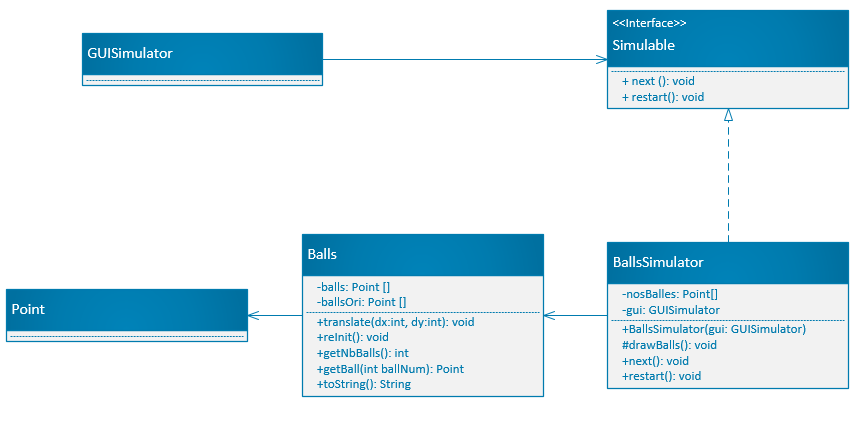
TPL POO Livrable

# Les Balles



## Choix de conception

### Classe Balls

* Le constructeur initialise le premier tableau balls[] qui contient les positions des balles courantes a des coordonnées fixées au centre puis les copie dans ballsOri[].
* Translate permet de déplacer toutes les balles de la simulation.
* reInit() réinitialise balls[] aux valeurs conservées dans ballsOri[].
* getNbBalls renvoie simplement la longueur du tableau balls ce qui évite d’ajouter une variable nbBalls dans la classe.
* getBall() permet d’accéder a une balle dans le tableau des balles courantes balls[].
* toString() renvoie un StringBuffer avec le nombre de points et leurs coordonnées.

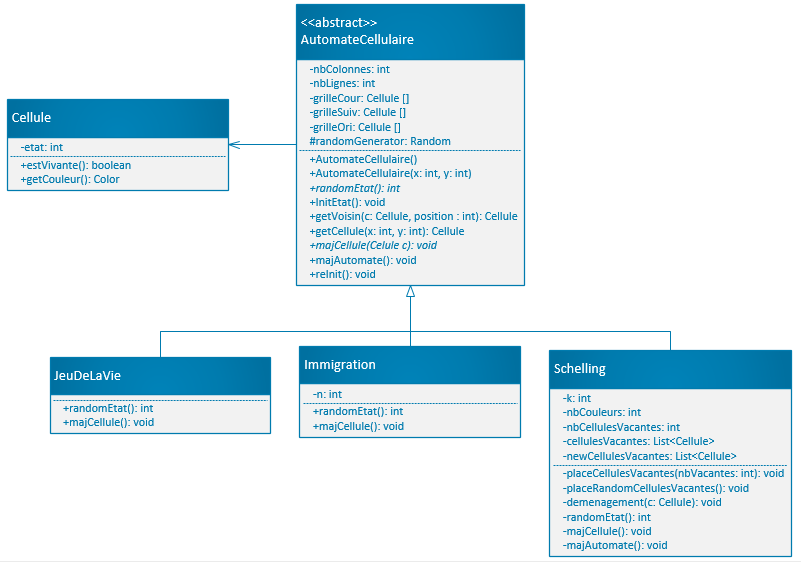
### BallsSimulator

* Le constructeur initialise le GUISimulator, les balles et les dessine grâce a la méthode drawBalls().
* Comme BallsSimulator réalise l’interface simulables elle doit redéfinir les méthodes next et restart.

## Tests et Résultats

La classe TestBalls vérifie me bon fonctionnement de Balls sur la console. La classe TestBallsSimulator test BallsSimulator. Grâce a un scanner l’utilisateur peut saisir un nombre de balles pour la simulation.

# Automates Cellulaires



## Choix de conception

Les automates sont tous regroupés sous une même classe abstraite: la classe AutomateCellulaire. Chaque automate possède trois grilles de cellules : grilleCour les cellules a l’instant t, grilleSuiv les cellules a l’instant t+1 et grilleOri les cellules dans leur position d’origine ainsi que leur nombre de lignes et de colonnes.

La classe Cellule hérite de Point, on lui ajoute en variable un état (int) et en méthode :

* getCouleur() qui renvoie selon l’état de la cellule une couleur parmi les 10 choisies
* estVivante qui renvoie une boolean true si l’état est supérieur à 1.

La classe abstraite AutomateCellulaire

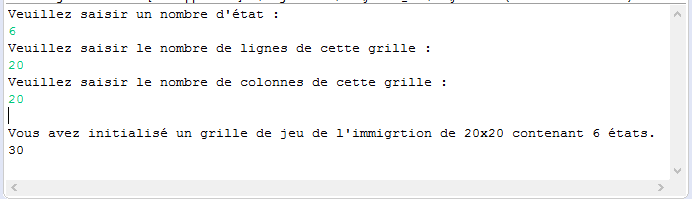
* Elle possède deux constructeurs : celui par défaut et un constructeur avec choix de lignes et de colonnes.
* Une méthode initEtat qui initialise les états lors de la création du jeu.
* Un abstract int randomEtat qui génère un état aléatoire selon l’automate.
* getVoisin(Cellule c,int i) qui permet de récupère la cellule voisine de c en position i.
* majAutomate() qui met à jour la grille grâce à la méthode abstraite majCellule qui elle-même met à jour une cellule selon les règles de chaque jeu.

Ainsi dans chaque automate il n’y a plus qu’à définir randomEtat, majCellule, leurs constructeurs, ainsi que leurs variables propres :

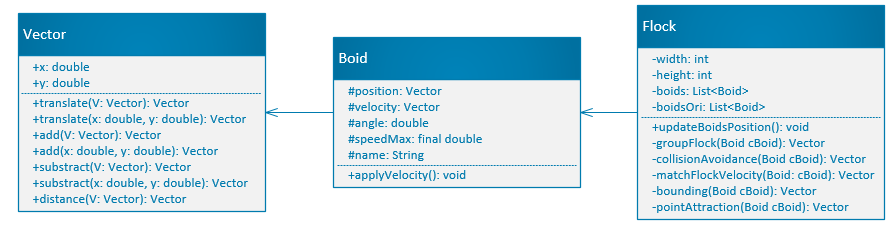
* Immigration : on ajoute une variable int n = le nombre d’états
* Schelling : on ajoute les variables int k = seuil de la simulation, int nbCouleurs, nbCellulesVacantes et les listes de cellules celluleVacantes et newCellulesVacantes ainsi que les méthodes suivantes : placeCellulesVacantes pour placer un nombre fixe de cellules vacantes sur la grille courante, placerandomCelluleVacantes pour en placer un nombre aléatoire, déménagement pour déplacer les familles.  
  Remarque : il faut aussi redéfinir la méthode MajAutomate pour cet Automate car la liste des cellules vacantes doit se mettre à jour.

## Tests & Résultats

Dans les classes TestJdlvSimulator, TestImigrationSimulator et TestSchellingSimulator on ajoute des scanners qui permettent à l’utilisateur de choisir la taille de la grille, et le nombre d’états/couleurs pour les deux derniers :



# Boids



## Choix de conception

Nous avons decider d’implementer 5 regles pour notre flock de boids :

1. Deplacement vers le centre du groupe
2. Separation entre les boids
3. Eviter les bors de la simulation
4. Attraction vers la nourriture