JEE

Projet Jeu 4X

À l'attention de M.Mohamed Lamine Ben

BEZET Camille

PAVOT Fiona

BAIVEL Alexandra

MITSAKIS David

ING2 GSIA2

Le 10 Janvier 2025

Sommaire

Introduction

Les jeux 4X, caractérisés par leurs axes fondamentaux d'eXploration, d'eXpansion, d'eXploitation et d'eXtermination, sont des piliers du genre des jeux de stratégie. Ce projet s'inscrit dans le cadre de notre formation en génie logiciel, et vise à développer un jeu 4X multijoueur en utilisant les technologies JEE (Servlets et JSP) tout en respectant l'architecture MVC.

L'objectif principal est de consolider les connaissances acquises en cours, tout en explorant des concepts essentiels du développement web et des applications côté serveur. Grâce à la mise en œuvre des fonctionnalités spécifiques à ce type de jeu, telles que la gestion des joueurs, les mécanismes de jeu au tour par tour, et la synchronisation des actions entre utilisateurs, ce projet offre une opportunité concrète de mettre en pratique des compétences techniques variées.

Ce rapport détaille les différentes étapes du projet, de la conception à la mise en œuvre, en passant par les défis rencontrés et les solutions apportées.

Code source

Le code source du projet se trouve sur le GitHub suivant : https://github.com/abaivel/Projet_JEE.git.

Architecture du projet

L'application est codée en Java 21 déployée sur un conteneur Tomcat 11.

Base de données

L'architecture de la base de données du projet repose sur le framework JPA (Java Persistence API) pour gérer la persistance des données, en s'appuyant sur MySQL comme système de gestion de base de données relationnelle. Les entités principales du projet, *Joueur*, *Partie* et *JoueurPartie*, sont mappées aux tables correspondantes dans la base de données. Chaque classe d'entité est annotée avec @Entity, et les relations entre les entités sont définies à

l'aide d'annotations comme @OneToOne et @JoinColumn, permettant de gérer les associations et les clés étrangères. Par exemple, la classe *JoueurPartie* représente une table intermédiaire qui relie un joueur à une partie tout en stockant des informations supplémentaires comme le score (voir Annexe 1).

Le fichier de configuration persistence.xml joue un rôle central en spécifiant les paramètres de connexion à la base de données MySQL, tels que le driver JDBC, l'URL de connexion, les identifiants d'accès, et le mode de génération des schémas. Ce dernier, défini avec la propriété jakarta.persistence.schema-generation.database.action, permet la création automatique des tables lors de l'exécution. L'environnement MySQL est supervisé via l'outil MySQL Workbench, qui facilite la visualisation, la gestion et l'exploration des schémas générés. Le gestionnaire de persistance est implémenté dans les classes *JoueurDAO*, *JoueurPartieDAO* et *PartieDAO*, selon quelle table est utilisée dans les requêtes.

Structure MVC

La logique applicative et les DTO sont placés dans un module "modèle" et "service" . Un modèle est une classe et définit les objets du jeu tels que les soldats, les joueurs, le plateau de jeu, etc. Les services assurent la logique du jeu en faisant appel aux classes du modèle et exposent des fonctionnalités aux servlets. Les Servlets reçoivent des requêtes envoyées par les pages JSP et réagissent en appelant les fonctions des services concernés en backend. Et enfin les pages JSP assurent l'interaction avec l'utilisateur et redirigent vers les Servlet correspondantes aux actions des joueurs (voir Annexe 4).

Objets et Héritages du modèle

PartieDTO représente la partie en cours, elle contient une *Carte* et une liste de *JoueurDTO*. Un *JoueurDTO* de cette liste est également dans l'attribut *joueurTour* de la partie qui détermine lequel peut jouer.

JoueurDTO est la classe qui modélise l'utilisateur. Il possède des *Soldats*, des *Villes*, des *points de production*, un *nombre de combats gagnés* et un *nombre de combats perdus*. Enfin, ils sont identifiés à travers le code par leur attribut *login*, qui est unique parmi tous les utilisateurs enregistrés dans la base de données.

Carte contient une matrice de *Tuiles* 10 par 10 qui fournit les fonctionnalités de détection des différents objets présents sur le plateau de jeu. Elle est générée aléatoirement au début de la partie.

Tuile contient un *Soldat* et un *Element*. Ces deux attributs peuvent être vides si la case correspondante est vide. Les *Tuiles* sont identifiées par leurs coordonnées sur la *Carte*.

Element est la classe mère des éléments de décors du jeu, tels que, les *Villes*, les *Forets* et les *Montagnes*. Les *Forets* ont un nombre de *points de production* aléatoire fixé lors de la génération de la *Carte*. Les *Villes* ont un propriétaire *JoueurDTO* (cet attribut peut être null) ainsi que des *points de défense* et *de production* définis aléatoirement lors de la génération de la *Carte*.

Un *Soldat* a un propriétaire *JoueurDTO*, des *points de défense*, des *points de défense maximum* et un attribut booléen *canPlay*.

Un *Élément* ou un *Soldat* est accessible par la Tuile sur laquelle il est. Plus particulièrement, un *Soldat* ou une *Ville* est accessible par les attributs *villes* et *soldats* de leur propriétaire.

Fonctionnalités développées

Soldats

Sélection

Les icônes des soldats du joueur sont entourées de vert pour les différencier des soldats ennemis. Lorsque le joueur sélectionne le soldat qui fera une action, son contour devient bleu. Les coordonnées du soldat sur le plateau de jeu (*Carte*) sont alors récupérées par une fonction en JavaScript et seront données aux fonctions d'appel des servlets du javascript.

Chaque soldat ne peut faire qu'une action par tour. Les soldats ont un attribut booléen *canPlay* qui permet de contrôler s'ils ont déjà fait une action. Si *canPlay* est à TRUE les soldats sont sélectionnables et pourront faire une des actions décrites ci-dessous.

Mouvements

Des flèches directionnelles font une requête de mouvement à la servlet /move qui va appeler la fonction moveTuile de "CarteService". Cette fonction va vérifier si le *Soldat* peut jouer et s'il peut aller dans la direction souhaitée avant d'autoriser le mouvement. Ensuite, s'il

entre en contact avec une ville ou un soldat ennemi, elle retourne la Tuile correspondante. Si la *Tuile* destination est libre, elle change les coordonnées du *Soldat*, retire le *Soldat* de la *Tuile* de départ, l'ajoute dans la *Tuile* d'arrivée, et elle retourne *null*. Dans ce cas là, /move renvoie juste le nouvel état de la partie au client.

Dans le cas où /move reçoit une *Tuile*, alors elle renverra une réponse avec le code 301 et des données en JSON. Le code en JavaScript dans game.jsp reçoit cette réponse et redirige vers la page indiquée dans le JSON. Dans le cas présent, la page redirige vers /combat avec une requête GET ayant pour paramètres les coordonnées des deux combattants.

Soigner

Un bouton "Se soigner" fait une requête à la servlet /soigner qui appelle la fonction soignerSoldat de "CarteService". Cette fonction ajoute 10 points de défense au *Soldat*. La méthode setPoint_defense() de *Soldat* interdit les points de défense du soldat de dépasser ses points de défense maximum, Donc, si un *Soldat* a perdu moins de 10 points de défense et se soigne, il sera simplement au maximum de sa santé. S'il est déjà au maximum de sa vie, le *Soldat* ne peut pas se soigner et peut alors effectuer une autre action.

Fourrage des forêts

Si le soldat se trouve sur une *Tuile* avec une *Foret*, un nouveau bouton "Fourrager" apparaît. "Fourrager" fait une requête à la servlet /fourrage qui appelle la fonction fourrage de "CarteService". La fonction vérifie si la *Tuile* du *Soldat* contient également une *Forêt*, si oui elle ajoute le nombre de *points de production* de la *Foret* au *JoueurDTO*.

Chacune des fonctions des services appelée par ces actions affectera FALSE à l'attribut canPlay du soldat joué une fois celle-ci effectuée.

Recrutement

Le bouton "Recruter un soldat : 15 points de production" fait une requête à la servlet /recruter qui appelle la fonction addSoldat de "CarteService". Celle-ci vérifie si le Joueur a assez de *points de production* pour recruter un *Soldat*. Si oui, il va appeler la fonction addSoldat de *Carte* pour créer un nouveau *Soldat* sur le plateau de jeu sur une *Tuile* aléatoire.

Finir le tour

Un bouton “finir le tour” fait une requête à la servlet /pass qui appelle la fonction passerTour de “CarteService”. Celle-ci fera appel à la méthode resetSoldats() de *JoueurDTO* pour affecter TRUE à l’attribut *canPlay* de chacun des soldats du joueur. Elle appelle la méthode TourSuivant() de *Partie* pour passer la main au prochain joueur.

Combat

Une fois que la requête GET a été envoyée par le JSP game.jsp, le combat est déclenché entre les deux combattants envoyés dans la requête GET. On redirige l’utilisateur vers une interface de combat. Les dégâts seront calculés par des jets aléatoires de dés entre 1 et 6, chacun son tour, jusqu’à ce qu’un des deux arrive à 0 ou moins. Si un *Soldat* perd un combat, il disparaît de la carte et est remplacé par le *Soldat* vainqueur sur sa position. Si une *Ville* perd un combat, elle prend le *JoueurDTO*, propriétaire du *Soldat*, comme nouveau propriétaire.

Gestion du score

Le score est calculé dans la méthode getScore() de la classe *JoueurDTO* à partir de ses attributs. On additionne 5 points par combat gagné, 10 points par ville conquise et on soustrait 6 points par combat perdu.

Carte aléatoire

Le constructeur de la classe *Carte* appelle une de ses méthodes appelée setGrilleAleatoire(). Cette méthode crée des coordonnés aléatoires entre 0 et 9, si la *Tuile* choisie contient déjà un élément, elle relance les coordonnées aléatoires jusqu’à en trouver une libre et y place un *Element*. La méthode effectue cette opération pour 5 *Montagnes*, 10 *Forets* et 5 *Villes*.

Affichage des scores

L’affichage des scores à la fin d’une partie, et l’affichage des scores des anciennes parties du joueur reposent sur une architecture basée sur le modèle MVC (Modèle-Vue-Contrôleur). Le RecapScoresController et RecapScoresJoueurController,

implémentés comme des servlets, traitent les requêtes HTTP entrantes et interagissent avec le service métier ScoreService. Ces derniers utilisent le DAO *JoueurPartieDAO* pour récupérer les scores depuis la base de données via une requête JPQL.

Le DAO (Data Access Object) est une couche d'abstraction qui encapsule la logique d'accès aux données. Dans ce projet, *JoueurPartieDAO* fournit des méthodes spécifiques pour interagir avec la base de données, telles que la récupération des scores triés par ordre décroissant ou des scores associés à un joueur spécifique. Cette approche permet de centraliser et de simplifier les opérations liées à la persistance des données, tout en respectant le principe de séparation des responsabilités.

Une fois les données extraites par le DAO et traitées par le service, elles sont transmises à la vue recapScores.jsp ou recapScoresJoueur.jsp, les pages JSP responsables de l'affichage. Ces vues génèrent dynamiquement un tableau HTML listant les joueurs ou les parties ainsi que les scores associés grâce à une boucle, et un style visuel personnalisé en CSS. Si aucun score n'est disponible, un message explicite est affiché à l'utilisateur.

Enfin, un bouton de navigation permet à l'utilisateur de revenir à l'accueil.

Démarches et outils utilisés

Démarches

Gestion d'un projet en équipe

Pour assurer une gestion efficace et collaborative du projet, nous avons mis en place une méthodologie structurée centrée sur la communication et le contrôle de la qualité du code. Tout d'abord, nous avons utilisé un dépôt Github avec des règles strictes de gestion des branches. Les contributions au projet passent par des pull-requests, et aucune modification n'est directement poussée sur la branche main sans qu'un membre de l'équipe ait validé la pull request. Cette démarche garantit une revue systématique du code, réduit les risques d'introduction de bugs et assure une meilleure compréhension collective des modifications.

La communication a également été un élément clé de notre organisation. Nous avons utilisé Discord comme plateforme principale pour échanger des informations, partager des ressources, et nous entraider en temps réel sur les différentes problématiques rencontrées. En complément, nous avons organisé des réunions en visioconférence pour discuter de manière plus approfondie des aspects techniques et expliquer nos implémentations de code respectives. Ces échanges en visio ont permis de clarifier des points complexes, d'aligner nos approches et de renforcer notre cohésion d'équipe.

Outils

Inscription et connexion avec un pseudo unique

La page login.jsp (voir Annexe 3) est la première que l'utilisateur voit, celle-ci permet à la fois de créer un nouveau compte et de s'y connecter. Une fois le formulaire rempli, celui-ci envoie une requête à la servlet /accueil. La servlet vérifie que la réponse envoyée est valide et non vide, sinon elle renvoie un code d'erreur. Une fois la requête validée, elle appelle "JoueurService" (voir Annexe 5) qui va vérifier si le *login* du joueur est dans la base de données grâce à un *JoueurDAO*. Si le *login* n'existe pas dans la base de données, nous créons un compte avec le login et le mot de passe donné. Si le *login* existe, nous vérifions le mot de passe. S'il correspond, nous connectons l'utilisateur, sinon nous envoyons un message d'erreur sur la page de login. En cas de connexion réussie ou de nouveau compte créé, nous créons un nouveau *JoueurDTO* pour gérer la session de l'utilisateur en backend et nous l'ajoutons à la partie active du serveur s'il n'y était pas déjà. (voir Annexe 6)

Multijoueurs

Pour le joueur en train de jouer, le plateau de jeu est rafraîchi à chaque appel serveur que ce dernier effectue. Pour les joueurs en train d'attendre leur tour, le plateau de jeu est rafraîchi par un polling toutes les 5 secondes.

Difficultés rencontrées

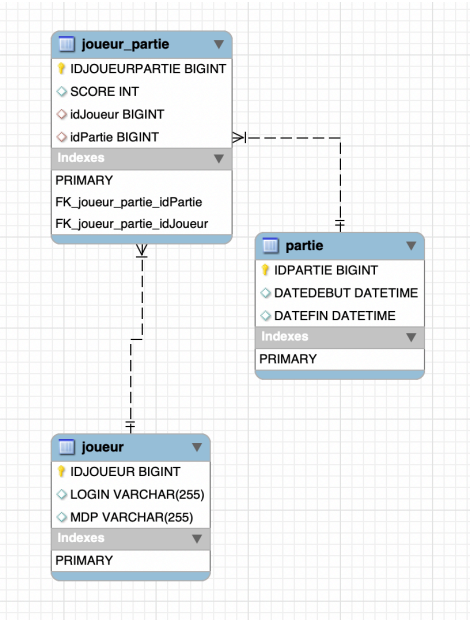
Gestion de la base de données

La gestion de la base de données a représenté une difficulté importante dans ce projet, tant sur le plan technique que conceptuel. Tout d'abord, il fallait réussir à maîtriser les JPA et la bibliothèque Persistance, pour pouvoir enregistrer des données. Il a ensuite fallu concevoir une structure de données adaptée aux besoins de ce jeu, c'est-à-dire capable d'enregistrer les joueurs, les parties et les scores de chaque joueur lors d'une partie.

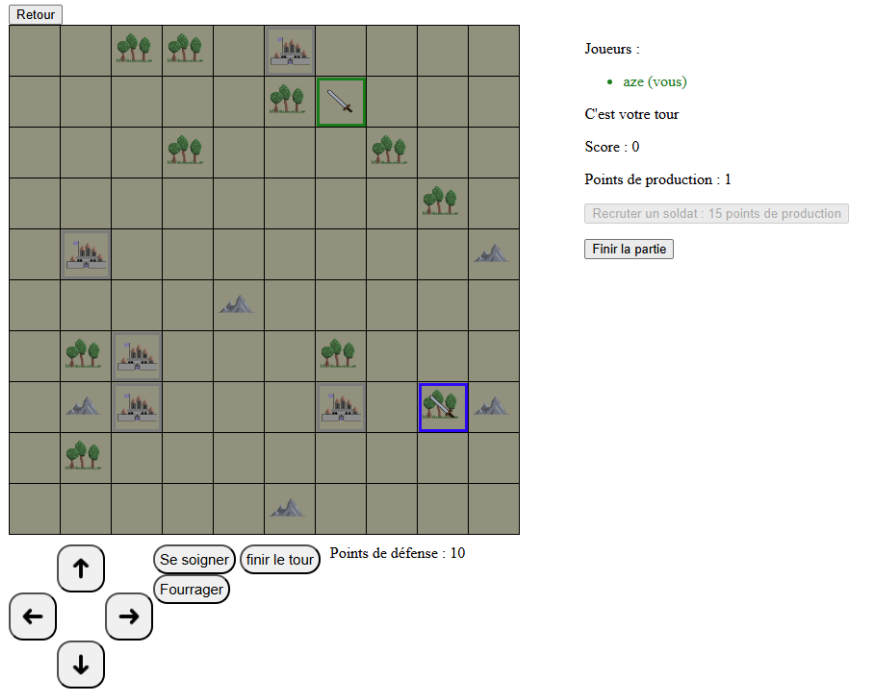
Conclusion

Ce projet de développement d'un jeu 4X multijoueur en JEE a constitué une expérience formatrice et enrichissante, tant sur le plan technique que collaboratif. En appliquant les principes fondamentaux de l'architecture MVC et en exploitant des technologies comme JPA, Servlets et JSP, nous avons pu concevoir et mettre en œuvre une application web robuste, respectant les exigences du genre des jeux de stratégie. L'intégration de fonctionnalités complexes telles que la gestion des scores, la synchronisation des joueurs en temps réel, et les interactions dynamiques avec la base de données ont permis de mettre en pratique une grande variété de compétences.

Annexe



Annexe 1. Diagramme de classe de la BDD



Annexe 2. Page du jeu

Annexe 3. Page de login

```
function callServlet(data, servlet){
    const apiUrl = '${pageContext.request.contextPath}'+servlet;
    const requestOptions = {
        method: 'POST',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
        },
        body: data.toString(),
    };

    fetch(apiUrl, requestOptions)
        .then(response => {
            if (response.status === 301){
                return response.json();
            }else if (!response.ok) {
                throw new Error('Network response was not ok');
                //afficher une erreur sur la page
            }
            else{
                location.reload();
            }
        })
        .then(data => {
            if (data.redirect) {
                window.location.href = data.redirect; // Redirection vers l'URL renvoyée
            }
        })
        .catch(err => {
            console.log(err.message);
        });

}
```

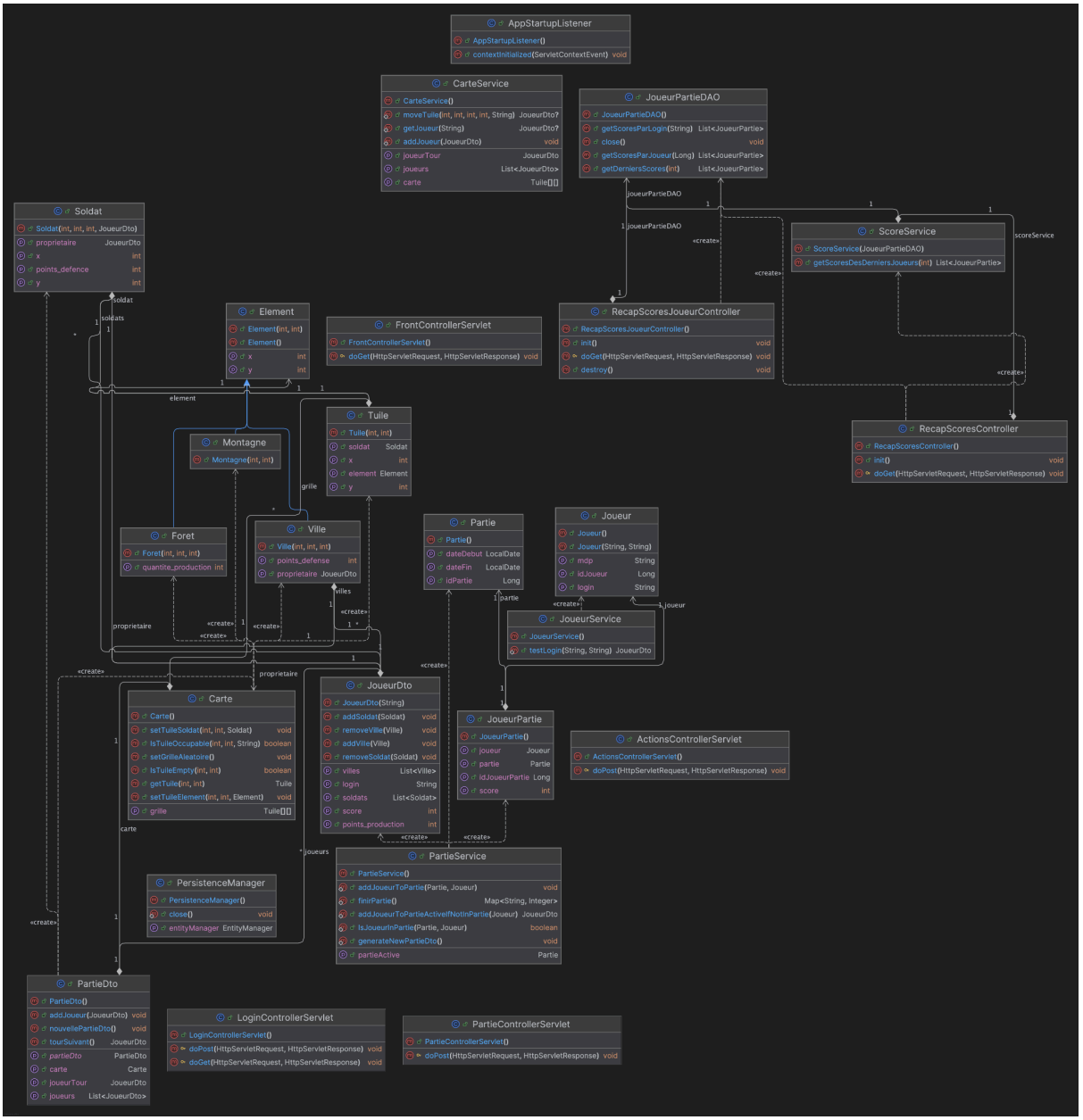
Annexe 4. Fonction d'appel de Servlet de game.jsp

```
1 usage  Alexandra Baivel
public JoueurDto testLogin(String login, String password) {
    boolean result = false;
    Joueur joueurconnecte = null;
    JoueurDto joueurdto = null;
    joueurconnecte =  joueurDAO.getJoueurByLogin(login);
    if (joueurconnecte == null) {
        joueurconnecte = joueurDAO.createJoueur(login, password);
        result = true;
    }else{
        result = joueurconnecte.getMdp().equals(password);
    }
    if (result){
        joueurdto = partieService.addJoueurToPartieActiveIfNotInPartie(joueurconnecte);
    }
    return joueurdto;
}
```

Annexe 5. Méthode de création et connexion des comptes joueurs.

```
1 usage  Alexandra Baivel
public void addJoueur(JoueurDto joueur){
    if (joueurs.isEmpty()) {
        joueurTour = joueur;
    }
    joueurs.add(joueur);
    int xRandom = 0;
    int yRandom = 0;
    do {
        xRandom = (int) (Math.random() * 10);
        yRandom = (int) (Math.random() * 10);
    }while (!carte.IsTuileEmpty(xRandom, yRandom));
    carte.setTuileSoldat(xRandom,yRandom,new Soldat(xRandom,yRandom,  points_defence: 20, joueur));
}
```

Annexe 6. Méthode d'ajout d'un joueur dans une PartieDTO.



Annexe 7. Diagramme de classe du projet