Projet de Module

Design Patterns

Développement d'un Jeu Vidéo

Application des Patrons de Conception

Enseignant : Haythem Ghazouani

Année universitaire : 2025-2026

Durée du projet : Toute la durée du module

3 octobre 2025

Table des matières

1 Présentation Générale du Projet

1.1 Objectifs Pédagogiques

Ce projet vise à consolider vos connaissances en programmation orientée objet et à maîtriser l'application pratique des patrons de conception (Design Patterns) dans un contexte réel de développement logiciel.

Compétences Visées

- Comprendre et appliquer au moins 4 patrons de conception.
- Concevoir une architecture logicielle maintenable et extensible.
- Développer une interface graphique utilisateur.
- Implémenter un système de traçabilité (logging).
- Utiliser un système de gestion de versions (Git).
- Produire une documentation technique (diagramme de classes).

1.2 Description du Projet

Vous devez développer un **petit jeu vidéo** en appliquant les principes de conception orientée objet et en intégrant plusieurs patrons de conception. Le jeu doit être fonctionnel, interactif et démontrer une compréhension approfondie des Design Patterns.

1.3 Contraintes Techniques

- **Langage :** Java (recommandé) ou tout autre langage orienté objet (C#, Python, C++, etc.).
- **Interface graphique :** Obligatoire (même simple).
- **Patrons de conception :** Minimum 4 patterns obligatoires.
- **Système de logging :** Traçabilité des événements importants.
- **Gestion de versions :** Utilisation de Git obligatoire.

2 Spécifications Techniques Détaillées

2.1 Patrons de Conception Obligatoires

Vous devez implémenter **au minimum 4 patrons de conception** parmi lesquels :

</> Patterns Obligatoires

`State Pattern` **(Obligatoire)**

— Gestion des différents états du jeu (Menu, En cours, Pause, Game Over, Victoire).
— Gestion des états des personnages (Idle, Running, Jumping, Attacking, Dead).
— Transitions d'états avec traçabilité dans les logs.

`Decorator Pattern` **(Obligatoire)**

— Ajout dynamique de capacités aux personnages (bouclier, vitesse, armes).
— Power-ups et bonus temporaires.
— Effets visuels additionnels.
— **Chaque décoration doit être tracée dans le fichier log.**

`Composite Pattern` **(Obligatoire)**

— Structure hiérarchique des éléments du jeu (scènes, niveaux, objets).
— Gestion des groupes d'ennemis ou d'objets.
— Organisation des composants graphiques.

2.2 Patrons de Conception Additionnels (Choisir au moins 1)

+ Patterns Recommandés

`Singleton`	Gestionnaire de jeu, gestionnaire de ressources, système de logging.
`Factory`	Création d'ennemis, d'obstacles, de power-ups.
`Observer`	Système d'événements, mise à jour du score, notifications.
`Strategy`	Comportements d'IA des ennemis, algorithmes de déplacement.
`Command`	Gestion des commandes utilisateur, système d'undo/redo.
`Facade`	Simplification de l'interface avec les sous-systèmes complexes.
`Builder`	Construction de niveaux complexes.
`Prototype`	Clonage d'objets de jeu.

2.3 Interface Graphique

2.3.1 Frameworks Recommandés pour Java

2.3.2 Exigences Minimales de l'Interface

— **Menu principal** : Démarrer, Options, Quitter.
— **Zone de jeu** : Affichage du personnage, ennemis, obstacles, décors.
— **HUD (Head-Up Display)** : Score, vies, temps, power-ups actifs.
— **Écrans de transition** : Game Over, Victoire, Pause.

Framework	Caractéristiques	Difficulté
JavaFX	— Interface moderne et riche. — Excellente pour les jeux 2D. — Bonne documentation. — Support des animations.	**
Swing	— Mature et stable. — Facile à apprendre. — Nombreux tutoriels disponibles. — Intégré au JDK.	*
LibGDX	— Framework de jeu complet. — Haute performance. — Multi-plateforme. — Courbe d'apprentissage plus élevée.	***
Processing	— Très simple pour débuter. — Orienté graphique. — Idéal pour prototypage rapide.	*

TABLE 1 – Comparaison des frameworks graphiques

— **Contrôles** : Clavier et/ou souris.

2.4 Système de Traçabilité (Logging)

2.4.1 Événements à Tracer

Logging Obligatoire

Le fichier log doit enregistrer **au minimum** :

1. **Changements d'états** :
   — États du jeu (Menu → En cours → Pause → Game Over).
   — États des personnages (Idle → Running → Jumping).
   — Timestamp de chaque transition.
2. **Applications de décorateurs** :
   — Type de décorateur appliqué.
   — Objet décoré.
   — Durée d'effet (si applicable).
   — Timestamp d'application et de retrait.
3. **Événements de jeu importants** :
   — Démarrage et fin de partie.
   — Création/destruction d'entités.
   — Collisions importantes.
   — Changements de niveau.

2.4.2 Format du Fichier Log

```
[2024-12-15 14:23:45] [INFO] Game started
[2024-12-15 14:23:47] [STATE] Game: MENU -> PLAYING
[2024-12-15 14:23:50] [STATE] Player: IDLE -> RUNNING
[2024-12-15 14:23:52] [DECORATOR] SpeedBoost applied to Player
[2024-12-15 14:23:55] [STATE] Player: RUNNING -> JUMPING
[2024-12-15 14:24:02] [DECORATOR] SpeedBoost removed from Player
[2024-12-15 14:24:05] [DECORATOR] Shield applied to Player
[2024-12-15 14:24:10] [STATE] Game: PLAYING -> GAME_OVER
[2024-12-15 14:24:10] [INFO] Final score: 1250
```

2.4.3 Bibliothèques de Logging Recommandées

— **Java** : Log4j2, SLF4J + Logback, java.util.logging.

— **C#** : NLog, Serilog, log4net.

— **Python** : logging (module standard), loguru.

3 Propositions de Jeux

3.1 Jeux Recommandés

Idées de Jeux

1. Mini Super Mario (Platformer)

— **Patterns applicables** : State (états du personnage), Decorator (power-ups), Composite (niveaux), Factory (ennemis).

— **Complexité** : Moyenne.

— **Gameplay** : Déplacement horizontal, sauts, collecte de pièces, ennemis.

2. Tower Defense Simplifié

— **Patterns applicables** : Strategy (comportements des tours), Composite (vagues d'ennemis), Decorator (améliorations), Observer (événements).

— **Complexité** : Moyenne-Élevée.

— **Gameplay** : Placement de tours, gestion de ressources, vagues d'ennemis.

3. Space Invaders Amélioré

— **Patterns applicables** : State (états du jeu), Composite (formations d'ennemis), Decorator (armes), Factory (projectiles).

— **Complexité** : Faible-Moyenne.

— **Gameplay** : Tir vertical, esquive, power-ups, boss.

4. RPG Combat Simplifié

— **Patterns applicables** : State (états de combat), Decorator (équipements), Strategy (IA ennemis), Command (actions).

— **Complexité** : Moyenne.

— **Gameplay** : Combat au tour par tour, inventaire, compétences.

5. Puzzle Game (Tetris-like)

— **Patterns applicables** : State (états des pièces), Composite (grille), Factory (génération de pièces), Command (rotations).

— **Complexité** : Faible-Moyenne.

— **Gameplay** : Placement de pièces, élimination de lignes, score.

6. Pac-Man Revisité

— **Patterns applicables** : State (états des fantômes), Strategy (IA), Decorator (power-ups), Observer (événements).

— **Complexité** : Moyenne.

— **Gameplay** : Labyrinthe, collecte, ennemis avec IA.

3.2 Critères de Choix du Jeu

— **Faisabilité** : Le jeu doit être réalisable dans le temps imparti.
— **Applicabilité des patterns** : Doit permettre l'utilisation naturelle de 4+ patterns.
— **Intérêt technique** : Présente des défis de conception intéressants.
— **Originalité** : Vous pouvez proposer votre propre idée (validation requise).

4 Livrables et Évaluation

4.1 Livrables Obligatoires

1. **Code source complet**
   — Code bien structuré et commenté.
   — Respect des conventions de nommage.
   — Organisation claire des packages/modules.
2. **Diagramme de classes UML global**
   — Représentation de l'architecture complète.
   — Mise en évidence des patterns utilisés.
   — Format : PDF ou image haute résolution.
   — Outils recommandés : PlantUML, draw.io, StarUML, Visual Paradigm.
3. **Dépôt Git**
   — Historique de commits réguliers et significatifs.
   — README.md avec instructions d'installation et d'exécution.
   — Fichier .gitignore approprié.
   — Branches si travail collaboratif.
4. **Fichier de log**
   — Démonstration de la traçabilité.
   — Exemple de session de jeu complète.
5. **Exécutable ou instructions de compilation**
   — JAR exécutable (Java) ou équivalent.
   — Documentation de dépendances.

4.2 Soutenance Technique

Modalités de Soutenance

Date : Dernière séance du module. **Durée :** 15-20 minutes par groupe. **Format :**

- **Démonstration** (5-7 min) : Présentation du jeu fonctionnel.
- **Explication technique** (8-10 min) :
  - Architecture globale.
  - Patterns implémentés avec exemples de code.
  - Système de logging.
  - Défis rencontrés et solutions.
- **Questions/Réponses** (5 min) : Discussion technique avec l'enseignant.

Support : Diagramme de classes + code source + jeu en exécution.

4.3 Critères d'Évaluation

Critère	Description	Points
Patterns	Application correcte et pertinente d'au moins 4 patterns.	30%
Architecture	Qualité de la conception, maintenabilité, extensibilité.	20%
Fonctionnalité	Jeu fonctionnel, interface graphique, gameplay.	20%
Logging	Système de traçabilité complet et pertinent.	10%
Documentation	Diagramme UML, README, commentaires.	10%
Git	Utilisation appropriée, commits réguliers.	5%
Soutenance	Clarté de présentation, maîtrise technique.	5%

TABLE 2 – Grille d'évaluation

5 Organisation du Travail

5.1 Composition des Groupes

- **Taille recommandée** : 2 à 4 étudiants.
- **Travail individuel** : Possible mais plus exigeant.
- **Groupes de 4** : Nécessite une répartition claire des tâches.

5.2 Répartition des Tâches Suggérée

Organisation en Équipe

Pour un groupe de 4 personnes :

Membre 1 Architecture globale, patterns State et Composite.

Membre 2 Interface graphique, intégration visuelle.

Membre 3 Patterns Decorator et Factory, système de logging.

Membre 4 Gameplay, logique métier, tests.

Note : Tous les membres doivent comprendre l'ensemble du projet !

5.3 Planning Recommandé

Semaine	Tâches
1-2	— Choix du jeu. — Conception de l'architecture. — Identification des patterns. — Diagramme UML préliminaire.
3-4	— Implémentation des patterns de base. — Structure du projet. — Configuration Git.
5-7	— Développement de l'interface graphique. — Implémentation du gameplay. — Intégration des patterns.
8-9	— Système de logging. — Tests et débogage. — Finalisation du diagramme UML.
10-11	— Polissage et optimisation. — Documentation. — Préparation de la soutenance.
12	Soutenance technique.

TABLE 3 – Planning indicatif sur 12 semaines

6 Ressources et Outils

6.1 Gestion de Versions (Git)

6.1.1 Plateformes Recommandées

— **GitHub** : https://github.com (le plus populaire).

— **GitLab** : https://gitlab.com (CI/CD intégré).

— **Bitbucket** : https://bitbucket.org (intégration Jira).

6.1.2 Bonnes Pratiques Git

git Conventions Git

Messages de commit :

```
feat: Add player jump mechanic
fix: Correct collision detection bug
refactor: Implement State pattern for game states
docs: Update README with installation instructions
```

Structure de branches :

— `main` : Version stable.

— `develop` : Développement en cours.

— `feature/nom-feature` : Nouvelles fonctionnalités.

— `fix/nom-bug` : Corrections de bugs.

6.2 Outils de Modélisation UML

— **PlantUML** : Génération de diagrammes par code (recommandé).

— **draw.io** : Gratuit, en ligne, facile d'utilisation.

— **StarUML** : Professionnel, version gratuite disponible.

— **Visual Paradigm** : Complet, version communautaire gratuite.

— **Lucidchart** : En ligne, collaboratif.

6.3 Documentation et Tutoriels

6.3.1 Design Patterns

— **Livre de référence** : "Design Patterns : Elements of Reusable Object-Oriented Software" (Gang of Four).

— **Site web** : https://refactoring.guru/design-patterns.

— **Exemples Java** : https://github.com/iluwatar/java-design-patterns.

6.3.2 Développement de Jeux

— **JavaFX** : https://openjfx.io/.

— **LibGDX** : https://libgdx.com/.

— **Game Programming Patterns** : https://gameprogrammingpatterns.com/.

7 Exemples d'Implémentation

7.1 Exemple : State Pattern pour le Joueur

```
// Interface State
public interface PlayerState {
```

```
    void handleInput(Player player, Input input);
    void update(Player player);
}
// États concrets
public class IdleState implements PlayerState {
    @Override
    public void handleInput(Player player, Input input) {
        if (input.isPressed(Key.SPACE)) {
            Logger.log("STATE", "Player: IDLE -> JUMPING");
            player.setState(new JumpingState());
        } else if (input.isPressed(Key.RIGHT)) {
            Logger.log("STATE", "Player: IDLE -> RUNNING");
            player.setState(new RunningState());
        }
    }

    @Override
    public void update(Player player) {
        player.setVelocity(0, 0);
    }
}
```

7.2 Exemple : Decorator Pattern pour Power-ups

```
// Composant de base
public interface Character {
    void draw();
    int getSpeed();
    int getStrength();
}
// Décorateur abstrait
public abstract class PowerUpDecorator implements Character {
    protected Character decoratedCharacter;

    public PowerUpDecorator(Character character) {
        this.decoratedCharacter = character;
        Logger.log("DECORATOR",
            this.getClass().getSimpleName() +
            " applied to " + character.getClass().getSimpleName());
    }
}
// Décorateurs concrets
public class SpeedBoost extends PowerUpDecorator {
    public SpeedBoost(Character character) {
        super(character);
    }

    @Override
```

```
    public int getSpeed() {
        return decoratedCharacter.getSpeed() * 2;
    }
}
```

7.3 Exemple : Composite Pattern pour les Niveaux

```
// Composant
public interface GameComponent {
    void update();
    void render();
    void add(GameComponent component);
    void remove(GameComponent component);
}
// Composite
public class Level implements GameComponent {
    private List<GameComponent> components = new ArrayList<>();

    @Override
    public void add(GameComponent component) {
        components.add(component);
    }
}
```

8 Questions Fréquentes (FAQ)

FAQ

Q : Puis-je utiliser un autre langage que Java ? R : Oui, tout langage orienté objet est acceptable (C#, Python, C++, etc.), mais Java est fortement recommandé.

Q : Dois-je créer un jeu complexe avec de nombreux niveaux ? R : Non, la complexité du gameplay n'est pas le critère principal. L'accent est mis sur la qualité de l'architecture.

Q : Combien de patterns dois-je implémenter au maximum ? R : Il n'y a pas de maximum, mais 4-6 patterns bien implémentés sont suffisants.

Q : Le diagramme UML doit-il inclure tous les détails ? R : Le diagramme doit montrer l'architecture globale et mettre en évidence les patterns.

Q : Puis-je utiliser des bibliothèques externes ? R : Oui, pour l'interface graphique, le logging, etc. Documentez toutes les dépendances.

Q : Comment gérer les conflits Git dans un groupe ? R : Utilisez des branches séparées pour chaque fonctionnalité et communiquez régulièrement.

Q : Que faire si mon jeu ne fonctionne pas parfaitement le jour de la soutenance ? R : Concentrez-vous sur la démonstration de l'architecture et des patterns.

9 Conseils et Recommandations

9.1 Conseils de Conception

Bonnes Pratiques

1. Commencez simple : Implémentez d'abord une version basique.
2. Pensez extensibilité : Votre architecture doit permettre d'ajouter facilement de nouvelles fonctionnalités.
3. Documentez au fur et à mesure.
4. Testez régulièrement.
5. Utilisez les patterns naturellement.
6. Revue de code : Faites relire votre code par vos coéquipiers.
7. Commits atomiques.
8. Préparez la soutenance tôt.

9.2 Pièges à Éviter

Attention

— Over-engineering : N'ajoutez pas de complexité inutile.
— Patterns forcés : Utilisez les patterns là où ils apportent une vraie valeur.
— Manque de communication : Dans un groupe, communiquez régulièrement.
— Procrastination : Ne commencez pas le projet la dernière semaine.
— Ignorer Git : Utilisez Git dès le début.
— Négliger le logging : Implémentez le système de traçabilité dès le début.
— Code non commenté : Commentez les parties complexes et les patterns.

10 Contact et Support

10.1 Informations de Contact

Contact

Enseignant : Haythem Ghazouani. **Pour toute question :**

— Pendant les séances de cours/TD.
— Par email (consulter la plateforme pédagogique).
— Heures de permanence (à définir).

Validation du sujet : Si vous proposez votre propre idée de jeu, faites-la valider avant de commencer l'implémentation.

10.2 Dates Importantes

Événement	**Date**
Lancement du projet	Première séance du module.
Validation du sujet	À définir.
Point d'avancement (optionnel)	Mi-semestre.
Dépôt du code et diagramme	Avant la dernière séance.
Soutenance technique	Dernière séance du module.

TABLE 4 – Calendrier du projet

A Checklist de Validation

A.1 Avant la Soutenance

Le jeu est fonctionnel et démarre sans erreur.

Au moins 4 design patterns sont correctement implémentés.

Le système de logging fonctionne et trace les événements requis.

Le diagramme UML est complet et à jour.

Le dépôt Git contient un historique de commits cohérent.

Le README.md contient les instructions d'installation et d'exécution.

Le code est commenté et bien structuré.

L'interface graphique est fonctionnelle.

La présentation de soutenance est prête.

Tous les membres du groupe maîtrisent l'ensemble du projet.

B Template README.md

```
# Nom du Projet
## Description
Brève description du jeu et de ses fonctionnalités principales.
## Membres du Groupe
- Nom Prénom 1
- Nom Prénom 2
- Nom Prénom 3
## Technologies Utilisées
- Langage : Java 17
- Framework GUI : JavaFX
- Logging : Log4j2
- Build : Maven/Gradle
## Design Patterns Implémentés
1. **State Pattern** : Gestion des états du jeu et du joueur.
2. **Decorator Pattern** : Système de power-ups.
3. **Composite Pattern** : Structure hiérarchique des niveaux.
4. **Factory Pattern** : Création des ennemis.
## Installation
### Prérequis
- JDK 17 ou supérieur
- Maven 3.6+
### Étapes
1. Cloner le dépôt : `git clone [URL]`
2. Compiler : `mvn clean install`
3. Exécuter : `java -jar target/game.jar`
## Utilisation
- Flèches directionnelles : Déplacement
- Espace : Saut
- Échap : Pause
```

```
## Structure du Projet
```