Rapport du TPL de Programmation Orientée Objet
Simulation de systèmes Multi-agents

Introduction et Objectifs :

Ce projet Java se concentre sur la simulation de systèmes multi agents à travers le développement d'une application graphique. L'objectif initial est d'explorer en premier temps une simple simulation de Balles. Ensuite,l'implémentation de trois types d'automates cellulaires dans une première phase : le jeu de la vie de Conway, un jeu de l'immigration, et le modèle de ségrégation de Schelling. La seconde phase se penchera sur le modèle de Boids pour simuler le mouvement d'essaims auto-organisés.

Ce rapport détaillera la conception, l'implémentation, les tests et les résultats pour chaque système multiagent abordé, reflétant notre compréhension des concepts clés de la programmation orientée objet et de la simulation de systèmes multi agents.

Etat du projet:

Le projet est actuellement opérationnel, mettant en œuvre trois modèles distincts: Balls, Cells, et Boids.

Balls:

- Génère un nombre aléatoire de balles (entre 1 et 200) de couleurs variées.
- Les balles peuvent rebondir contre les murs.
- Le GUI a une taille de 1850*930 pixels.

Cells:

- La configuration du nombre de lignes et de colonnes est flexible pendant l'exécution (typiquement 40 lignes et 60 colonnes).
- Pour le modèle 2 et 3, le nombre d'états est également ajustable (typiquement entre 4 et 5).
- Pour le modèle 3, le seuil de voisins nécessaire pour déclencher un déplacement est également modifiable (habituellement entre 2 et 4).

Boids:

- Le nombre de boids est modulable, tout comme le nombre d'espèces.
- 30% des boids ont une masse de 1, tandis que les autres sont des prédateurs plus grands qui peuvent se préder entre eux.
- La taille des boids est proportionnelle à leur masse.
- Les boids sont attirés par les boids de masse inférieure ou égale.
- Les boids sont repoussés par les boids plus grands.
- Les boids ont tendance à se déplacer dans la même direction que leurs voisins.

Compilation et exécution :

Avec un makefile c'est plus facile !
On vous donne la chance de choisir les paramètres pour les automates cellulaires , le jeu de Conway et le modèle de Schelling !

Compilation:
Exécutez la commande “**make**” dans le terminal à la racine du projet. Cette commande compile l'ensemble des jeux et démos, y compris avec et sans l'EventManager.

Choix des Paramètres :
Vous avez la possibilité de choisir les paramètres pour les automates cellulaires, le jeu de Conway et le modèle de Schelling et les Boids.

Exécution d'un Jeu Spécifique :
Pour exécuter un jeu particulier avec l'EventManager, utilisez la commande spécifique fournie dans le Makefile.
Par exemple : “**make runTestBoidsEvent”**.

Choix de Conception:

Événements

Pour chaque type d'événement, comme l'exemple donné avec MessageEvent, des classes spécifiques (BallsEvent, CellsEvent, et BoidsEvent) sont créées, toutes héritant de la classe générique Event. Chaque classe d'événement implémente une méthode

execute(), qui appelle la méthode NextLogic (translate pour BallsEvent). Cette méthode NextLogic effectue le travail principal de la mise à jour des données, en faisant un pas dans le temps.

Simulateur

Un simulateur (BallsEventSimulator, CellsEventSimulator ou BoidsEventSimulator) est mis en place. Ce simulateur utilise la méthode scheduleSimulationEvents() pour créer des événements correspondant à 10000 pas de temps en utilisant l'EventManager. Ces événements sont liés à la structure spécifique, telle que Balls, Cells, ou Boids.

Classe Balls, Cells et Boids

Chaque classe possède une méthode nextLogic (translate pour Balls) qui implémente la logique d'évolution spécifique à cette classe. C'est ici que la mise à jour des données et le passage du temps sont effectués pour les entités correspondantes (balles, cellules, boids).

Classe Cells

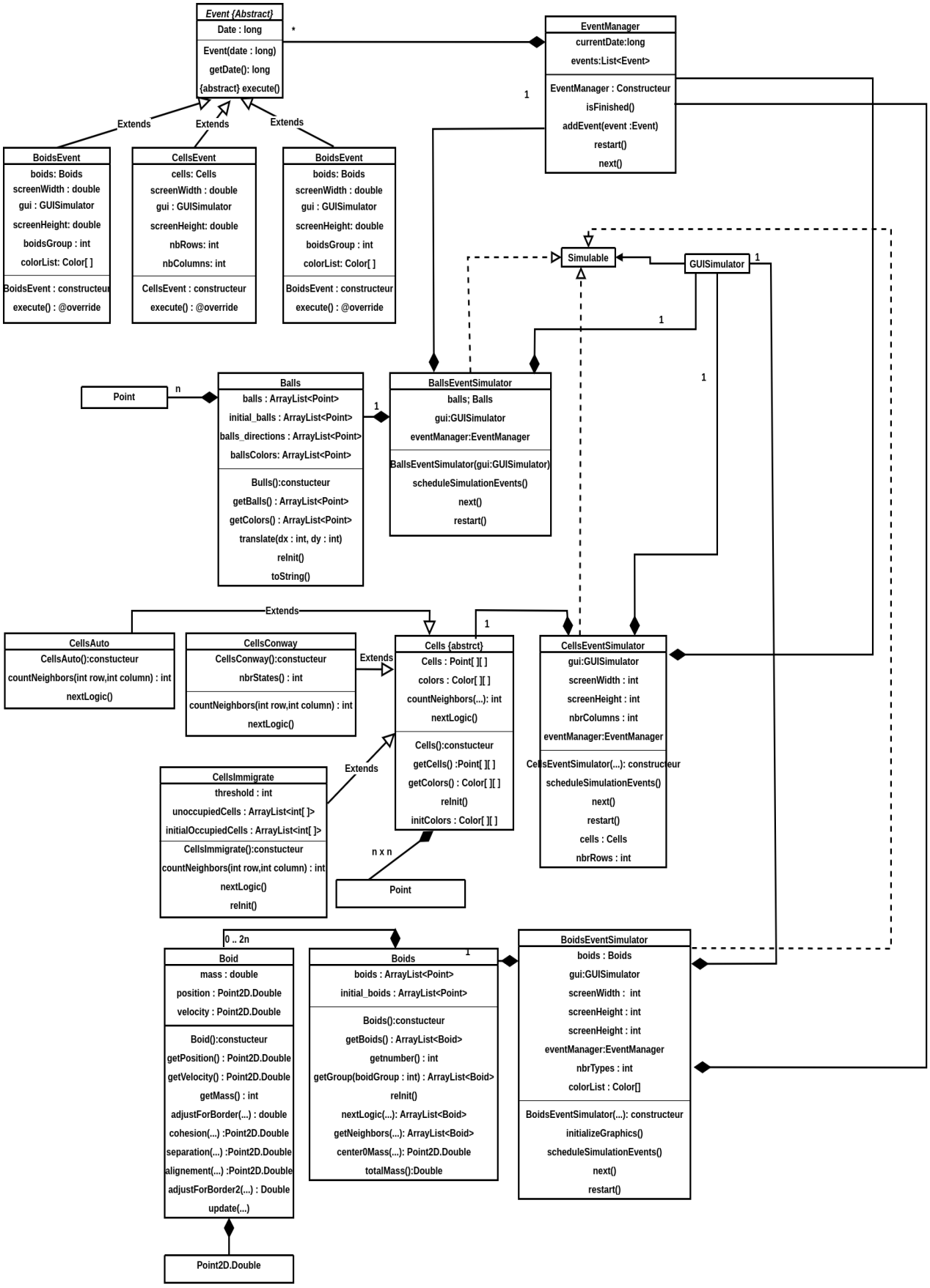
La classe Cells est conçue comme une classe abstraite qui contient du code partagé entre trois modèles d'automates cellulaires différents : le Jeu de la Vie, le Modèle de Shelling, et un autre modèle non spécifié.

Pour éviter la redondance et pour favoriser la réutilisation du code, chaque modèle d'automate cellulaire spécifique (CellsAuto, CellsConway, CellsImmigrate) hérite de la classe abstraite Cells. Cette approche permet de centraliser le code commun dans la classe abstraite, évitant ainsi la duplication inutile.

En résumé, la classe Cells sert de base générique pour les 3 modèles en factorisant le code commun, ce qui facilite la maintenance, la gestion des modifications, et offre une flexibilité pour l'ajout éventuel de nouveaux modèles. Chaque modèle spécifique hérite de cette classe abstraite pour partager et étendre le code commun.

Choix de Collections:

Dans Boids on utilise une arrayList pour le Dynamisme de la taille de la liste nécessaire (on mange de boids)
Dans Cells on utilise une arrayList pour les unoccupied celles (les maisons vacantes) pour la même raison, le nombre global ne change pas mais on peut modifier la liste simplement.


Event {Abstract}
Date : long
Event(date : long)
getDate(): long
{abstract} execute()
*
EventManager
currentDate:long
events:List<Event>
EventManager : Constructeur
isFinished()
addEvent(event :Event)
restart()
next()
1
Extends
Extends
Extends
BoidsEvent
boids: Boids
screenWidth : double
gui : GUISimulator
screenHeight: double
boidsGroup : int
colorList: Color[ ]
BoidsEvent : constructeur
execute() : @override
CellsEvent
cells: Cells
screenWidth : double
gui : GUISimulator
screenHeight: double
nbRows: int
nbColumns: int
CellsEvent : constructeur
execute() : @override
BoidsEvent
boids: Boids
screenWidth : double
gui : GUISimulator
screenHeight: double
boidsGroup : int
colorList: Color[ ]
BoidsEvent : constructeur
execute() : @override
Simulable
GUISimulator
1
1
1
Point
n
Balls
balls : ArrayList<Point>
initial_balls : ArrayList<Point>
balls_directions : ArrayList<Point>
ballsColors: ArrayList<Point>
Bulls():constucteur
getBalls() : ArrayList<Point>
getColors() : ArrayList<Point>
translate(dx : int, dy : int)
relnit()
toString()
1
BallsEventSimulator
balls; Balls
gui:GUISimulator
eventManager:EventManager
BallsEventSimulator(gui:GUISimulator)
scheduleSimulationEvents()
next()
restart()
1
Extends
CellsAuto
CellsAuto():constucteur
countNeighbors(int row,int column) : int
nextLogic()
CellsConway
CellsConway():constucteur
nbrStates() : int
countNeighbors(int row,int column) : int
nextLogic()
Extends
Cells {abstrct}
Cells : Point[ ][ ]
colors : Color[ ][ ]
countNeighbors(...): int
nextLogic()
Cells():constucteur
getCells() :Point[ ][ ]
getColors() : Color[ ][ ]
relnit()
initColors : Color[ ][ ]
1
CellsEventSimulator
gui:GUISimulator
screenWidth : int
screenHeight : int
nbrColumns : int
eventManager:EventManager
CellsEventSimulator(...): constructeur
scheduleSimulationEvents()
next()
restart()
cells : Cells
nbrRows : int
CellsImmigrate
threshold : int
unoccupiedCells : ArrayList<int[ ]>
initialOccupiedCells : ArrayList<int[ ]>
CellsImmigrate():constucteur
countNeighbors(int row,int column) : int
nextLogic()
relnit()
Extends
n x n
Point
0 .. 2n
Boid
mass : double
position : Point2D.Double
velocity : Point2D.Double
Boid():constucteur
getPosition() : Point2D.Double
getVelocity() : Point2D.Double
getMass() : int
adjustForBorder(...) : double
cohesion(...) :Point2D.Double
separation(...) :Point2D.Double
alignement(...) :Point2D.Double
adjustForBorder2(...) : Double
update(...)
Point2D.Double
Boids
boids : ArrayList<Point>
initial_boids : ArrayList<Point>
Boids():constucteur
getBoids() : ArrayList<Boid>
getnumber() : int
getGroup(boidGroup : int) : ArrayList<Boid>
relnit()
nextLogic(...): ArrayList<Boid>
getNeighbors(...): ArrayList<Boid>
center0Mass(...): Point2D.Double
totalMass():Double
1
BoidsEventSimulator
boids : Boids
gui:GUISimulator
screenWidth : int
screenHeight : int
screenHeight : int
eventManager:EventManager
nbrTypes : int
colorList : Color[]
BoidsEventSimulator(...): constructeur
initializeGraphics()
scheduleSimulationEvents()
next()
restart()