1 Conception générale

Le projet *Ant Colony Rush* suit une approche *MVC* (*Modèle-Vue-Contrôleur*). Le code source est structuré de la façon suivante :

- Le **Modèle** contient toutes les classes représentant la logique et les données du jeu (`Terrain`, `Nid`, `Ressource`, `Abri`, `Fourmi`, `Crapaud`, etc.).
- La **Vue** comprend les classes d'interface utilisateur (par exemple `TerrainPanel`, `JeuFrame`, etc.), qui dessinent les éléments graphiques.
- Le **Contrôleur** est composé des classes comme `TerrainController`, `StartMenuController`, `GestionScore`, etc. qui gèrent les interactions et la synchronisation des éléments (clics souris, transitions d'états, fils d'exécution, etc.).

L'utilisation du modèle MVC garantit une séparation claire des responsabilités, simplifiant ainsi les développements futurs et améliorant la qualité globale du projet.

1.1 Threads Principaux

- `GestionnaireEnergie` : met à jour l'énergie des fourmis.
- `GestionnaireDeplacement` : synchronise les déplacements.
- `GestionnaireCrapaud` : gère le comportement du crapaud (déplacements, satiété, sommeil).
- `GestionScore` : surveille la création de nouvelles fourmis via le score.

2 Conception détaillée

- **Les structures de données** utilisées (principales classes, collections, variables pertinentes, etc.).
- **Les constantes du modèle** (valeurs fixes influençant l'algorithme ou le comportement).
- **L'algorithme abstrait** mis en place (sans recopier le code, mais en expliquant la logique).
- **Les conditions limites** à respecter (cas extrêmes ou hypothèses de cohérence).
- **Les interactions** avec les autres fonctionnalités (points de communication ou dépendances).

3 Fonctionnalité 1 : Génération aléatoire de la carte

3.1 Structures de données

— `Terrain` : classe centrale stockant la liste des `ObjetFixe` (`Nid`, `Abri`, `Ressource`...) et contenant la logique de génération initiale.

```
// Terrain.java (extrait)
public class Terrain {
    public static final int LARGEUR = 800;
    public static final int HAUTEUR = 800;
    private static ArrayList<ObjetFixe> elts;
    private ArrayList<Deplacement> expeditions;
    private Crapaud crapaud;

    public Terrain() {
        elts = new ArrayList<>();
        expeditions = new ArrayList<>();
        // Init du Nid, abris, ressources, crapaud, etc.
    }
    // ...
}
```

— `GenerationRessource` et `GenerationAbri` : classes utilitaires permettant de créer des instances de `Ressource` et d'`Abri` à des coordonnées aléatoires.
— `Random` : générateur de nombres aléatoires (`java.util.Random`).
— `ArrayList<ObjetFixe>` : collection des objets fixes placés sur la carte.

3.2 Constantes du modèle

— `Terrain.LARGEUR` et `Terrain.HAUTEUR` (800x800) : limites du champ de jeu.
— `ObjetFixe.HALF_SIZE` : rayon de demi-taille utilisé pour éviter la superposition.
— `GenerationRessource` : tableaux de poids et de valeurs nutritives prédéfinis `{1,2,3,4}` et `{10,30,60,100}`.
— `GenerationAbri` : tableau de capacités `{1,2,3}`.

3.3 Algorithme

1. À la création du `Terrain`, le programme place d'abord un `Nid` à des coordonnées aléatoires (en veillant à rester dans les limites).
2. Ensuite, il appelle les méthodes `ajouterAbris(nb)` et `ajouterRessources(nb)` qui s'appuient sur `GenerationAbri` et `GenerationRessource`.
3. Chacune de ces méthodes génère des coordonnées (x,y) aléatoires, vérifie que la zone est libre (pas d'overlap important avec un objet existant), puis crée l'objet (`Ressource` ou `Abri`) en l'ajoutant à la liste `elts` du `Terrain`.
4. Cette logique se répète pour chaque élément à insérer.

3.4 Conditions limites

— La carte ne doit pas contenir de collisions majeures (on évite l'insertion de deux `ObjetFixe` trop proches).
— Les coordonnées générées doivent être comprises entre `[HALF_SIZE, LARGEUR-HALF_SIZE]` et `[HALF_SIZE, HAUTEUR-HALF_SIZE]`.

3.5 Interaction avec les autres fonctionnalités

— Les ressources et abris ainsi placés servent à la *Gestion des ressources*, *Gestion de l'énergie*, etc.
— Le `Crapaud` est également initialisé à une position aléatoire sur la carte.

4 Fonctionnalité 2 : Gestion des ressources

4.1 Structures de données

— `Ressource` : héritée de `ObjetFixe`, stocke `poids`, `valeurNutritive` et la liste des fourmis présentes.
— `Transport` : classe de `Deplacement` dédiée au transport d'une `Ressource` par les fourmis jusqu'au `Nid`.
— `Terrain.expeditions` : liste des déplacements en cours (`Deplacement`).

4.2 Constantes du modèle

— `Ressource.isReadyToGo()` : le poids de la ressource doit être couvert par le nombre de fourmis présentes pour autoriser son déplacement.
— `Ressource.getValeurNutritive()` : sert à calculer le gain de `score` dans le `Nid`.

4.3 Algorithme

1. Lorsqu'un joueur (ou l'IA) décide de déplacer une `Ressource` (cf. clic + touche `ENTER`), on vérifie `isReadyToGo()`.
2. Si suffisant, on crée un objet `Transport` (`Terrain.ramenerRessourceALaMaison()`) qui prend en charge la mise à jour du déplacement.
3. À l'arrivée au `Nid`, la ressource augmente le `score` et les fourmis ayant participé se retrouvent dans le `Nid`.

4.4 Conditions limites

— La ressource ne se déplace que si le nombre de fourmis associées est au moins égal à son `poids`.
— Si pendant le transport, certaines fourmis meurent (énergie épuisée), le transport peut être interrompu (la ressource est alors reposée au sol).

4.5 Interaction avec les autres fonctionnalités

— L'énergie des fourmis est décrémentée pendant le transport (fonctionnalité *Gestion de l'énergie*).
— Le crapaud peut intercepter la ressource si elle est dans son champ de vision (fonctionnalité *Interaction avec le prédateur*).

— Le `Nid` reçoit la ressource et augmente le `score` (fonctionnalité *Système d'amélioration du nid* et *Tableau de scores*).

5 Fonctionnalité 3 : Gestion de l'énergie

5.1 Structures de données

— `Fourmi` : chaque fourmi a une `energie` allant de 0 à `MAX_ENERGIE` (100).
— `GestionnaireEnergie` : thread (`extends Thread`) qui appelle périodiquement `majEnergieFourmis()` sur le `Terrain`.

5.2 Constantes du modèle

— `Fourmi.MAX_ENERGIE = 100`, `Fourmi.VITESSE_MAX = 8`.
— `GestionnaireEnergie.DELAY = 100`ms : intervalle entre deux décrémentations d'énergie.

5.3 Algorithme

1. Toutes les `Fourmi` présentes, que ce soit en déplacement (`Deplacement.decrEnergieFourm` ou stationnées (`ObjetFixe.majEnergieFourmis()`), voient leur énergie s'ajuster (diminuer ou se recharger).
2. Dans un `Nid`, l'énergie des fourmis *augmente* (`incrEnergie()`) pour simuler la «recharge».
3. Dans un `Abri` ou en extérieur, l'énergie peut se stabiliser ou diminuer plus lentement (selon la logique exacte).
4. Quand l'énergie d'une fourmi passe à 0, elle meurt (`isDead() = true`), et est alors retirée des déplacements ou des listes.

5.4 Conditions limites

— Si une fourmi n'a plus d'énergie avant d'atteindre le `Nid`, elle n'est plus utilisable (on la supprime de la simulation).
— Le `GestionnaireEnergie` doit tourner en continu pour maintenir la cohérence des états d'énergie.

5.5 Interaction avec les autres fonctionnalités

— La vitesse d'une fourmi dépend directement de son énergie (lié à la formule dans `Fourmi.getVitesse()`).
— Les déplacements, la génération de ressources, le crapaud, etc. s'appuient sur cet état énergétique pour calculer la faisabilité ou la durée d'une action.

6 Fonctionnalité 4 : Interaction avec un prédateur (crapaud)

6.1 Structures de données

— `Crapaud` : classe gérant la position, le champ de vision (`visionRange`), la satiété, et l'état (réveillé / endormi).
— `GestionnaireCrapaud` : thread qui met régulièrement à jour le `Crapaud` (déplacements, baisse de satiété, etc.).

6.2 Constantes du modèle

— `Crapaud.MAX_SATIETE = 10` : seuil à partir duquel il s'endort.
— `Crapaud.SATIETE_AU_REVEIL = 5` : satiété avec laquelle il se réveille.
— `Crapaud.SEUIL_FAIM = 3` : en dessous, il commence à manger les ressources elles-mêmes (et pas seulement les fourmis).
— `Crapaud.DUREE_SIESTE = 10000`ms : durée du sommeil.

6.3 Algorithme (abstrait)

1. Le crapaud se déplace aléatoirement sur la carte, changeant parfois de direction (petite probabilité).
2. S'il est *affamé* (satiété faible), il mange directement les ressources (et les fourmis associées).
3. Sinon, il chasse d'abord les fourmis en déplacement (`Deplacement`) ; s'il n'en trouve pas, il s'attaque aux fourmis présentes dans les ressources proches.
4. Chaque fois qu'il mange, sa `satiété` augmente. Si elle dépasse `MAX_SATIETE`, il s'endort (*isAsleep* = true) pendant `DUREE_SIESTE`.
5. Quand il dort, `GestionnaireCrapaud` décrémente périodiquement sa `satiété`, jusqu'à ce qu'elle retombe à `SATIETE_AU_REVEIL`, ce qui le réveille.

6.4 Conditions limites

— Le crapaud doit être dans les limites du `Terrain` (on inverse la direction s'il atteint le bord).
— Lorsqu'il dort, il ne peut plus se déplacer ni manger (les fourmis peuvent donc en profiter pour transporter des ressources).

6.5 Interaction avec les autres fonctionnalités

— Le `Crapaud` peut détruire une expédition `Deplacement` (cas d'une fourmi interceptée).
— Il peut aussi consommer la `Ressource` elle-même si `satiète < SEUIL_FAIM`.
— `GestionnaireCrapaud` fonctionne en parallèle du `GestionnaireEnergie` et du `GestionnaireDeplacement`.

7 Fonctionnalité 5 : Système d'amélioration du nid

7.1 Structures de données

— `Nid` : stocke un `score` qui augmente à chaque ressource rapportée.
— `GestionScore` : thread qui surveille la volonté d'ajouter une fourmi dans le `Nid`.
— `Score` : encapsule la logique de «points» ou de «ressources accumulées».

7.2 Constantes du modèle

— `Score` incrémente ou décrémente via `augmenterScore(...)` / `diminuerScore()` : 3 points sont dépensés pour créer une nouvelle fourmi.
— `Score.AjoutFourmiPossible()` : teste si le `score` courant est suffisant (au moins 3).

7.3 Algorithme

1. À chaque ressource déposée au `Nid`, on augmente le `score` (en fonction de la valeur nutritive).
2. Le `GestionScore` attend (dans une boucle) qu'on *demande* la création d'une fourmi (par exemple, via un bouton ou une action).
3. S'il y a assez de points, `diminuerScore()` est appelé et le `Nid` ajoute une nouvelle fourmi (`nid.ajouterFourmi()`).

7.4 Conditions limites

— Le `Score` ne peut pas être négatif. Si < 3, on ne peut pas créer de nouvelle fourmi.
— La création de fourmis se fait *dynamiquement* pendant la partie : pas de limitation de quantités sauf le score.

7.5 Interaction avec les autres fonctionnalités

— Le `Nid` reçoit les ressources (fonctionnalité *Gestion des ressources*).
— La *Tableau de scores et statistiques* utilise la même logique de `Score`.

8 Fonctionnalité 6 : Tableau de scores et statistiques

8.1 Structures de données

— `Score` : classe contenant un entier représentant le score courant.
— `Sauvegarde` : classe permettant de lire/écrire des données dans un fichier (pour conserver un historique).

8.2 Constantes du modèle

— Le *chemin du fichier* de sauvegarde (`cheminFichier`) est paramétrable dans `Sauvegarde`.

8.3 Algorithme

1. En fin de partie (ou selon certaines actions), on appelle `Sauvegarde.sauvegarder(...)` en y passant la liste des scores ou statistiques à écrire.
2. Pour charger, `Sauvegarde.charger()` lit le fichier et reconstruit la liste de résultats.
3. Les données peuvent être affichées à l'écran via l'interface (par exemple sous forme de `JLabel` ou `JTable`).

8.4 Conditions limites

— Il faut un accès en écriture au fichier (sinon l'opération échoue).
— Les données lues doivent être cohérentes : si le fichier est mal formaté, l'opération peut échouer ou ignorer les données incorrectes.

8.5 Interaction avec les autres fonctionnalités

— Le `Score` est mis à jour par la collecte de ressources (fonctionnalité *Gestion des ressources*) et consulté par l'*amélioration du nid*.
— À la fin d'une partie, on peut sauvegarder le `Score` final et d'autres statistiques (nombre de fourmis mortes, etc.).

9 Fonctionnalité 7 : Implémentation de niveaux de difficulté

9.1 Structures de données

— `StartMenuController`, `DifficultePanel` : gèrent l'interface de sélection (facile, moyen, difficile).
— `Terrain` et ses gestionnaires : ajustent la *vitesse des fourmis*, la *consommation d'énergie*, la *rareté des ressources*, etc. (en fonction du niveau choisi).

9.2 Constantes du modèle

— Les vitesses ou taux de consommation d'énergie pourraient être modifiés en fonction d'une variable *difficulté* (p. ex. `difficultyMultiplier`).
— La `DUREE_SIESTE` ou `MAX_SATIETE` du crapaud peuvent aussi varier en mode *Difficile*.

9.3 Algorithme

1. Au lancement du jeu, l'utilisateur choisit la difficulté via `DifficultePanel`.
2. Le `StartMenuController` lit cette information et configure certains paramètres :
   — Nombre de ressources placées.
   — Consommation d'énergie par unité de temps.
   — Vitesse de déplacement des fourmis (`Fourmi.VITESSE_MAX`).
   — Etc.
3. Le reste du jeu (`GestionnaireEnergie`, `GestionnaireDeplacement`, etc.) s'exécute en tenant compte de ces valeurs.

9.4 Conditions limites

— La configuration initiale d'une difficulté doit être cohérente : par exemple, si on rend les ressources trop rares et les fourmis trop lentes, la partie peut devenir impossible.
— La sélection de difficulté doit être réalisée *avant* de lancer le `Terrain`.

9.5 Interaction avec les autres fonctionnalités

— Toutes les mécaniques sont plus ou moins affectées par le niveau de difficulté (énergie, fréquence de ressources, vitesse du crapaud, etc.).