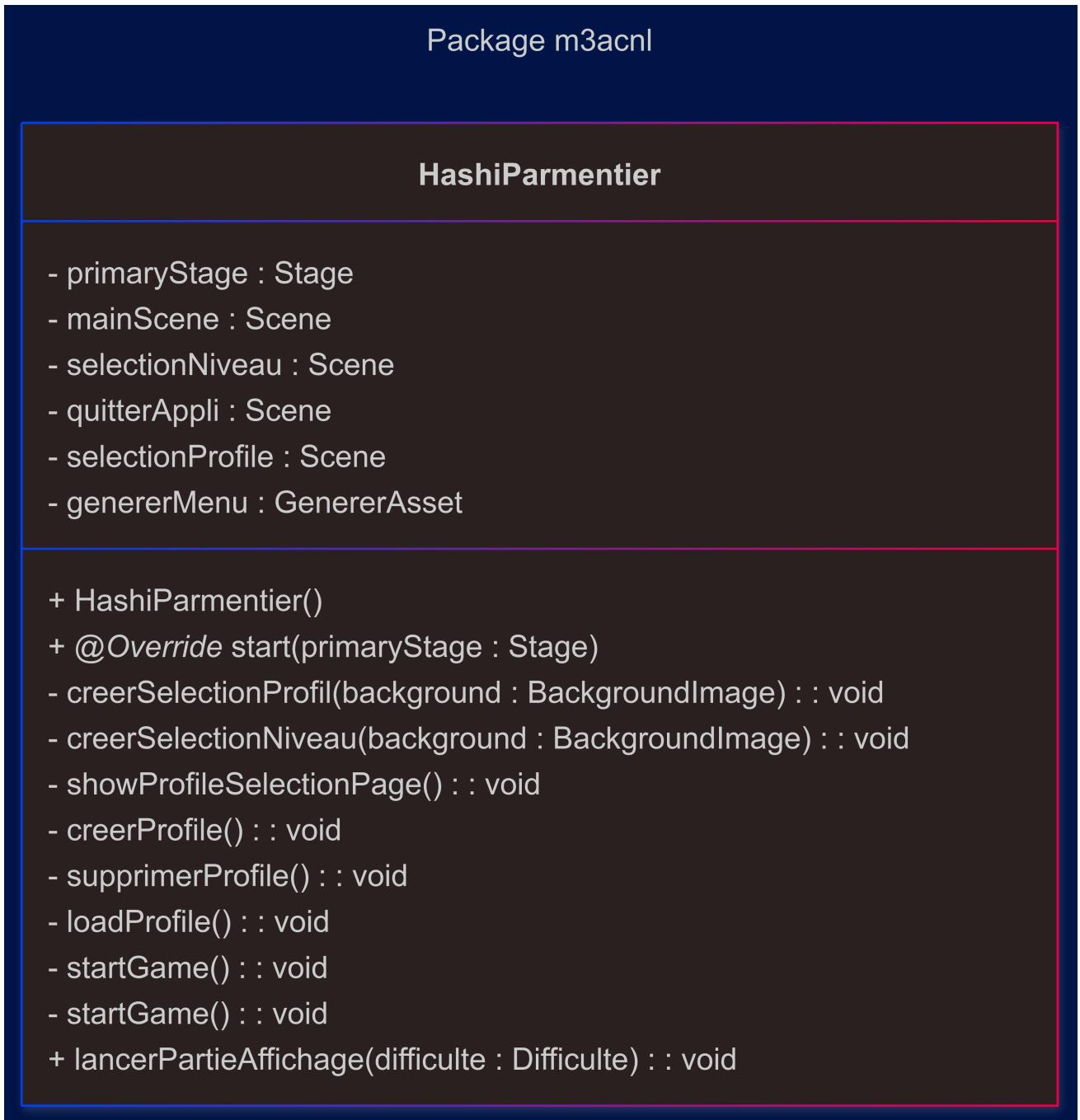


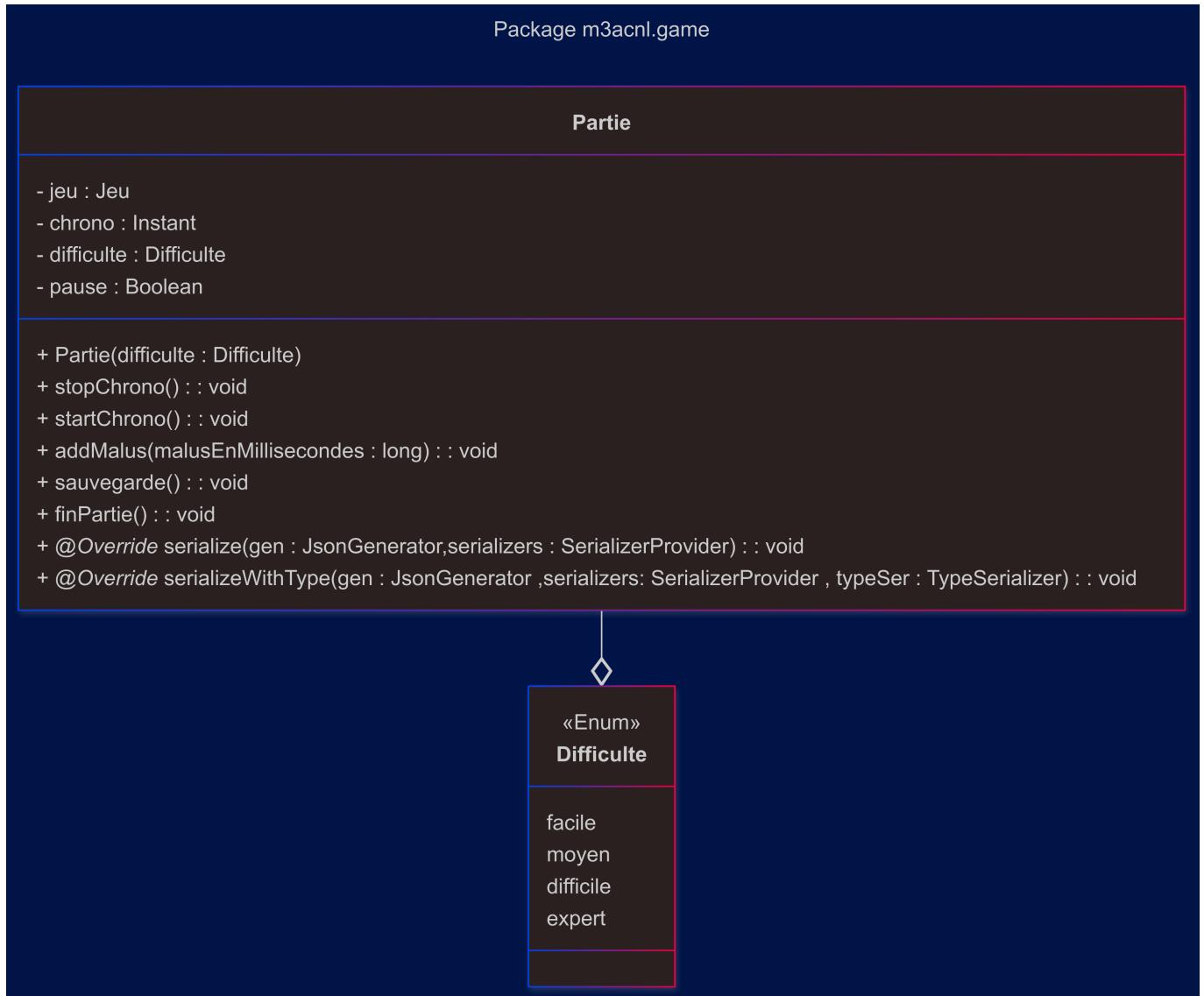
Diagramme de classe du package du jeu.

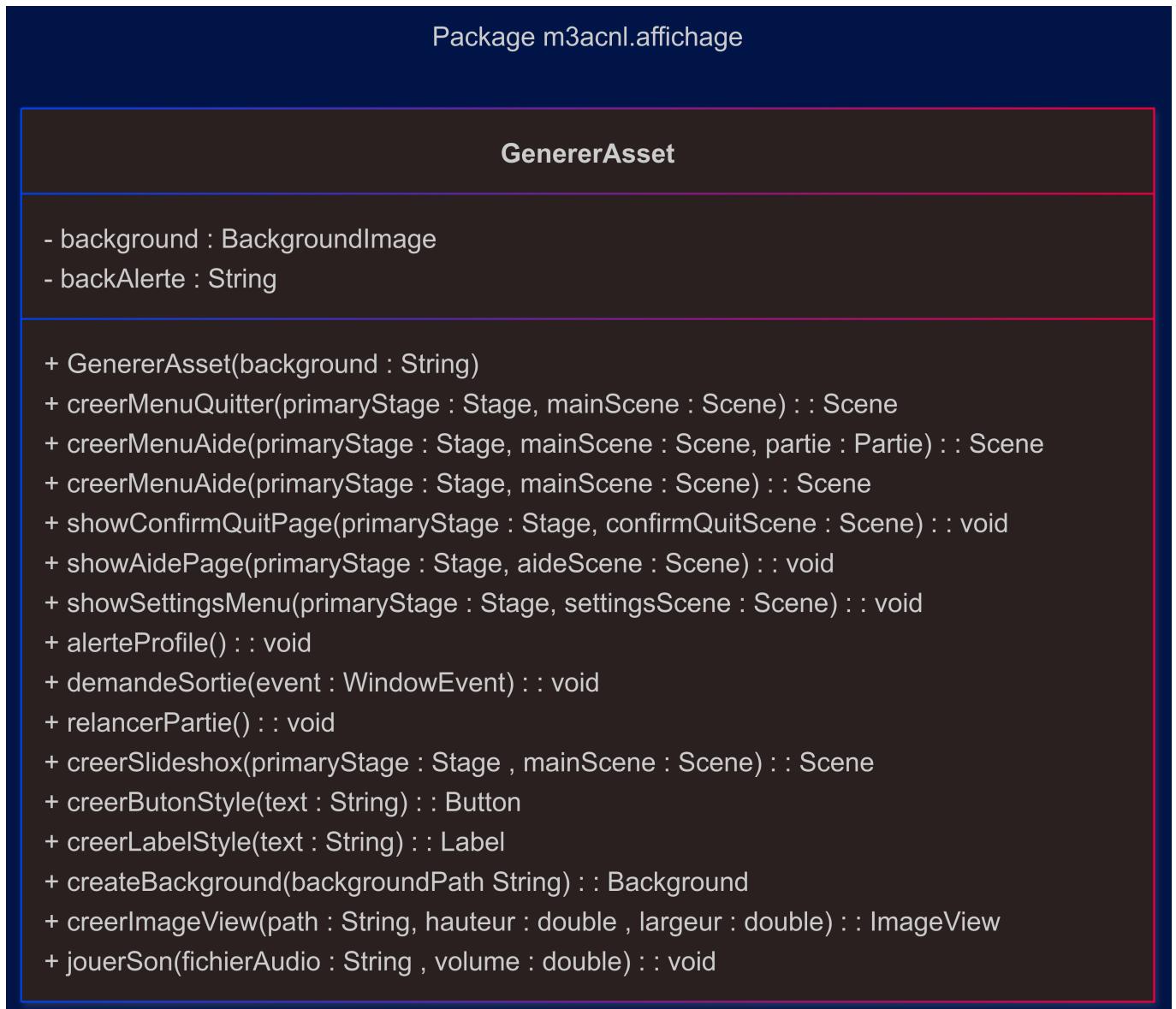
Diagramme de classe de la partie graphique.

Diagramme de classe de la partie graphique et partie logique.

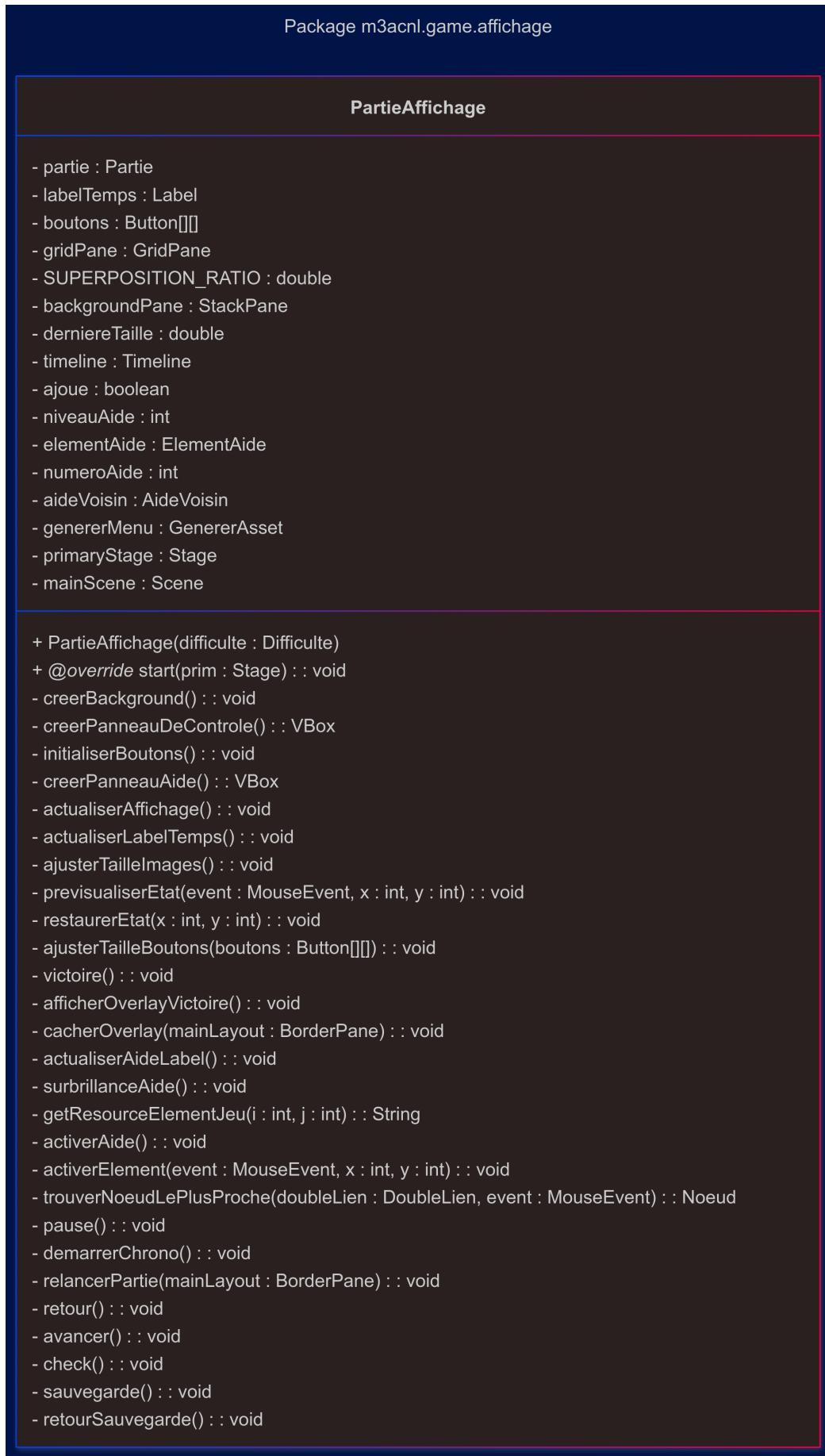


Diagramme de classe des managers.

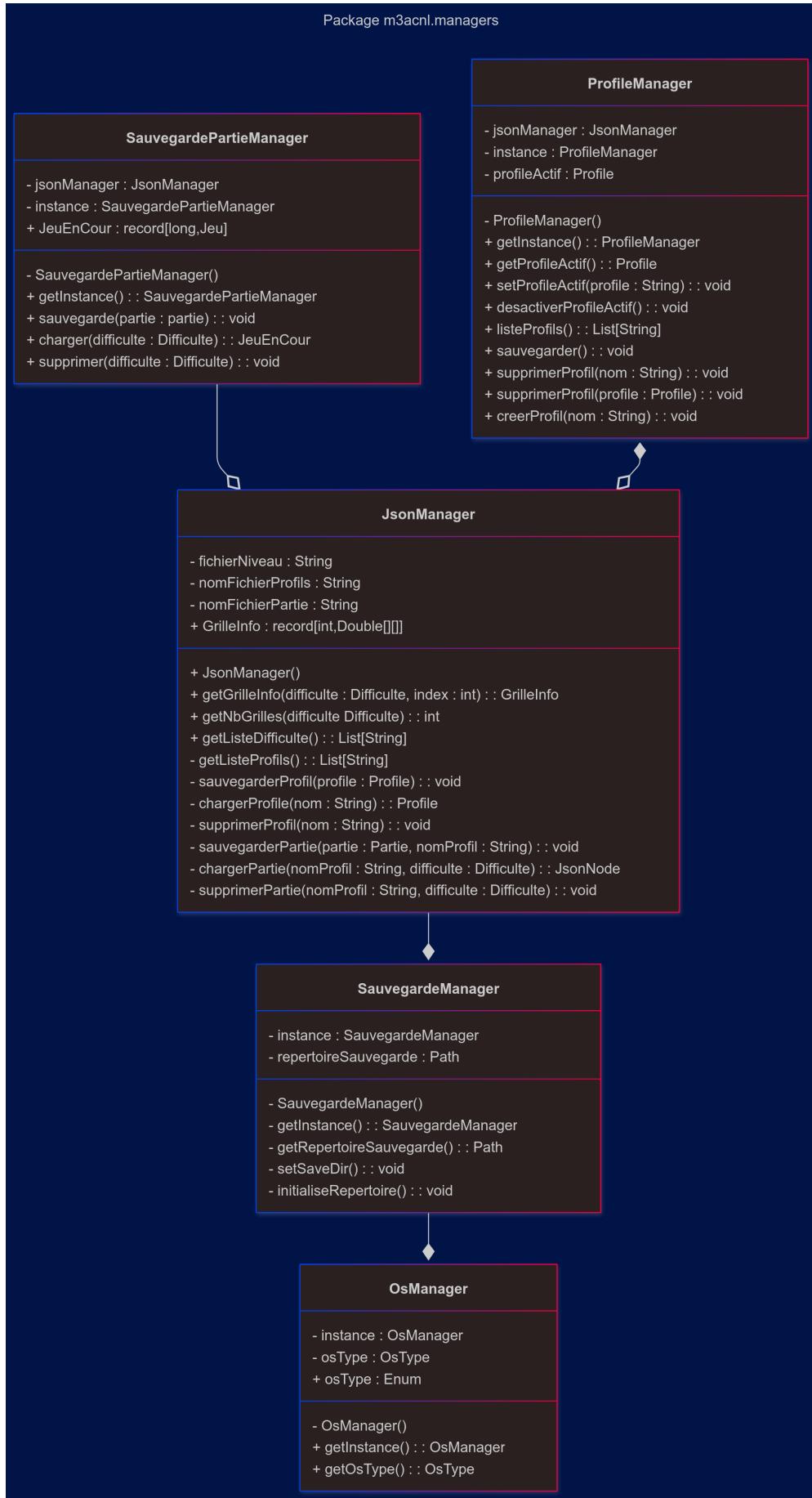


Diagramme de classe des profils.

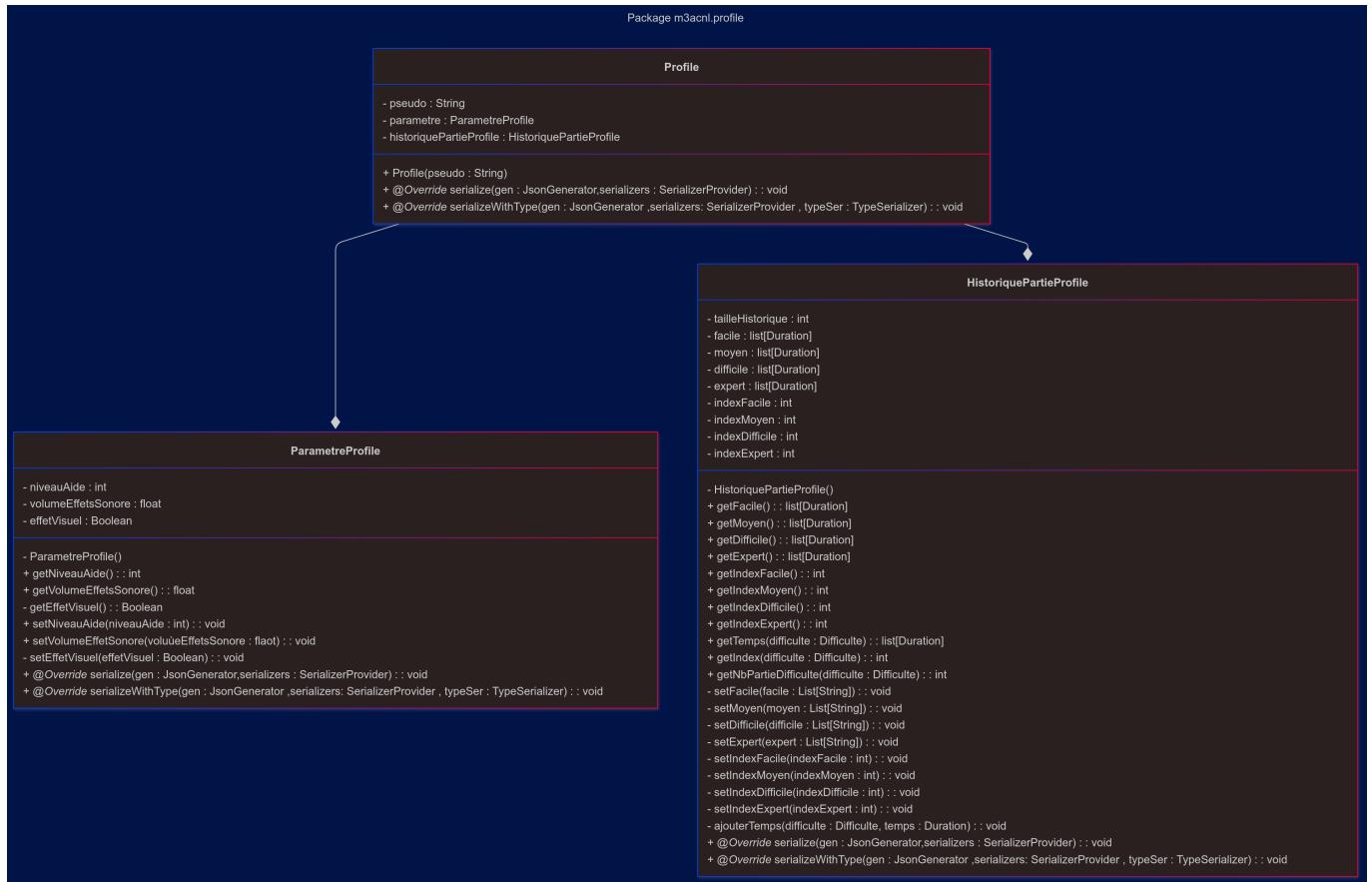


Diagramme de classe de la logique du jeu.

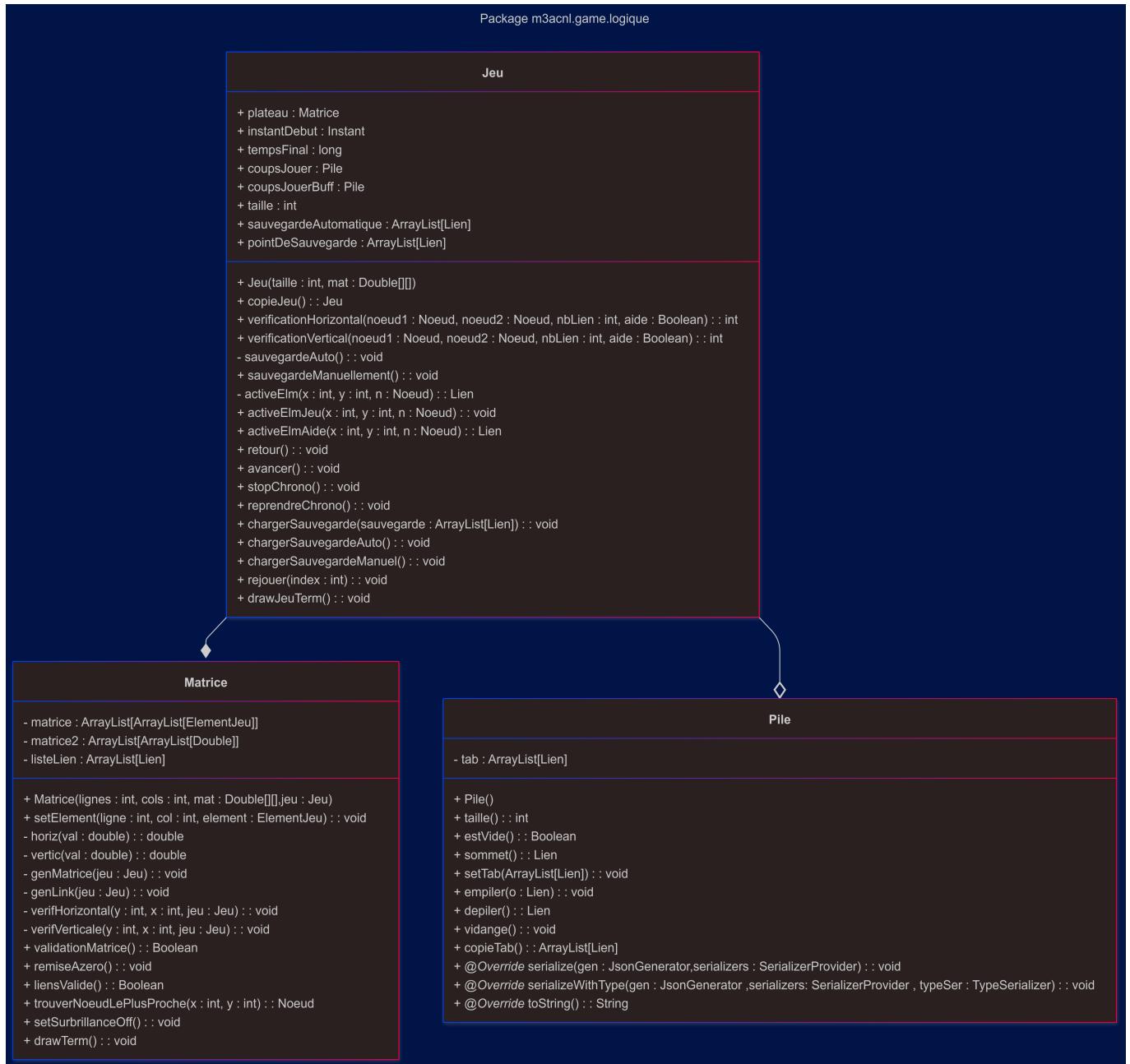


Diagramme de l'interface d'elementJeu.

