



Rapport de Projet GL

Jeu UNO en Java

Démonstration de l'architecture MVC et des patrons de conception

kacimi abderraouf	G4	232331488814
Mamache Aimen	G1	232331432510
Meziane Mohamed	G3	232331492102
Benzaoui Youcef	G4	232331413609

1. Présentation de l'application

1.1 Vue d'ensemble

Cette application est une implémentation complète du jeu de cartes UNO en Java. Elle démontre l'utilisation de l'architecture Modèle-Vue-Contrôleur (MVC) ainsi que quatre patrons de conception fondamentaux : Singleton, Strategy, Composite et Observer.

Conception de l'interface graphique (GUI)

Pour l'interface utilisateur, nous avons adopté une approche de "Design in Code" (Conception par le code). Au lieu d'utiliser des outils de prototypage externes, nous avons construit l'interface directement en Java Swing, ce qui nous a permis de:

- **Contrôle Précis** : Gérer le positionnement des éléments dynamiquement via les LayoutManagers (BorderLayout, BoxLayout) pour s'adapter à la fenêtre.
- **Architecture MVC** : Lier directement les composants visuels (Boutons, Panels) aux données du modèle via le pattern Observer.
- **Style Visuel** : Définir une charte graphique sombre et moderne ("Dark Mode") directement dans le code pour assurer une cohérence parfaite sur tous les écrans.

Technologies utilisées

- **Langage** : Java 11+\n
- **Interface graphique** : Swing (bibliothèque Java standard)\n
- **Interface console** : Entrée/sortie standard Java\n
- **Persistance** : Fichiers texte\n

1.2 Fonctionnalités principales

- • Support de 2 à 4 joueurs (humains et IA)
- • Deux modes d'interface : console (texte) et GUI (graphique)
- • Implémentation complète des règles UNO
- • Toutes les cartes spéciales : Skip, Reverse, Draw Two, Wild, Wild Draw Four
- • Sauvegarde et chargement de parties
- • Architecture MVC claire et maintenable
- • Quatre patrons de conception intégrés

2. Spécification des besoins fonctionnels et non fonctionnels

2.1 Besoins fonctionnels

Initialisation du jeu : Le système doit permettre de créer une partie avec 2 à 4 joueurs, en spécifiant le nombre de joueurs humains et IA.

Distribution des cartes : Le système doit distribuer 7 cartes à chaque joueur au début de la partie et placer une carte initiale sur le tas de défausse.

Gestion des tours : Le système doit gérer l'alternance des tours entre les joueurs, dans le sens horaire ou antihoraire selon l'état du jeu.

Validation des coups : Le système doit vérifier qu'une carte peut être jouée (correspondance de couleur, de numéro ou de type).

Effets des cartes spéciales : Le système doit appliquer les effets des cartes spéciales :
- Skip : passer le tour du joueur suivant
- Reverse : inverser le sens de jeu
- Draw Two : forcer le joueur suivant à piocher 2 cartes
- Wild : permettre de choisir une nouvelle couleur
- Wild Draw Four : forcer le joueur suivant à piocher 4 cartes et choisir une couleur

Pioche de cartes : Le système doit permettre à un joueur de piocher une carte s'il ne peut pas jouer.

Détection de victoire : Le système doit détecter quand un joueur n'a plus de cartes et le déclarer vainqueur.

Intelligence artificielle : Le système doit fournir une IA capable de jouer automatiquement pour les joueurs non-humains.

Interface double : Le système doit offrir deux interfaces utilisateur : console (texte) et graphique (Swing).

Sauvegarde/Chargement : Le système doit permettre de sauvegarder l'état d'une partie et de la charger ultérieurement.

2.2 Besoins non fonctionnels

Performance : Le jeu doit répondre instantanément aux actions des joueurs sans latence perceptible.

Utilisabilité : Les interfaces (console et GUI) doivent être intuitives et faciles à utiliser. Les messages d'erreur doivent être clairs.

Maintenabilité : Le code doit être bien structuré avec des commentaires JavaDoc, facilitant la maintenance et l'évolution.

Extensibilité : L'architecture doit permettre d'ajouter facilement de nouveaux types de cartes ou de nouvelles règles.

Portabilité : L'application doit fonctionner sur toute plateforme supportant Java 11 ou supérieur (Windows, macOS, Linux).

Fiabilité : Le jeu ne doit pas planter et doit gérer correctement les erreurs (entrées invalides, fichiers manquants, etc.).

Modularité : Le code doit être organisé en packages logiques suivant le modèle MVC.

3. Diagramme de classes illustrant l'utilisation des patrons de conception et du modèle MVC

3.1 Architecture MVC

L'application suit strictement le modèle Modèle-Vue-Contrôleur (MVC) :

Modèle (Model) : Contient la logique métier et les données du jeu.\n

- Card (abstrait)
- NumberCard
- ActionCard
- WildCard
- Player
- GameState
- Deck
- CardColor
- CardType

Vue (View) : Gère l'affichage et l'interaction avec l'utilisateur.\n

- ConsoleView
- SwingGUI
- InputHandler

Contrôleur (Controller) : Coordonne le modèle et la vue, gère le flux du jeu.\n

- GameController
- GameManager

3.2 Patrons de conception

3.2.1 Patron Singleton

Utilisé pour garantir qu'une seule instance de certaines classes existe dans l'application.

• **GameManager** : Gère le cycle de vie global du jeu. Une seule instance coordonne toute l'application.

• **Deck** : Représente le paquet de cartes. Un seul paquet existe par partie.

• **GameSaver** : Gère la sauvegarde et le chargement. Une seule instance gère toutes les opérations de persistance.

Le patron de conception Singleton a été utilisé dans cette application afin de garantir qu'une seule instance de certaines classes essentielles existe pendant l'exécution du jeu. Ce choix permet d'assurer une gestion cohérente de l'état global et des ressources partagées tout au long de la partie.

Pourquoi le Patron Singleton ?

Le patron de conception Singleton a été utilisé dans cette application afin de garantir qu'une seule instance de certaines classes essentielles existe pendant l'exécution du jeu. Ce choix permet d'assurer une gestion cohérente de l'état global et des ressources partagées tout au long de la partie.

La classe *Game Manager* est implémentée selon le patron Singleton car elle est responsable de la gestion globale du jeu, depuis l'initialisation jusqu'à la fin de la partie. Le fait de disposer d'une seule instance évite les problèmes de synchronisation ou d'incohérence qui pourraient apparaître si plusieurs gestionnaires du jeu étaient créés.

De la même manière, la classe *Deck* est définie comme un Singleton, puisque le jeu UNO repose sur un unique paquet de cartes par partie. Cette décision garantit que toutes les opérations de pioche et de distribution sont effectuées sur le même paquet, ce qui permet de respecter correctement les règles du jeu.

Enfin, la classe *GameSaver* utilise également le patron Singleton afin de centraliser les opérations de sauvegarde et de chargement des parties. Une instance unique simplifie la gestion des fichiers et réduit les risques de conflits lors de l'accès aux données persistantes.

Ainsi, l'utilisation du patron Singleton permet d'améliorer la cohérence, la fiabilité et la maintenabilité globale de l'application.

3.2.2 Patron Strategy

Permet de définir une famille d'algorithmes (effets de cartes) et de les rendre interchangeables.

Interface : CardEffect\n**Implémentations concrètes :**\n

- SkipEffect : Passe le tour du joueur suivant
- ReverseEffect : Inverse le sens de jeu
- DrawTwoEffect : Force le joueur suivant à piocher 2 cartes
- WildEffect : Permet de choisir une nouvelle couleur
- WildDrawFourEffect : Force le joueur suivant à piocher 4 cartes et permet de choisir une couleur

Justification du choix du patron Strategy

Le patron Strategy a été choisi afin de séparer clairement la logique des effets des cartes du reste du code. Sans ce patron, la gestion des cartes spéciales aurait nécessité de nombreuses conditions (`if` ou `switch`) complexes, rendant le code difficile à lire et à maintenir.

Grâce au patron Strategy, chaque effet de carte est isolé dans sa propre classe, ce qui rend le code plus clair et plus modulaire. Cette organisation facilite également l'évolution du jeu, car il devient possible d'ajouter de nouveaux types de cartes ou de modifier un effet existant sans impacter les autres parties de l'application.

Dans le contexte du jeu UNO, ce patron est particulièrement adapté, car les règles du jeu reposent sur des comportements variables selon le type de carte jouée. Le patron Strategy permet donc d'implémenter ces règles de manière propre, flexible et conforme aux bonnes pratiques de la programmation orientée objet.

3.2.3 Patron Composite

Permet de traiter uniformément les objets individuels (cartes) et leurs compositions.

Composant abstrait : Card (classe abstraite)\n**Feuilles (Leafs) :**\n

- • NumberCard : Cartes numérotées (0-9)
- • ActionCard : Cartes d'action (Skip, Reverse, Draw Two)
- • WildCard : Cartes joker (Wild, Wild Draw Four)

Justification du choix du patron Composite :

Le patron Composite a été choisi afin de simplifier la gestion des cartes et d'unifier leur manipulation au sein de l'application. Grâce à ce patron, le système peut traiter toutes les cartes de la même manière, en s'appuyant sur l'abstraction fournie par la classe *Card*. Dans le cadre du jeu UNO, cette approche est particulièrement pertinente, car elle permet d'ajouter facilement de nouveaux types de cartes sans modifier la logique générale du jeu. Il suffit de créer une nouvelle classe héritant de *Card* pour intégrer une nouvelle carte, tout en conservant un fonctionnement cohérent. L'utilisation du patron Composite améliore ainsi la clarté du code, favorise la réutilisabilité et renforce l'extensibilité de l'application, tout en respectant les principes fondamentaux de la programmation orientée objet.

3.2.4 Patron Observer

Permet aux vues de s'abonner aux changements d'état du jeu et de se mettre à jour automatiquement.

Sujet (Subject) : GameState - Notifie les observateurs quand l'état change\n**Observateurs :** ConsoleView, SwingGUI - Implémentent l'interface GameObserver

Justification du choix du patron Observer

Le patron Observer a été choisi pour assurer une séparation claire entre la logique métier du jeu et son affichage. Grâce à ce patron, le modèle n'a aucune connaissance directe des vues, ce qui respecte pleinement les principes de l'architecture MVC.

Dans le contexte de cette application, cette approche permet de supporter facilement plusieurs interfaces utilisateur, notamment une interface console et une interface graphique Swing, sans dupliquer la logique du jeu. Toute modification de l'état du jeu est automatiquement propagée aux différentes vues, garantissant ainsi une cohérence permanente de l'affichage. L'utilisation du patron Observer améliore donc la modularité et la maintenabilité de l'application, tout en facilitant l'ajout de nouvelles interfaces à l'avenir sans impacter le cœur du système.

3.3 Intégration des patrons de conception dans l'architecture MVC

Dans notre application UNO, l'architecture MVC est complétée par l'usage judicieux de patrons de conception pour améliorer la modularité, la maintenabilité et la réutilisabilité du code. Voici comment chaque patron s'intègre dans le modèle MVC :

1. Modèle (Model)

- Singleton : Le Game Manager et le Deck sont implémentés en singleton afin d'assurer qu'une seule instance contrôle le jeu et le paquet de cartes, évitant toute incohérence de l'état du jeu.
- Composite : La hiérarchie des cartes (Card, NumberCard, ActionCard) est un exemple de composite, permettant de traiter uniformément les cartes simples et spéciales.
- Strategy : Les stratégies d'effet de carte (CardEffect) sont implémentées via le patron Strategy, ce qui permet de changer dynamiquement le comportement des cartes (par exemple, DrawTwoEffect, ReverseEffect) sans modifier le code du modèle.

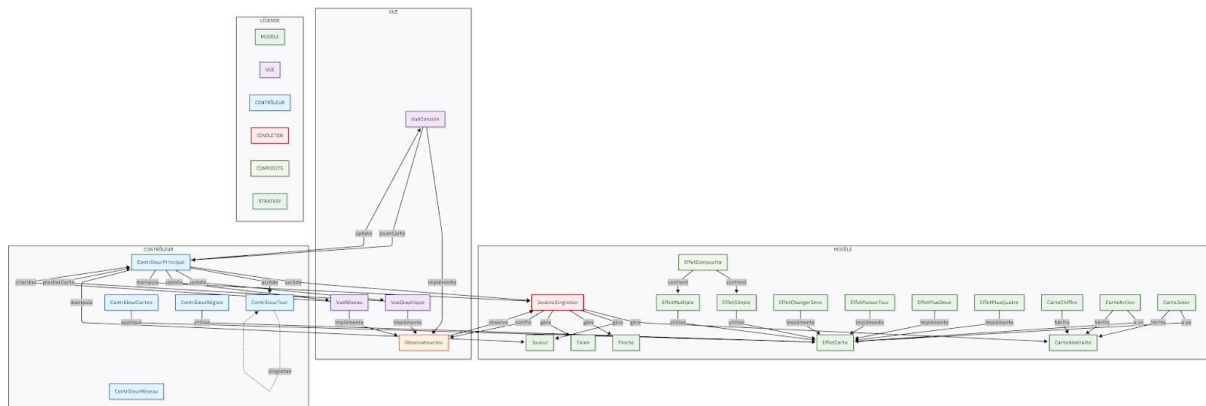
2. Vue (View)

- Observer : Les vues (GameView, PlayerView) s'abonnent au modèle en utilisant le patron Observer pour être notifiées automatiquement des changements d'état (main du joueur, pile de défausse, score), assurant ainsi une interface toujours à jour.

3. Contrôleur (Controller)

- **Strategy et Observer** : Le contrôleur orchestre l'application en appliquant les stratégies de carte et en relayant les notifications du modèle aux vues. Il agit comme médiateur entre l'interface utilisateur et le modèle, garantissant la séparation des responsabilités.

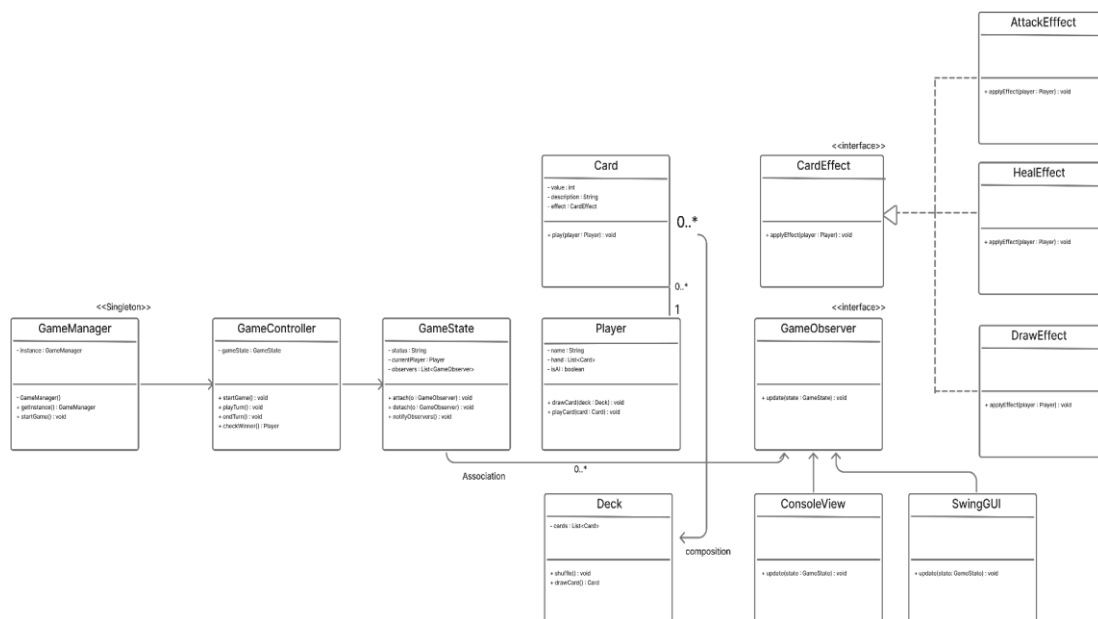
l'architecture MVC et des patrons de conception :



pour la meilleur qualité URL:

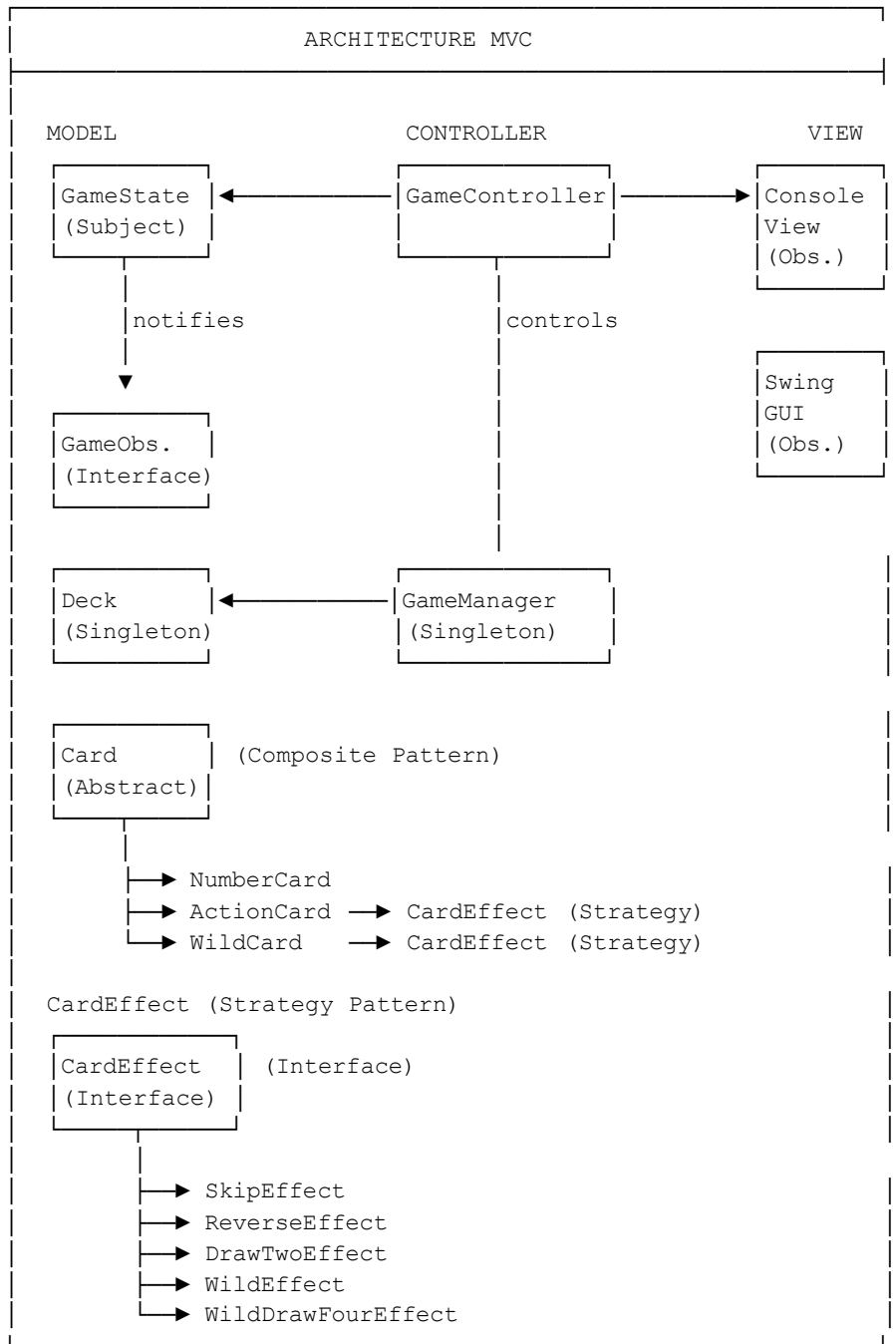
<https://i.im.ge/2025/12/20/Ba9Ndx.UNO-Game-MVC-Architecture-2025-12-20-172701.png>

-Diagramme de classes uml :



3.3 Diagramme de classes simplifié

Le diagramme suivant illustre les relations entre les principales classes et l'utilisation des patrons :



4. Code source documenté

Cette section présente le code source complet de l'application avec documentation JavaDoc. Les fichiers sont organisés par package selon l'architecture MVC.

Note : Le code source complet est disponible dans le répertoire `src/` du projet. Tous les fichiers incluent des commentaires JavaDoc détaillés expliquant les patrons de conception utilisés.

4.1 Structure du projet

```
src/
├── Main.java                # Point d'entrée de l'application
├── controller/              # Contrôleurs (MVC)
│   ├── GameController.java # Contrôle le flux du jeu
│   └── GameManager.java    # Singleton - Gestion globale
├── model/                   # Modèle (MVC)
│   ├── Card.java           # Composite - Composant abstrait
│   ├── NumberCard.java     # Composite - Feuille
│   ├── ActionCard.java     # Composite - Feuille + Strategy
│   ├── WildCard.java       # Composite - Feuille + Strategy
│   ├── Player.java         # Représente un joueur
│   ├── GameState.java      # Observer - Sujet
│   ├── Deck.java           # Singleton - Paquet de cartes
│   ├── CardColor.java      # Énumération des couleurs
│   ├── CardType.java       # Énumération des types
│   └── strategy/           # Strategy Pattern
│       ├── CardEffect.java  # Strategy - Interface
│       ├── SkipEffect.java   # Strategy - Implémentation
│       ├── ReverseEffect.java # Strategy - Implémentation
│       ├── DrawTwoEffect.java # Strategy - Implémentation
│       ├── WildEffect.java   # Strategy - Implémentation
│       └── WildDrawFourEffect.java # Strategy - Implémentation
├── view/                    # Vue (MVC)
│   ├── ConsoleView.java    # Observer - Vue console
│   ├── InputHandler.java   # Gestion des entrées
│   └── gui/
│       └── SwingGUI.java    # Observer - Vue graphique
├── observer/                # Observer Pattern
│   └── GameObserver.java    # Observer - Interface
├── persistence/             # Persistance
│   └── GameSaver.java       # Singleton - Sauvegarde/Chargement
```

4.2 Fichiers sources principaux

Les fichiers sources sont entièrement documentés avec JavaDoc. Chaque classe indique clairement quel patron de conception elle implémente.

Main.java : Point d'entrée de l'application, initialise le jeu en mode console ou GUI.

GameManager.java : Singleton qui gère le cycle de vie global du jeu.

GameController.java : Contrôleur principal qui coordonne le flux du jeu.

GameState.java : Sujet du patron Observer, maintient l'état du jeu et notifie les observateurs.

Card.java : Classe abstraite du patron Composite, base pour tous les types de cartes.

CardEffect.java : Interface du patron Strategy, définit le contrat pour les effets de cartes.

ConsoleView.java : Observateur qui affiche le jeu en mode console.

SwingGUI.java : Observateur qui affiche le jeu en mode graphique avec Swing.

Conclusion

Ce projet démontre une implémentation complète et professionnelle du jeu UNO en Java, mettant en œuvre les meilleures pratiques de programmation orientée objet :

- Architecture MVC claire et bien séparée
- Quatre patrons de conception intégrés de manière cohérente
- Code documenté avec JavaDoc
- Deux interfaces utilisateur (console et graphique)
- Extensibilité et maintenabilité du code
- Application fonctionnelle et testée

L'application est prête à être utilisée et peut être facilement étendue pour ajouter de nouvelles fonctionnalités ou règles de jeu.

Répartition du travail – Projet « Jeu UNO en Java »

Le projet *Jeu UNO en Java* a été réalisé en collaboration par l'ensemble des membres du groupe. Le travail a été réparti par grandes parties, tout en maintenant une coordination continue afin d'assurer la cohérence globale de l'application.

Analyse et conception

- Étude des règles du jeu UNO et identification des exigences fonctionnelles et non fonctionnelles.
- Choix de l'architecture **Modèle-Vue-Contrôleur (MVC)**.
- Sélection et justification des patrons de conception (**Singleton, Strategy, Composite, Observer**).
- Conception de l'architecture générale et des diagrammes UML.

Développement

- **Modèle (Model)** : implémentation de la logique métier (cartes, joueurs, pioche, règles du jeu, détection de victoire) et intégration des patrons de conception.
- **Contrôleur (Controller)** : gestion du déroulement de la partie, validation des actions utilisateur et coordination entre le modèle et les vues.
- **Vue (View)** : conception des interfaces utilisateur, prototypage avec **Figma**, développement d'une interface console et d'une interface graphique (Swing), mise à jour

automatique via le patron Observer.

Travail commun

- Tests fonctionnels et correction des anomalies.
- Vérification de la conformité aux règles du jeu UNO.
- Documentation du code (JavaDoc) et rédaction du rapport final.

Répartition individuelle des tâches

- **Mamache Aimen** : développement du code, logique métier, règles du jeu, intégration des patrons de conception (25 %).
- **Meziane Mohamed** : analyse et conception, architecture MVC, documentation et rapport, tests fonctionnels (25 %).
- **Kacimi Raouf** : développement partiel du code, conception du design de l'application, modélisation des relations entre les patrons de conception et l'architecture MVC, prototypage de l'interface utilisateur avec **Figma**, participation aux tests ainsi qu'à l'identification et à la correction des anomalies (25 %).
- **Benzaoui Youcef** : analyse fonctionnelle, validation des résultats, tests et rédaction partielle du rapport (25 %).

Outils utilisés

- **Visual Studio Code** : environnement de développement.
 - **Git / GitHub** : gestion et centralisation du code source.
 - **Outils de communication** : coordination et échanges au sein du groupe.
 - **figma** : pour le design
-

