Sup Galilée

Troisième année
Motifs et conceptions
Rapport

Projet de modélisation - Patron MVC

Élèves :
Sidane Alp
Hamza Salah
Wissal Bendidi

Enseignant :
G. Santini

25 janvier 2025

Table des matières

1 Introduction

Dans le cadre de notre projet de programmation orientée objet, nous avons développé une application complète de gestion de zoo en Java.

Notre équipe de trois étudiants s'est organisée selon une répartition des responsabilités qui nous a semblé optimale pour mener à bien ce projet. Nous avons confié à chacun un aspect crucial du développement :

L'architecture MVC et le développement du contrôleur(implémentation de ZooController,...) ont été pris en charge par Sidane ALP.

La modélisation et l'implémentation des classes d'animaux et d'enclos ont été confiées à Wissal BENDIDI.

La gestion des employés (gestion des droits, interface de connexion,...) et le système financier ont été développés par Hamza SALAH.

Veuillez retrouver comment compiler et lancer le programme dans l'annexe tout à la fin du rapport.

2 Présentation du projet

2.1 Objectifs et contraintes

Le développement de notre application s'est effectué dans un cadre contraint que nous nous sommes efforcés de respecter scrupuleusement :

- **Contrainte technologique** : En nous limitant aux librairies standards Java, nous avons dû développer nos propres solutions pour la gestion des données et l'interface utilisateur.
- **Interface utilisateur** : Face à la contrainte d'une interface en mode console, nous avons conçu une expérience utilisateur intuitive malgré les limitations du terminal. Nous avons notamment développé un système de menus hiérarchiques et une gestion robuste des entrées utilisateur.
- **Architecture** : L'application du pattern MVC et des principes SOLID nous a conduits à définir précisément les responsabilités de chaque composant et leurs interactions.
- **Documentation** : La nécessité d'une documentation exhaustive via Javadoc nous a amenés à maintenir une documentation rigoureuse tout au long du développement.

2.2 Choix du sujet et originalité

Le gestionnaire de zoo nous est apparu comme un choix particulièrement pertinent pour plusieurs raisons :

- **Richesse du domaine métier** : Un zoo implique de nombreuses entités avec des comportements et des interactions complexes, ce qui nous a permis d'explorer en profondeur les concepts de la programmation orientée objet.
- **Hiérarchie naturelle** : Les différentes espèces d'animaux nous ont fourni une base idéale pour mettre en pratique l'héritage et le polymorphisme. Nous avons ainsi implémenté des comportements spécifiques à chaque espèce tout en maintenant une structure cohérente.

— **Évolutivité** : L'architecture que nous avons conçue permet facilement l'ajout de nouvelles espèces d'animaux ou de nouveaux types d'employés, démontrant ainsi la flexibilité de notre conception.

2.3 Fonctionnalités principales

Dans notre application, nous avons implémenté trois profils utilisateurs distincts, chacun disposant de fonctionnalités spécifiques à son rôle :

— **Gestion des animaux** : Nous avons développé un système complet incluant :
  — Un suivi dynamique de la santé et de la faim de chaque animal
  — Des comportements spécifiques par espèce (chasse pour les lions, vol pour les oiseaux)
  — Un système de nourrissage et de soins nécessitant des permissions appropriées
— **Gestion des enclos** : Nous avons implémenté :
  — Un système de propreté évoluant en fonction du nombre d'animaux
  — Une gestion de la capacité maximale
  — Des assignations spécifiques aux agents d'entretien
— **Gestion du personnel** : Nous avons mis en place :
  — Un système de spécialisation des soigneurs par espèce
  — Une attribution dynamique des enclos aux agents d'entretien
  — Une gestion des salaires et des compétences

3 Architecture globale

3.1 Pattern MVC

Nous avons structuré notre application selon une architecture MVC rigoureuse, en portant une attention particulière à la séparation des responsabilités.

Le Modèle représente le cœur métier de notre application. Nous l'avons organisé en plusieurs sous-packages :

— `animal/` : Gestion des différentes espèces et leurs comportements
— `employee/` : Gestion du personnel et leurs spécialisations
— `enclos/` : Gestion des espaces et leur maintenance
— `financial/` : Suivi budgétaire et transactions
— `user/` : Gestion des droits et authentification

La Vue est implémentée via une interface console robuste. Voici un exemple simplifié de son fonctionnement :

```
public class ConsoleView {
    public void displayMainMenu(Role userRole) {
        System.out.println("=== " + zoo.getName() + " ===");
        displayMenuOptions(userRole);
    }
}
```

Cette interface assure une expérience utilisateur cohérente malgré les limitations du terminal. **Le Contrôleur** orchestre les interactions entre la vue et le modèle. Il gère notamment :

— La validation des actions utilisateur

— La coordination des mises à jour d'état
— La gestion des exceptions métier
— La synchronisation des différents composants

3.2 Organisation des packages

Notre architecture reflète une séparation claire des préoccupations :

```
fr.zoomanager/
              controller/
              model/
              view/
              Main.java
```

Cette organisation nous apporte plusieurs avantages :

— **Maintenance facilitée** : Chaque package a une responsabilité unique et bien définie, permettant des modifications ciblées sans effets de bord
— **Développement parallèle** : Notre équipe a pu travailler simultanément sur différents aspects du projet sans conflits
— **Évolutivité** : L'ajout de nouvelles fonctionnalités peut se faire en minimisant l'impact sur l'existant

Cette architecture nous a également permis de gérer efficacement les dépendances entre les composants. Le contrôleur agit comme une façade, protégeant le modèle des accès directs et assurant la cohérence des données. La vue, quant à elle, reste totalement découplée du modèle, ne recevant que les données nécessaires à l'affichage via le contrôleur.

En termes de communication, nous avons mis en place un système d'événements permettant aux différents composants d'être notifiés des changements pertinents sans créer de couplage fort. Par exemple, lorsqu'un animal voit son état de santé changer, le modèle émet un événement qui est capté par le contrôleur, qui décide ensuite si la vue doit être mise à jour.

4 Application des principes SOLID

Dans notre projet, nous avons appliqué rigoureusement les principes SOLID pour garantir la qualité et la maintenabilité de notre code.

4.1 Single Responsibility Principle

Le principe de responsabilité unique a guidé la conception de nos classes. Prenons un exemple concret :

```
public abstract class Animal {
    private HealthManager healthManager;
    private FeedingManager feedingManager;
    public void updateState() {
        healthManager.update();
        feedingManager.update();
    }
}
```

Plutôt que de gérer directement la santé et l'alimentation, la classe Animal délègue ces responsabilités à des gestionnaires spécialisés. Cette approche nous a permis de :

— Simplifier la maintenance en isolant les changements
— Faciliter les tests en permettant le mock des gestionnaires
— Permettre l'évolution indépendante des différents aspects

4.2 Open/Closed Principle

Nous avons conçu nos classes pour être extensibles sans modification. Un exemple éloquent est notre système de comportements :

```
public interface MovementBehavior {
    void move();
}
public abstract class Animal {
    private MovementBehavior movementBehavior;
}
```

Cette conception nous permet d'ajouter facilement de nouveaux comportements sans modifier les classes existantes. Par exemple, nous pouvons ajouter un nouveau type de déplacement simplement en implémentant l'interface MovementBehavior.

4.3 Liskov Substitution Principle

Le principe de substitution de Liskov a été particulièrement important dans notre hiérarchie d'animaux. Nous avons veillé à ce que chaque sous-classe respecte le contrat de sa classe parente. Par exemple, tous nos animaux peuvent être gérés de manière polymorphe dans les enclos :

— Les méthodes communes (`feed()`, `updateState()`) ont un comportement cohérent
— Les spécialisations (`hunt()` pour les lions, `fly()` pour les oiseaux) n'altèrent pas le comportement de base
— Les pré et post-conditions sont respectées dans toute la hiérarchie

4.4 Interface Segregation Principle

Au lieu de créer des interfaces monolithiques, nous avons défini des interfaces spécifiques pour chaque comportement :

```
public interface Feedable {
    void feed();
    double getHunger();
}
public interface Cleanable {
    void clean();
    double getCleanliness();
}
```

Cette approche permet à chaque classe de n'implémenter que les comportements dont elle a réellement besoin, évitant ainsi les dépendances inutiles.

4.5 Dependency Inversion Principle

L'injection de dépendances est au cœur de notre architecture. Notre contrôleur principal en est un excellent exemple :

```java
public class ZooController {
    private final Zoo zoo;
    private final ConsoleView view;
    private final UserManager userManager;

    public ZooController(Zoo zoo, ConsoleView view,
                         UserManager userManager) {
        this.zoo = zoo;
        this.view = view;
        this.userManager = userManager;
    }
}
```

Les avantages de cette approche sont multiples :
— Découplage fort entre les composants
— Facilité de tests grâce à la possibilité de mock
— Flexibilité pour changer d'implémentation

4.6 Impact sur la qualité du code

L'application systématique de ces principes nous a permis d'obtenir :
— Un code plus maintenable avec des responsabilités clairement définies
— Une meilleure testabilité grâce au découplage des composants
— Une évolutivité facilitée par la conception extensible
— Une réutilisabilité accrue des différents composants

5 Patterns de conception utilisés

5.1 Pattern Observer

Pour gérer les mises à jour d'état dans notre zoo, nous avons implémenté le pattern Observer. Ce choix s'est révélé particulièrement pertinent pour :
— Notifier automatiquement la vue des changements d'état des animaux
— Alerter le personnel soignant en cas de problème de santé
— Maintenir la cohérence entre les différents composants du système

```java
public class Zoo implements ZooSubject {
    private List<ZooObserver> observers = new ArrayList<>();
    public void updateState() {
        notifyObservers(new StateUpdateEvent());
    }
}
```

5.2 Pattern Strategy

Le pattern Strategy nous a permis de gérer efficacement les différents comportements des entités du zoo. Par exemple, pour le nettoyage des enclos :
— Nettoyage standard pour l'entretien quotidien
— Nettoyage approfondi pour les enclos à forte occupation
— Désinfection complète en cas de maladie

L'implémentation de ce pattern nous offre une grande flexibilité dans la gestion des comportements :

```
public interface CleaningStrategy {
    void clean(Enclos enclos);
}
public class AgentMenage {
    private CleaningStrategy strategy;
    public void cleanEnclos(Enclos enclos) {
        strategy.clean(enclos);
    }
}
```

6 Interface utilisateur et gestion des erreurs

6.1 Architecture de la vue console

Face au défi de créer une interface utilisateur en mode console à la fois intuitive et robuste, nous avons développé une approche structurée :

- **Menus hiérarchiques** : Navigation claire entre les différentes fonctionnalités
- **Affichage contextuel** : Adaptation des options selon le rôle de l'utilisateur
- **Retours utilisateur** : Messages clairs pour chaque action effectuée

Voici un aperçu simplifié de notre système de menus :

```
public class ConsoleView {
    public void displayMenu(Role userRole) {
        System.out.println("\n=== Menu Principal ===");
        displayOptionsForRole(userRole);
        handleUserInput();
        }
}
```

6.2 Gestion des erreurs

Nous avons mis en place un système complet de gestion des erreurs qui couvre trois aspects principaux :

1. Validation des entrées utilisateur

- Vérification du format des données
- Contrôle des valeurs autorisées
- Messages d'erreur explicites en cas de saisie incorrecte

2. Exceptions métier

- Hiérarchie d'exceptions personnalisées
- Traitement spécifique selon le type d'erreur
- Journalisation des erreurs critiques

3. Récupération des erreurs Nous avons implémenté un système de reprise sur erreur :

```
public class ErrorHandler {
    public void handle(Exception e) {
        logError(e);
        displayUserMessage(e);
        suggestRecoveryAction(e);
```

```
    }
}
```

6.3 Exemples de scénarios

6.3.1 Scénario de succès

```
=== Menu Principal ===
1. Gérer les animaux
2. Gérer les enclos
3. Quitter

Votre choix : 1
Action effectuée avec succès !
```

6.3.2 Scénario d'erreur

```
=== Menu Principal ===
1. Gérer les animaux
2. Gérer les enclos
3. Quitter

Votre choix : abc
Erreur : Veuillez entrer un nombre entre 1 et 3
```

7 Difficultés rencontrées et solutions

7.1 Challenges techniques majeurs

Au cours du développement, nous avons été confrontés à plusieurs défis significatifs :

1. Gestion des états dynamiques

— **Problème** : La mise à jour régulière de l'état des animaux et des enclos créait des risques de conflits et d'incohérences

— **Solution** : Nous avons développé un système de gestion d'état centralisé :

```
public class StateManager {
    public void startStateUpdates() {
        scheduler.scheduleAtFixedRate(
            this::updateAllStates,
            0, 1, TimeUnit.MINUTES
        );
    }
}
```

2. Synchronisation des actions

— **Problème** : Plusieurs employés pouvaient tenter d'interagir simultanément avec le même enclos ou animal

— **Solution** : Mise en place d'un système de verrouillage des ressources :

— Verrouillage temporaire pendant les opérations critiques

— File d'attente pour les actions concurrentes

— Timeout pour éviter les blocages

7.2 Défis architecturaux

Nous avons rencontré plusieurs challenges dans la conception :

1. Séparation des responsabilités

— **Difficulté** : Définir clairement les frontières entre les différents composants

— **Solution** :

  — Création d'interfaces claires entre les modules

  — Définition précise des responsabilités de chaque classe

  — Revues de code régulières pour maintenir la cohérence

2. Évolutivité du système

— **Difficulté** : Permettre l'ajout de nouvelles fonctionnalités sans modifier le code existant

— **Solution** :

  — Architecture modulaire basée sur les interfaces

  — Utilisation intensive des design patterns

  — Documentation exhaustive des points d'extension

8 Conclusion

8.1 Bilan

Au terme de ce projet, nous avons réussi à développer une application complète de gestion d'un zoo, démontrant notre maîtrise des concepts avancés de programmation orientée objet. La répartition des tâches au sein de notre équipe s'est révélée efficace, chacun ayant pu se concentrer sur des aspects spécifiques du projet tout en maintenant une cohérence globale dans la gestion de ce zoo numérique.

8.2 Perspectives d'évolution

Pour l'avenir, nous envisageons plusieurs améliorations :

— **Interface graphique** : Développement d'une interface JavaFX

— **Base de données** : Intégration d'une persistance des données

— **API REST** : Exposition des fonctionnalités via une API

9 Annexes

9.1 Guide d'utilisation

Pour installer et exécuter l'application :

```
javac -d bin -sourcepath src src/fr/zoomanager/Main.java
java -cp bin fr.zoomanager.Main
```