Teodoro Douglas, Humbert Jérôme, Stettler Christian,
Issom David

h e g | Haute école de gestion de Genève

Projet 632-1

Modalité du projet

Durée

- Rendu : semaine 15

Groupes

- 2 personnes

Évaluation

- Ce projet est noté et comptera pour 15% de votre note COO/JAVA.
- En cas de triche, la note de 1 est attribuée **au module**
  - **Les projets de l'année précédente seront aussi comparés**

Critères d'évaluation

- Les fonctionnalités du projet
- La documentation de vos méthodes et vos classes
- Le respect des standards de programmation
  - Indentation
  - Noms de classes/variables
  - Conventions
- Qualité de votre code (couplage/cohésion)
- Structures de données utilisées

Temps à disposition pendant le cours

- Heures de laboratoires

Ce projet est basé sur le projet de Gauthier Picard
(https://www.emse.fr/~picard/cours/1A/java/projet.html)

Teodoro Douglas, Humbert Jérôme, Stettler Christian,
Issom David

Planning

Avant la semaine du 11

- Formation de groupes : envoyer les noms du binôme à Jérôme

Semaine 11

- Lecture du sujet
- Création du projet BlueJ et Importation de la librairie
- Analyse du problème et pré-codage des classes

Semaine 12

- Coder l'interpréteur

Semaine 13

- Coder le simulateur et le comportement du terrain

Semaine 14

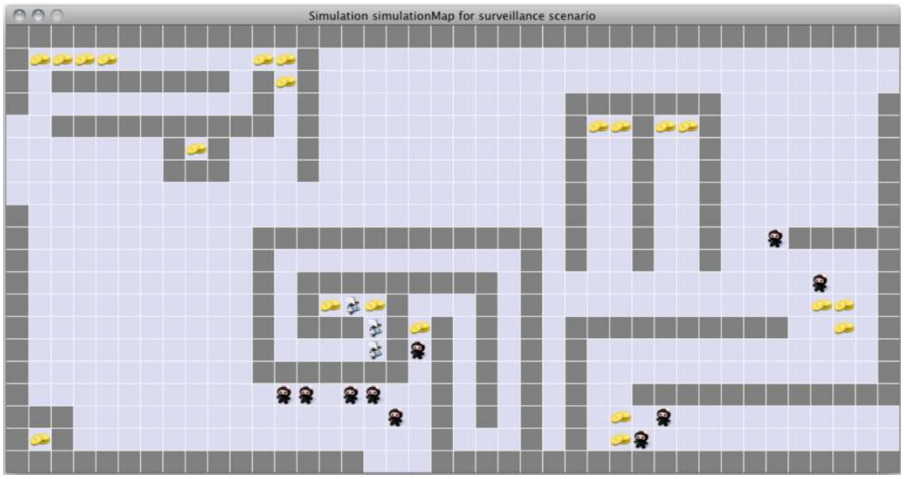
- Coder le comportement des entités
- Finalisation et livraison (code) du projet

Teodoro Douglas, Humbert Jérôme, Stettler Christian,
Issom David

Introduction

Le projet de Java a pour but de développer, en binômes, une plate-forme de simulation à temps discret (pas à pas). Le programme réalisé devra simuler une opération de surveillance d'une carte par un ensemble de robots autonomes (ou drones). Des intrus seront également présents sur la carte afin de dérober des sources d'argent.

Les drones doivent attraper les intrus, les intrus doivent attraper les sources d'argent puis fuir par les sorties de la carte. La simulation se termine lorsque plus aucun intrus n'est sur la carte, soit parce qu'ils ont été attrapés par les drones, soit parce qu'ils ont réussi à s'enfuir par une sortie.



Description du monde

Le scénario de surveillance présente plusieurs sortes d'entités qui sont activées par un simulateur :

- `Drone` : un robot autonome poursuivant les intrus ;
- `Intrus` : un intrus cherchant les sources d'argent, et à sortir de la carte une fois les sources épuisées ;
- `Argent` : une source d'argent (ou sac) recherchée par les intrus ;
- `Murs` : des barrières (ou murs) qui ne peuvent être traversées par les entités.

Le simulateur

Le positionnement des diverses entités sera indiqué dans un fichier texte à interpréter avant le lancement de la simulation. Une fois le fichier de description de la carte chargée, la simulation est effectuée pas à pas : à chaque pas de simulation (ou tour) toutes les entités qui peuvent se déplacer (drone ou intrus) se déplacent puis agissent (e.g. un drone se déplace puis attrape un intrus), les unes après les autres. Les autres entités (source d'argent et barrières) ne se déplacent pas. L'ordre d'action est déterminé aléatoirement à chaque pas.

Le simulateur manipule au moins les données suivantes :

- La carte du monde : une grille de cases pouvant contenir chacune au plus une entité ;
- La liste des drones présents ;
- La liste des intrus présents ;
- La liste des sources d'argent.

Les drones

Les drones sont des entités autonomes cherchant à attraper les intrus. Un robot peut attraper un intrus uniquement lorsque ce dernier se situe dans une case adjacente (même en diagonale) à celle du drone. Un drone ne peut attraper qu'un seul intrus par tour, et ne peut se déplacer que d'une seule case par tour (même en diagonale). En tant qu'entité mobile, un drone doit se déplacer (s'il n'est pas bloqué) puis agir si cela est possible (e.g. un intrus est dans une case adjacente, il peut alors l'attraper).

Les intrus

Les intrus agissent comme les drones, mais leurs cibles sont les sources d'argent. Comme un drone, un intrus ne peut attraper qu'une seule source d'argent par tour, et ne peut se déplacer que d'une seule case par tour. En tant qu'entité mobile, un intrus doit se déplacer (s'il n'est pas bloqué) puis agir si cela est possible (e.g. une source d'argent est dans une case adjacente, il peut alors l'attraper). Un intrus peut récupérer un maximum de 2 sacs, mais se voit ralenti : s'il transporte n sacs un intrus peut agir 3−n tours tous les 3 tours (ex : un intrus transportant 2 sacs agira 1 tour sur 3).

Lorsqu'aucune source d'argent n'est présente dans la carte ou qu'il ne peut plus transporter de sac, un intrus doit atteindre une sortie de la carte. Une sortie est une case vide située au bord de la carte. Un intrus peut disparaître de la carte pour deux raisons : soit il a été capturé par un drone, soit il s'est échappé par une sortie de la carte. Dans ces deux cas, un intrus ne peut plus agir dans le monde.

Les sources d'argent (ou sacs)

Les sources d'argent n'agissent pas sur le monde (pas de déplacement, etc.). Elles disparaissent de la carte lorsqu'un intrus les attrape (on parle alors de sacs). Si l'intrus est attrapé, le sac revient au drone, qui ne sera pas entravé de la même manière qu'un voleur : un drone n'est pas limité dans le nombre de sacs transportables, et sa vitesse reste constante.

Les murs

Les murs n'agissent pas sur le monde. Ils ne font que limiter les déplacements des drones et des intrus, car une fois positionné sur une case, un mur empêche une autre entité de l'occuper.

Tâches à réaliser

Afin de développer cette plate-forme, nous pouvons identifier les tâches suivantes que nous conseillons fortement de suivre :

- Analyser le problème, et identifier les classes à développer ;
- Pré-coder les classes représentant les entités (drone, intrus, etc.) même si pour l'instant elles n'ont aucun comportement de déplacement ou d'action ;
- Coder l'interpréteur de fichier texte pour charger des cartes ;
- Coder le simulateur afin de lancer la simulation pas à pas, i.e. charger la carte puis activer les entités tour par tour jusqu'à la fin de la simulation ;
- Coder le comportement des entités afin de respecter les besoins exprimés dans la section « Description du monde ».

Analyser le problème

Analyser le problème consiste à déterminer les classes à coder ainsi que leurs propriétés (attributs et méthodes) et leurs relations. On peut notamment utiliser un diagramme de classes UML pour représenter ces informations.

Pré-coder les classes

Dans un premier temps, on peut ne pas coder le comportement des classes précédemment identifiées, i.e. on peut laisser le corps des méthodes vide. Ce codage sera effectué plus tard, lorsque l'on dispose du simulateur et de la possibilité de visualiser le déroulement de la simulation. Il sera possible à tout moment d'ajouter des classes au modèle si nécessaire.

Coder l'interpréteur

Afin de pouvoir simuler différentes cartes, il est utile de disposer d'un interpréteur de fichier texte qui pour un fichier texte passé en paramètre renvoie un monde avec des entités positionnées initialement. Par exemple, il pourra fournir une méthode `World.parse(String file)` où `World` est la classe représentant un monde simulé, et `file` est le nom du fichier texte à interpréter. Donc l'interpréteur devra créer de nouveaux objets, comme un monde ou des voitures, par exemple.

Le format du fichier texte contenant la description de la carte est présenté dans la section suivante.

Afin de visualiser le déroulement de la simulation, une librairie graphique simple d'utilisation vous sera fournie. Ceci pourra s'avérer utile pour vérifier que le chargement de la carte est correct, par exemple.

Coder le simulateur

Le simulateur doit permettre de faire agir les entités au tour par tour. Le simulateur suit le même comportement jusqu'à ce que la simulation soit terminée (e.g. il n'y a plus d'intrus sur la carte). Avant son lancement, le simulateur doit donc charger une carte puis suivre une boucle qui fait agir les entités une à une dans un ordre aléatoirement déterminé à chaque tour. Le simulateur pourra notamment hériter de la classe `Simulator` fournie dans la librairie `simulation.jar`.

Coder le comportement des entités

Une fois l'infrastructure en place, simulateur, interpréteur, etc., le codage du comportement des entités peut commencer. Bien sûr cela n'empêche pas de revenir sur le code des classes précédemment codées, pour les rectifier (en cas de bug) ou les améliorer.

Attention, plusieurs choix de conception sont possibles, et peuvent demander plus ou moins d'effort de développement. Pensez à développer de manière incrémentale : commencer par des classes simples, puis leur ajouter des fonctionnalités au fur et à mesure. À chaque fois que vous ajoutez une fonctionnalité à une classe, pensez à la tester avant de continuer plus avant.

Le comportement des drones et des intrus nécessite qu'ils soient capables de trouver un trajet entre leur position et celle de leur cible (par exemple, un drone va calculer le plus court chemin entre sa position et celles des intrus afin de déterminer quelle sera sa prochaine case). Nous imposons l'utilisation de l'algorithme A*, présenté plus bas. Ce programme est fourni dans une librairie inclue dans la librairie graphique.

Format du fichier de description de carte

Le fichier de description de carte devra respecter le format suivant. La première ligne contient le nombre de lignes et le nombre de colonnes séparés par un caractère «espace» : <row> <col>. Par exemple : 5 7

Les lignes suivantes (autant que le nombre <row> indiqué en première ligne) doivent contenir autant de caractères (séparés par des caractères «espace») que le nombre <col> indiqué en première ligne. Ces caractères peuvent être :

`_` : pour une case vide ;

`#` : pour une barrière ;

`D` : pour un drone ;

`I` : pour un intrus ;

`$` : pour une source d'argent.

Par exemple :

```
# _ _ I _ $ #
```

Voici donc un exemple de fichier à 5 lignes, 7 colonnes, 1 intrus, 1 drone, 2 sources d'argent et une sortie :

```
5 7
# # # # # # #
# _ _ I _ $ #
# _ _ _ _ $ #
# D _ _ _ _ _
# # # # # # #
```

Il est à la charge de l'interpréteur de vérifier si le fichier est cohérent, i.e., le bon nombre de lignes, de colonnes, toutes les lignes ont le même nombre de caractères, etc.

Utilisation de la librairie simulation.jar

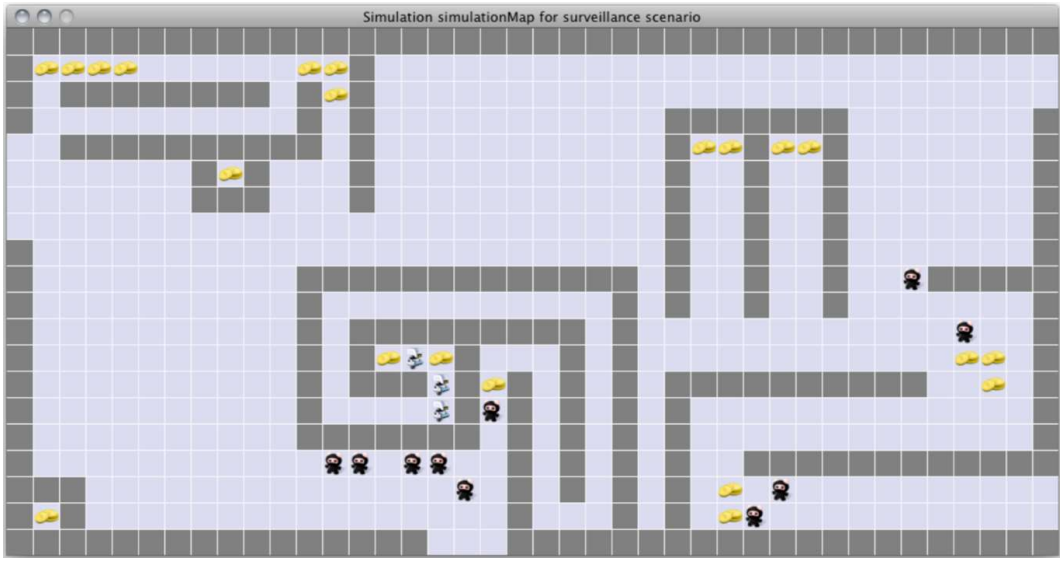
Afin de vous aider dans le développement de votre simulateur, nous vous fournissons des outils contenus dans une librairie. Pour l'utiliser, voici la démarche à suivre :

- Importer la librairie `simulation.jar` dans votre projet BlueJ :
  - Outils>Préférences>Bibliothèques>AddFiles
  - Sélectionnez `simulation.jar`
  - Redémarrer BlueJ

Cette librairie vous fournira des classes et interface pour visualiser une simulation ou utiliser l'algorithme A*.

Visualisation de la simulation

Au lieu de visualiser le monde simulé par du texte dans la console, nous proposons d'utiliser une librairie graphique inclue dans `simulation.jar`. Ainsi, lors de la simulation, une fenêtre apparaîtra et affichera le contenu de la carte, comme présenté dans la figure ci-dessous.



Créer et utiliser l'afficheur graphique

L'utilisation de cette librairie a été simplifiée afin de ne pas perdre de temps de conception sur l'interface graphique. La démarche à suivre pour utiliser une telle fenêtre est la suivante :

1. S'assurer d'avoir importé la librairie comme présenté ci-dessus.
2. Créer une nouvelle fenêtre :

```
MapFrame frame = new MapFrame(world);
```

Ici la variable world est une instance d'une classe implémentant l'interface `SimulationMap`, comme le montre la signature du constructeur : `public MapFrame(SimulationMap map)`.

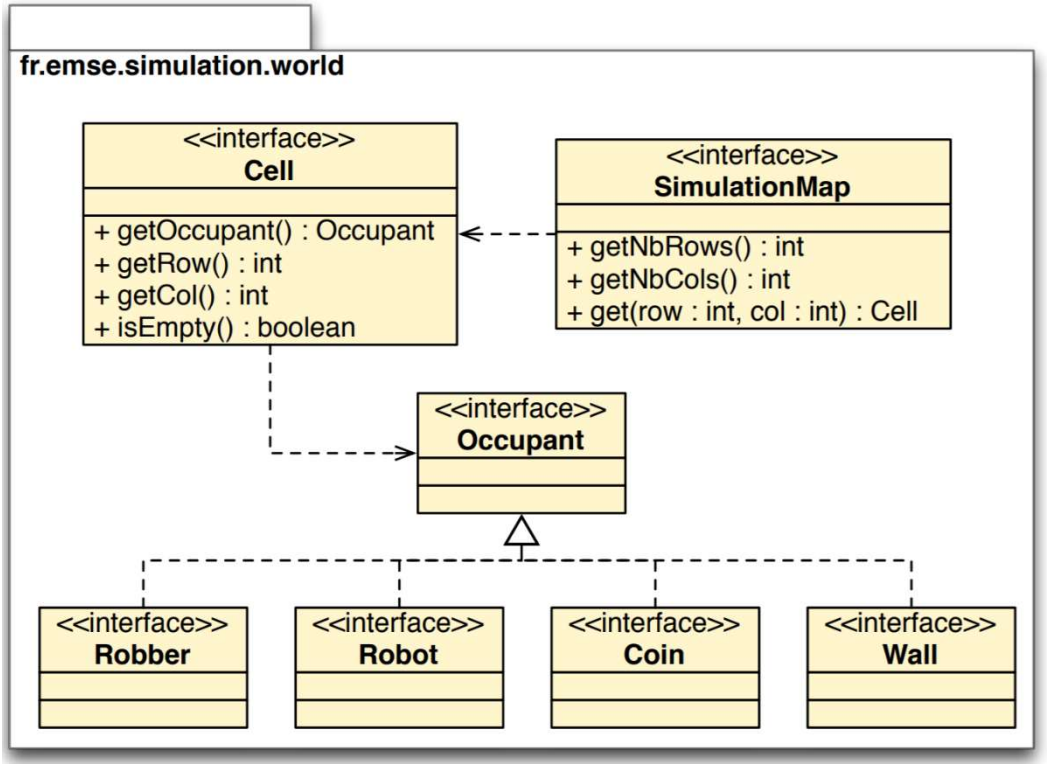
3. Chaque fois que l'on veut rafraîchir le contenu de la fenêtre (e.g. à chaque action d'une entité), appeler la méthode `public void repaint(int time)` de la fenêtre :

```
frame.repaint(time);
```

Ici la variable `time` est un entier `int` représentant le temps en millisecondes d'attente après que la fenêtre soit rafraîchie, afin que l'affichage ne soit pas trop rapide. Augmentez ce nombre pour ralentir l'animation, diminuez le pour accélérer.

Les interfaces utilisées par l'afficheur graphique

La figure suivante présente les interfaces fournies dans le paquetage `fr.emse.simulation.world` de la librairie `simulation.jar`.



Afin de savoir ce que doit afficher une `MapFrame`, nous fournissons un ensemble d'interfaces. En effet, une `MapFrame` affiche un objet de type `SimulationMap`, qui est une interface fournie dans `simulation.jar` :

```
public interface SimulationMap {
  public int getNbRows();
  public int getNbCols();
  public Cell get(int row, int col);
}
```

Une `MapFrame` affiche une carte `SimulationMap` en la parcourant grâce à la méthode `get(int, int)` qui retourne un objet de type `Cell` :

```
public interface Cell {
  public Occupant getOccupant();
  public int getRow();
  public int getCol();
  public boolean isEmpty();
}
```

Le contenu d'une `Cell` est de type `Occupant` :

```
public interface Occupant {}
```

Afin de savoir quelle forme afficher pour une cellule, `Occupant` présente plusieurs sous-types, `Robot`, `Robber` et `Coin`, correspondant à l'usage de trois images différentes. De plus, si un `Occupant` est de type `Wall`, la case sera simplement grisée.

Ainsi pour pouvoir afficher une carte dans l'afficheur, il faut créer une classe implémentant l'interface `SimulationMap`, contenant des objets d'une classe implémentant l'interface `Cell`, contenant des objets implémentant les interfaces `Robber` ou `Robot`. Il vous faudra donc développer des classes réalisant ces interfaces, et donc implémentant toutes leurs méthodes. Par exemple :

```
public class Carte implements SimulationMap {
  // Votre code ici
}
public class Cellule implements Cell {
  // Votre code ici
}
public class Drone implements Robot {
  // Votre code ici
}
```

L'algorithme A*

L'algorithme de recherche heuristique A* est un classique pour la recherche de parcours dans un graphe. Il peut aisément être adapté pour la recherche du plus court chemin dans une carte de type `SimulationMap`. L'idée de cet algorithme est d'explorer la carte de case en case, en évaluant à la fois le coût pour avoir atteint la case courante ainsi que le coût espéré pour atteindre la case objectif.

Au lieu d'implémenter vous-même cet algorithme, nous fournissons dans la librairie `simulation.jar` une classe qui permet de l'utiliser. Encore faut-il bien l'appeler... La principale classe pour exécuter l'algorithme A*est la classe `AStarPathFinder`, et notamment la méthode `getShortestPath` qui renvoie la liste des cellules à parcourir pour atteindre une cellule objectif en partant de la cellule actuelle. Voici un exemple d'utilisation de cette classe :

```
1  ArrayList<Class<? extends Occupant>> toIgnore = new ArrayList<Class<? extends Occupant>>();
2  toIgnore.add(Drone.class);
3  AStarPathFinder solver = new AStarPathFinder(
4        world,
5        new EuclideanDistanceHeuristic(),
6        new PreferEmptyCellsLocalCost(1, 3),
7        toIgnore);
8  ArrayList<Cell> path = solver.getShortestPath(position, target);
```

Le constructeur de `AStarPathFinder` (ligne 3) prend 4 arguments :

- Une carte (ligne 4) pour connaître les cellules ;
- Une heuristique (ligne 5) pour savoir quelle distance prendre en compte pour calculer les distances estimées (ici la distance euclidienne) ;
- Un coût (ligne 6) de passage au travers les cellules en fonction de leur état (occupée ou pas) ;
- Une liste de classes d'occupants à ignorer (ligne 8) pour calculer un itinéraire indépendamment des entités présentes sur les cases.

Ici, la liste de classe d'occupants à ignorer contient uniquement la classe `Drone` (lignes 1 et 2). Ainsi, lorsque le trajet sera calculé les cases actuellement occupées par des objets de type `Drone` (mais qui seront certainement libérées dans le futur) seront considérées comme traversables. Par contre, le fait qu'une case soit occupée peut être spécifié comme plus coûteux grâce au coût de type `PreferEmptyCellsLocalCost` qui ici considèrera plus intéressant de passer par une case vide (coût 1) qu'une case occupée (coût 3). Ainsi pour un même nombre de cases, un trajet passant par plus de cases vides sera préféré.

Enfin, l'appel à la méthode `getShortestPath()` renvoie une liste de cellules (ligne 8) à parcourir pour aller de la cellule position à la cellule target (position inclue). S'il n'existe aucun chemin pour aller de position à target, la méthode renverra `null`. Consultez la documentation de toutes ces classes avant de les utiliser.

Quelques liens utiles

- l'API Java : https://docs.oracle.com/en/java/javase/11/docs/api/index.html
- le tutoriel officiel Java, pour vous améliorer en Java : http://java.sun.com/docs/books/tutorial/
- l'article de wikipedia sur A* : http://en.wikipedia.org/wiki/A*
- La documentation de la librairie `simulation.jar` https://www.emse.fr/~picard/cours/1A/java/projet/doc/

Crédits

Gauthier Picard : https://www.emse.fr/~picard/cours/1A/java/projet.html