Rapport de Projet – Développement du Jeu Solo Chess avec LibGDX

Jack Massey (Projet individuel)

Section 1. Introduction

1.1 Contexte du Projet

Ce projet, intitulé **Solo Chess**, a pour objectif de proposer un jeu de réflexion stratégique inspiré des échecs, mais avec des règles simplifiées et une mécanique centrée sur la capture de pièces jusqu'à ce qu'il n'en reste plus qu'une seule. Le développement a été réalisé **en solo** par Jack Massey, ce qui implique que tous les rôles (conception, programmation, design, etc.) ont été assurés par une seule personne.

1.2 Objectifs

- **Objectif principal:**
  Développer un jeu en 2D, basé sur LibGDX, permettant de jouer à une variante des échecs appelée "Solo Chess".
  `Solo Chess - Lien du jeu sur chess.com`
- **Concevoir un code de qualité:**
  Appliquer plusieurs **design patterns** (Observer, Factory, State, MVC) afin de produire un code propre, modulable et maintenable. Cela implique une réflexion constante sur l'architecture et la capacité à **s'adapter aux difficultés** rencontrées (ex. création ou suppression de classes inutiles, ajustements suite à l'évolution des besoins).
- **Extensibilité et amélioration continue:**
  - Mettre en place un **moteur de génération ou de configuration de positions** permettant d'ajouter aisément de nouvelles configurations/puzzles.
  - Prévoir des fonctionnalités d'**amélioration de l'interface** (ex. possibilité de revenir en arrière, ajout de boutons, améliorations graphiques...).
  - Faciliter la maintenance à long terme en veillant à ce que chaque module (moteur de jeu, affichage, contrôleur) reste **indépendant** et **extensible**.
- **Gestion de projet et développement en solo:**
  - Acquérir et démontrer la **capacité à piloter un projet** de bout en bout (depuis la phase de conception jusqu'à l'implémentation).
  - Développer un **savoir-faire** en gestion de code (choix de technologies, organisation du référentiel, documentation) et en conduite de projet (gestion du temps, prise de décisions, résolution de problèmes).

1.3 Règles du Jeu

Les règles principales de **Solo Chess** sont les suivantes:

1. L'échiquier est initialisé avec plusieurs pièces, placées aléatoirement ou selon une configuration prédéfinie.
2. Chaque pièce peut se déplacer en respectant ses mouvements traditionnels aux échecs.

3. Une pièce ne peut effectuer que **deux mouvements** maximum. Après son deuxième mouvement, elle devient "noire", immobile et inutilisable.

4. Les pièces peuvent **capturer** d'autres pièces.

5. Le jeu se termine lorsqu'il ne reste plus qu'une pièce :

   - Si un **roi** est présent au départ, il doit être la dernière pièce restante.
   - Sinon, toute pièce restante peut conclure la partie.

6. Si toutes les pièces restantes sont "noires" avant d'en avoir une seule pièce libre à la fin, le joueur perd immédiatement.

1.4 Design Patterns Utilisés

Pour structurer ce projet, plusieurs *design patterns* de la programmation orientée objet ont été utilisés:

1. **Observer / Observable**
   Permet la communication entre le modèle (`Board`) et les différentes vues (par ex. `Renderer`). La classe `Board` implémente l'interface `Observable` et notifie les `Observer` (ex. le renderer) lorsqu'il y a un changement d'état (déplacement de pièce, capture, etc.).

2. **Factory Pattern**
   La classe `PieceFactory` est utilisée pour créer dynamiquement les différentes pièces (Roi, Dame, etc.). Cela facilite l'ajout de nouveaux types de pièces sans modifier la logique principale de création.

3. **State Pattern**
   Gère les différents états du jeu : en cours (`PlayingState`), en pause (`PausedState`), terminé (`GameOverState`). Chaque état implémente une interface commune (ex. `GameStateInterface`) pour uniformiser leur gestion (mise à jour, rendu).

4. **Model-View-Controller (MVC)**

   - **Modèle (`Board`)** : gère les données et la logique métier (position des pièces, captures...).
   - **Vue (`Renderer`)** : affiche l'état du jeu (pièces, échiquier).
   - **Contrôleur (`Controller`)** : gère les interactions utilisateur, orchestre la communication entre modèle et vue.

Section 2. Présentation du Projet

Cette section détaille l'approche technique adoptée pour **Solo Chess**, notamment l'utilisation de LibGDX, la gestion de l'affichage via Tiled et la structure générale du jeu.

2.1 Technologies et Outils Utilisés

- **LibGDX**
  Principal moteur de jeu 2D, utilisé pour gérer l'affichage (rendu graphique), la détection des entrées utilisateurs (clavier/souris) et le cycle de vie du jeu.

- **Tiled**
  Outil de conception de cartes 2D. Dans ce projet, Tiled a servi à élaborer la représentation visuelle de l'échiquier (en alternant des tuiles de deux couleurs) et à placer (via un calque spécifique) les différentes pièces d'échecs.

- **Git**
  Utilisé pour le versionnement du code et la gestion de différentes branches expérimentales.
  - Par exemple, la branche `SaveTheKing` illustre une tentative d'implémenter un **algorithme Minimax** pour gérer les déplacements et décisions de l'IA. Contrainte par le temps, cette approche n'a pas été finalisée, et une solution plus adaptée a ensuite été privilégiée.
  - Une autre branche a expérimenté l'intégration de Tiled via un *Tile Layer*, un *Map Loader*, etc.
- **IntelliJ**
  Environnement de développement (IDE) pour écrire et organiser le code, compiler et exécuter facilement le projet.
- **Gradle**
  Outil d'automatisation et de gestion de dépendances. Il permet de configurer et de lancer rapidement le projet via des tâches dédiées (par exemple, `gradlew desktop:run` dans le cas d'un projet LibGDX classique).
- **LLM / IA Assistants**
  - **GitHub Copilot** : assisté à la génération de portions de code et à la suggestion de snippets.
  - **ChatGPT** : support dans la conception et la documentation, pour clarifier des choix d'architecture ou résoudre certains problèmes de conception.

2.2 Fonctionnalités Implémentées

- **Déplacements de pièces**
  Chaque pièce (roi, dame, tour, fou, cavalier, pion) suit ses règles de mouvement spécifiques, conformément aux règles traditionnelles des échecs. *Remarque : dans la version actuelle, l'animation graphique des déplacements n'est pas encore finalisée.*
- **Captures**
  Les captures sont possibles si une pièce peut se rendre sur la case occupée par une autre pièce (alliée ou ennemie, selon la configuration du puzzle). La pièce capturée est alors retirée de l'échiquier. *Remarque : la visualisation des pièces capturées reste basique (pas d'effet visuel avancé pour le moment).*
- **Limitation à deux mouvements**
  Après deux déplacements, la pièce devient "noire" et ne peut plus bouger ni capturer. Cette mécanique est centrale dans Solo Chess et confère au jeu son aspect puzzle.
- **Condition de fin de partie**
  Le jeu se termine lorsque :
  1. Il ne reste plus qu'une seule pièce sur l'échiquier (roi obligatoire s'il était présent au départ).
  2. Ou lorsque toutes les pièces encore présentes sont devenues "noires" (immobiles), rendant la situation sans issue.

2.3 Configuration et Ajout de Contenu (Tiled)

Pour faciliter la création et la modification du plateau de jeu, **Tiled** a été utilisé :

1. **Board Layer**
   - Le plateau d'échecs est composé d'un calque (layer) contenant des tuiles de taille 64×64.
   - Les tuiles alternent entre deux couleurs (ex. **dark green** et **light white**) afin de reproduire un échiquier classique.

- Dans le fichier `.tmx`, chaque tuile est identifiée par un ID (ou "GID") représentant la couleur.

2. **Piece Layer**
   - Un deuxième calque, dit "piece layer", définit la position initiale des pièces.
   - Les différentes pièces sont référencées par des GID qui pointent vers un **tileset** associé (ex. un ensemble d'images représentant roi, dame, fou, etc.).
   - Chaque tuile du `piece layer` possède des **propriétés** (ex. nom de la pièce, couleur, etc.) qui sont lues au chargement pour instancier la bonne classe de pièce via la `PieceFactory`.
3. **Chargement de la carte**
   - Une fois la carte `.tmx` créée, LibGDX et son `MapLoader` chargent les informations (calques, tuiles, propriétés).
   - Le code identifie la couche "Board Layer" (pour le visuel de l'échiquier) et la couche "Piece Layer" (pour instancier les pièces).
   - Les propriétés du tileset (couleur de la pièce, type de la pièce) sont analysées afin de créer les instances via le `PieceFactory` et de positionner les pièces correctement dans la matrice interne (`Board`).
4. **Retour d'expérience / Branches expérimentales**
   - Une première tentative de gérer l'affichage et la logique entièrement via les *Tile Layers* (avec un `TileLoader`, `TileMap`, etc.) n'a pas abouti. Cette approche s'est avérée trop complexe pour gérer à la fois l'état visuel et l'état logique du jeu.
   - Finalement, la gestion logique est confiée à une **matrice de `Tile`** (modèle interne) indépendante, tandis que Tiled sert surtout de source initiale pour positionner les pièces et styliser l'échiquier.

Grâce à cette organisation, il est possible de **modifier aisément** la configuration du plateau (dimensions, placements, etc.) sans toucher au code Java, en ajustant simplement la carte `.tmx` ou le tileset associé.

Section 3. Architecture Générale du Moteur de Jeu (Aperçu)

Le jeu suit une **architecture modulaire**, basée sur les *design patterns* décrits précédemment. Les classes clés incluent :

- **`Point`** : représente les coordonnées sur l'échiquier.
- **`Tile`** (classe abstraite) : spécialisations
  - `EmptyTile`
  - `OccupiedTile`
- **`Piece`** (classe abstraite) :
  - `King`, `Queen`, `Bishop`, `Knight`, `Rook`, `Pawn`, etc.
- **`Board`** : gère la logique métier (placements, déplacements, captures) et notifie les observateurs via l'interface `Observable`.
- **`Renderer`** : se charge d'afficher l'échiquier et les pièces à l'écran (vue).
- **`Controller`** : gère les interactions utilisateur, valide les actions et communique avec le modèle (`Board`) et la vue (`Renderer`).

- `GameStateInterface` : définit l'interface commune pour les différents états du jeu (`PlayingState`, `PausedState`, `GameOverState`).

  Pour consulter l'**UML complet** associé à ce projet :

  - Un répertoire `UML` se trouve à la racine du projet.
  - Vous y trouverez notamment :
    - **`SoloChess.puml`** (UML principal, version actuelle).
    - **`SaveTheKing.puml`** (ancienne version, centrée sur une IA Minimax).
    - **`tsxtmxrelationships.puml`** (décrit les relations liées à Tiled).

Cette structuration, combinée aux patrons de conception (Observer, Factory, State, MVC), rend le moteur de jeu **extensible** et **maintenable**, tout en séparant clairement la logique de la présentation.

Section 4. Conclusion et Perspectives

4.1 Bilan et Retour d'Expérience

La réalisation du projet **Solo Chess** s'est avérée être une *expérience enrichissante* à plusieurs niveaux :

- **Découverte et apprentissage de LibGDX**
  Partir de zéro sur une librairie méconnue (LibGDX) a constitué un vrai défi. Il a fallu assimiler le fonctionnement du *cycle de vie* d'un jeu (render, update, input), la configuration des projets avec Gradle, et les principes de gestion des assets (images, tuiles, etc.).
- **Architecture et patrons de conception**
  L'implémentation de *design patterns* (Observer, Factory, State, MVC) s'est parfois faite **par tâtonnements**. Certaines idées (par exemple un *Strategy Pattern* pour l'IA) ont été explorées puis écartées ou reportées, illustrant l'évolution naturelle d'un projet où l'on teste, on se trompe, et on apprend.
- **Autonomie et gestion du projet en solo**
  Mener ce projet seul a nécessité de **couvrir tous les rôles** (concepteur, développeur, testeur...) et de prendre des décisions rapides, tout en restant flexible. Ce processus a renforcé la capacité à **identifier les priorités** et à **faire face aux imprévus** (bugs, limitations techniques, etc.).
- **Essais, erreurs et abandons**
  Plusieurs pistes ont dû être abandonnées en cours de route:
  - Une implémentation en JSON pour la configuration des pièces, finalement jugée superflue grâce à l'utilisation de Tiled.
  - Un algorithme Minimax pour gérer une IA "SaveTheKing", qui n'a pas pu être finalisé dans le cadre temporel imparti.

Au final, la version actuelle du projet **n'est pas totalement achevée** : l'affichage graphique est encore minimal, il manque une interface plus conviviale (boutons, menus...) et des animations plus abouties. Toutefois, la base de la logique de Solo Chess (déplacements, captures, limites de mouvement) est solide et démontre la **faisabilité** du concept.

4.2 Perspectives d'Amélioration

- **Interface Utilisateur et Rendu**
  - Améliorer l'affichage des pièces et ajouter des effets visuels (surbrillance des coups valides, animation de capture, etc.).
  - Intégrer des boutons et menus pour la navigation (revenir en arrière, réinitialiser la partie, etc.).
- **Évolutions Techniques**
  - Sauvegardes et chargements : utiliser éventuellement JSON (ou un autre format) pour stocker/reprendre l'état de la partie.
  - Gestion avancée de Tiled : approfondir l'intégration de Tiled pour définir différentes configurations plus facilement.
  - IA plus élaborée : adapter ou réintroduire le *Strategy Pattern* (voire Minimax) pour proposer des défis dynamiques, ou générer automatiquement des puzzles plus complexes.
- **Exploration de nouvelles voies**
  - Poursuivre la logique de théorie des graphes appliquée aux positions d'échecs, afin d'automatiser la création de puzzles.

En somme, **Solo Chess** a été un terrain d'expérimentation vaste: il a permis de tester des *concepts variés* (POO, design patterns, intégration d'assets via Tiled, versionnement Git), tout en soulignant l'importance d'*apprendre par l'erreur* et d'*itérer* pour faire évoluer la conception. Malgré les fonctionnalités manquantes et les limites actuelles, ce projet constitue un **socle** solide pour de futures améliorations et enrichissements.

Section Annexe

- Lien vers le dépôt Git : `https://github.com/TerminalGambit/PCOO-projetfinal`