Conception orientée objet

Miniprojet 1 - Canard Fighter Simulator

Philippe Roussille

3iL INGENIEURS
</les sciences informatiques>

1 Objectifs

1. Implémenter un simulateur de combat pour explorer les concepts avancés de la programmation orientée objet.
2. Comprendre et utiliser l'héritage, le polymorphisme et la liaison dynamique.
3. Développer une application modulaire et extensible en Java.

2 Introduction au Canard Fighter Simulator

Dans ce projet, vous créerez un jeu de combat mettant en scène des "Canards Légendaires" disposant de types, forces, faiblesses, attaques et capacités spéciales. Ce simulateur servira de base pour appliquer et approfondir vos connaissances en conception orientée objet.

2.1 Spécifications de base

- Chaque canard a un **nom**, un **type**, des **points de vie (PV)**, des **points d'attaque (PA)**, et des **capacities spéciales**.
- Les types incluent : Eau, Feu, Glace, Vent.
- Les relations de force/faiblesse sont définies comme suit :
  - Eau > Feu
  - Feu > Glace
  - Glace > Vent
  - Vent > Eau
- Chaque canard peut attaquer un autre, infligeant des dégâts calculés en fonction des types.
- Chaque type de canard dispose d'une capacité spéciale activable une fois par bataille.

3 Instructions du projet

3.1 Étape 1 : Modélisation des classes

1. Créez une classe de base `Canard` avec les accesseurs suivants :
   — `String getNom()`
   — `int getPointsDeVie()`
   — `int getPointsAttaque()`
   — `TypeCanard getType` (enum des types disponibles).
2. Ajoutez les méthodes suivantes à la classe `Canard` :
   — `attaquer(Canard autreCanard)` : inflige des dégâts en fonction du type.
   — `subirDegats(int degats)` : réduit les PV du canard.
   — `boolean estKO()` : retourne vrai si le canard est hors combat (PV 0).
3. Implémentez des classes filles :
   — `CanardEau`
   — `CanardFeu`
   — `CanardGlace`
   — `CanardVent`

Chaque classe fille peut redéfinir la capacité spéciale via une méthode `activerCapaciteSpeciale()`.

3.2 Étape 2 : Gestion des forces/faiblesses

1. Définissez un enum `TypeCanard` pour représenter les types.
2. Implémentez une méthode statique dans `TypeCanard` :
   — `double getMultiplicateur(TypeCanard attaquant, TypeCanard cible)` : retourne 1.5 (fort), 0.5 (faible), ou 1.0 (neutre).
3. Modifiez la méthode `attaquer` pour intégrer ce multiplicateur.

3.3 Étape 4 : Capacités spéciales

Implémentez des capacités spécifiques dans chaque classe fille. Par exemple :

— `CanardEau` : régénère 20 PV.
— `CanardFeu` : inflige des dégâts supplémentaires pendant un tour.
— `CanardGlace` : gèle un adversaire, lui faisant perdre un tour.
— `CanardVent` : augmente temporairement sa vitesse d'attaque (2x attaque, 3x attaque, etc.).

3.4 Étape 3 : Interface utilisateur minimaliste

Créez une classe `Main` qui :

1. Utilise un `Scanner` pour permettre à l'utilisateur de créer des canards.
2. Propose un menu simple :
   — Créer des canards.
   — Lancer une bataille.
   — Quitter.
3. Permet de simuler un combat tour par tour entre deux canards.

3.5 Étape 4 : Tests unitaires

Ajoutez des tests unitaires pour :

— Vérifier les interactions entre les types.
— Tester les méthodes principales de la classe `Canard`.

4 Questionnements sur la modélisation

Ajoutez les réponses aux questions suivantes dans le fichier `README.md` du dépôt :

1. **Quelles classes pourraient être abstraites ?**
2. **Quels comportements communs pourraient être définis dans une interface ?**
3. **Comment représenter un changement de statut (par exemple, brûlé ou paralysé) dans la modélisation ?**
4. **Quels seraient les avantages d'utiliser une classe ou une interface supplémentaire pour gérer les capacités spéciales ?**
5. **Quels défis sont associés à l'extensibilité de votre modèle pour ajouter de nouveaux types de canards ou de nouvelles capacités ?**

5 Au choix : Fonctionnalités avancées

Réalisez au moins une de ces fonctionnalités. Pour chacune, indiquez les modifications et la réflexion conceptuelle que vous avez entreprises.

5.1 Gestion avancée des combats

1. **Effets de statut** : Ajoutez des effets persistants comme "brûlé" (perd des PV à chaque tour), "gelé" (ne peut pas agir pendant 2 tours), ou "paralysé" (peut agir 50% du temps).
   — Implémentez une méthode `appliquerEffets()`.
2. **Points d'énergie (PE)** :
   — Limitez l'utilisation des attaques à l'aide d'une ressource PE.
   — Exemple : attaque = 5 PE, capacité spéciale = 15 PE.
3. **Attaques critiques** :
   — Ajoutez une probabilité (10%) d'attaques critiques infligeant 2x les dégâts.

5.2 Types et interactions supplémentaires

1. **Nouveaux types de canards** :
   — Ajoutez des types comme Électrique, Toxique ou Sol avec leurs forces/faiblesses.
   — Voici leurs exigences :
      — Canard Électrique
         — Avantage : neutralise les faiblesses (ignore les multiplicateurs de type lors d'une attaque).
      — Canard Toxique

  - Capacité spéciale : inflige un poison qui diminue les PV de l'adversaire de 5 PV par tour pendant 3 tours.
- Canard Sol
  - Résistant aux attaques Électriques et Feu.
  - Faible face aux attaques Eau.

Type	Fort contre	Faible contre
Eau	Feu, Sol	Glace, Vent
Feu	Glace, Toxique	Eau, Sol
Glace	Vent, Sol	Feu, Toxique
Vent	Eau, Toxique	Glace, Électrique
Électrique	Vent, Eau	Sol
Toxique	Glace, Eau	Feu, Sol
Sol	Feu, Électrique	Eau, Glace

2. **Attaques spéciales uniques** : Chaque type de canard peut disposer d'une attaque spéciale.
   - Canard Électrique : une attaque qui ignore les forces/faiblesses.
   - Canard Toxique : inflige un poison persistant.
   - Canard Sol : inflige des dégâts accrus (multiplicateur de 1.5) aux canards de type Vent et Glace.

5.3 Personnalisation et progression

1. **Création de canards personnalisés** :
   - Permettez au joueur de personnaliser nom, type, PV et PA dans des limites prédéfinies.
2. **Évolution des canards** :
   - Un canard peut gagner des statistiques ou débloquer de nouvelles capacités après une victoire.
3. **Objets utilisables** :
   - Intégrez des objets comme des potions pour restaurer PV/PE.

6 Livrables attendus

1. **Code source** :
   - Bien organisé, avec des classes séparées et documentées (JavaDoc).
   - Le dépôt doit être propre avec des commits réguliers et normalisés.
2. **Fichier README.md** :
   - Contient un diagramme UML des classes et une description des choix techniques.
   - Contient la liste des réalisations bonus, s'il y en a.
   -
3. **Fichier `HELPERS`** :
   - Créez un fichier `HELPERS` à la racine du dépôt avec la liste des noms des personnes qui vous ont aidés dans ce projet.
   - Chaque personne listée pourra recevoir un bonus sur sa moyenne en reconnaissance de son aide.

7 Exemple de départ : Classe `Main`

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Bienvenue dans Canard Fighter Simulator !");
        // Exemple minimaliste à compléter
        System.out.println("1. Créer un canard");
        System.out.println("2. Lancer une bataille");
        System.out.println("3. Quitter");
        scanner.close();
    }
}
```

Bonne chance !