Simulation d'une équipe de robots pompiers

Ensimag 2A 2014-15 - TP POO

Résumé

L'objectif de ce TP est de développer en `Java` une application permettant de simuler une équipe de robots pompiers évoluant de manière autonome dans un environnement naturel [1].

La première partie est très guidée : une conception de solution est proposée, définissant les classes représentant les données (carte, robots, incendies). Un code d'interface graphique vous est distribué pour l'affichage. Les parties suivantes permettent de réaliser des simulations de difficulté croissante : d'abord déplacer des robots de manière contrôlée sur la carte, puis être capable de trouver les plus courts chemins pour se rendre à un endroit donné. La dernière étape consiste ensuite à organiser les déplacements et interventions des différents robots afin d'éteindre tous les incendies au plus vite.

Ce TP vous permettra d'aborder au fur et à mesure les aspects fondamentaux de la programmation orientée objet : encapsulation, délégation, héritage, abstraction et utilisation des collections *Java*.

1 Première partie : les données du problème

Cette section présente l'ensemble des données du problème de simulation et les classes `Java` les représentant. Elle décrit également l'interface graphique fournie, qui vous permettra de créer la base du simulateur : la lecture puis l'affichage des données.

1.1 Les différentes entités

1.1.1 La carte et les cases

Tous les éléments de la simulation se situent sur une carte représentée par une matrice $n \times m$ de cases. Toutes les cases sont carrées, de même taille et de même altitude (le terrain est supposé plat). Une case est caractérisée par ses coordonnnées (ligne, colonne) et la nature du terrain qu'elle représente : terrain libre, forêt, roche, eau ou habitat.

La description d'une carte est lue dans un fichier (voir le format en annexe A) et ne ne sera jamais modifiée après création.

1.1.2 Les incendies

Un incendie est défini par la case sur laquelle il se situe et le nombre de litres d'eau nécessaires pour l'éteindre. La propagation des incendies ne sera pas modélisée dans le cadre de ce TP, même si cela pourrait être facilement fait en extension. L'état initial des incendies sera lu avec la carte dans le fichier de description, et le nombre de litres nécessaires sera simplement décrémenté lorsqu'un robot versera de l'eau sur l'incendie.

1.1.3 Les robots

Les robots sont des « pompiers élémentaires » qui peuvent se déplacer, déverser de l'eau sur un incendie et remplir leur réservoir.

Un robot est situé sur une case, et peut se déplacer d'une case à la fois dans les directions cardinales (Nord, Sud, Est ou Ouest) si la case voisine lui est accessible. Plusieurs robots peuvent se trouver simultanément sur une même case, et les robots peuvent traverser des cases en feu. Un robot dispose aussi d'un réservoir d'eau, et déverse une quantité d'eau donnée à chaque intervention. Lorsqu'il est vide, il doit aller se remplir.

Plusieurs types de robots peuvent être utilisés : terrestres (robot à roues, à chenilles ou à pattes) ou aériens (drones). Ils possèdent des propriétés différentes, inhérentes à leur type ou lues dans le fichier de description des données. Par exemple, un robot à chenilles ne peut pas se déplacer sur du rocher ou sera

1. Ce projet est directement inspiré d'un projet original de Ch. Garion (ISAE), avec son aimable autorisation.

ralenti en forêt, ou encore un robot à pattes a un réservoir de capacité illimitée (il utilise en fait de la poudre). De même, tous les robots terrestres peuvent se remplir à côté d'une case en eau, alors que les drônes se remplissent sur une case contenant de l'eau.

Les propriétés de ces différents robots sont décrites en annexe B.

1.2 Hiérarchie des classes représentant les données

La conception des classes permettant des représenter les différentes données est très guidée, même si vous pouvez modifier ce qui est proposé selon vos besoins. Les figures suivantes représentent des *diagrammes de classes*, avec la notation *UML*[2]. La description n'est pas forcément complète, et il vous revient de choisir comment implémenter ces classes et les relations entre elles.

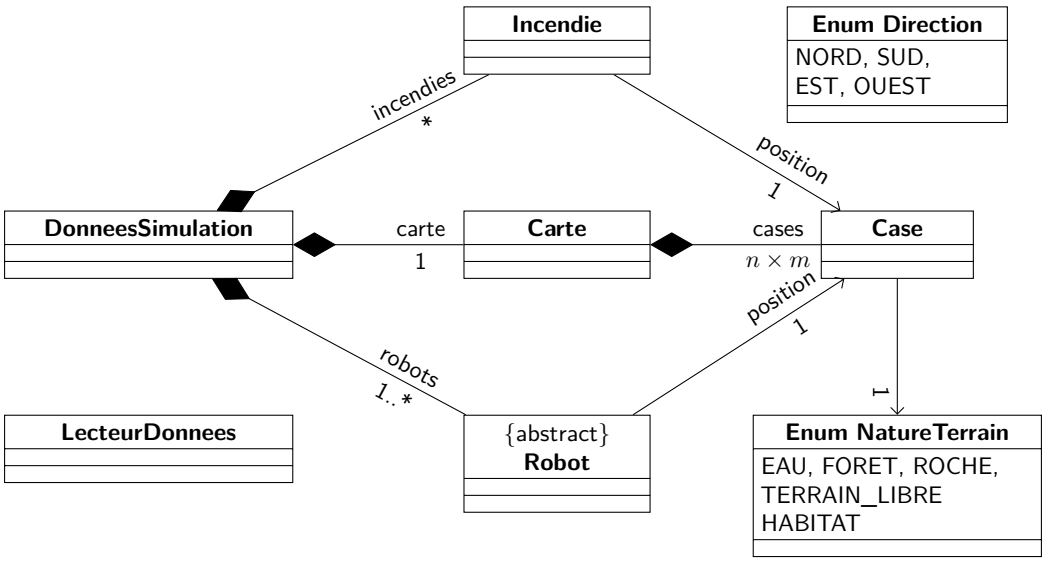


FIGURE 1 – Diagramme de classes des données du problème.

Rapidement, les classes sont spécifiées de la manière suivante :

- Une `Case` est simplement définie par deux coordonnées entières (ligne et colonne) et la nature du terrain qu'elle représente (un attribut de type énuméré `NatureTerrain`).
- Une `Carte` contient une matrice de `Case`, et la taille du côté des cases. Cette classe fournit notamment des méthodes pour accéder à une case en fonction de ses coordonnées, ou pour trouver le voisin d'une case dans une direction donnée (`Direction` est aussi un type énuméré : `NORD`, `SUD`, `EST`, `OUEST`).
- Un `Incendie` est simplement défini par sa position (une `Case`) et le nombre de litres d'eau nécessaires pour l'éteindre.
- Une classe `Robot` de haut niveau, abstraite, définit les propriétés et opérations communes à tous les robots. Elle devra être spécialisée pour représenter les différents types de robot. Les principales méthodes devront permettre :
  - d'accéder aux propriétés utiles d'un robot (sa position, le volume d'eau qu'il contient, *etc.*), si cela est nécessaire aux autres classes ;
  - de connaitre sa vitesse de déplacement en fonction de la nature d'un terrain. Le temps nécessaire pour se rendre d'une case à l'autre sera la moyenne de la vitesse sur chacune des cases multipliée par la taille des cases.
  - de le déplacer sur une case donnée, en vérifiant que cette case est bien accessible à ce robot (voisine et de nature compatible) ;
  - d'effectuer une intervention (déverser une quantité d'eau) sur la case où il se trouve ;
  - de remplir son réservoir, s'il est correctement positionné.
- La classe `DonneesSimulation` est la classe principale regroupant toutes les données du problème : une carte, les incendies et les robots.

2. *Unified Modeling Langage*. Une fiche de référence est disponible sur le `kiosk`.

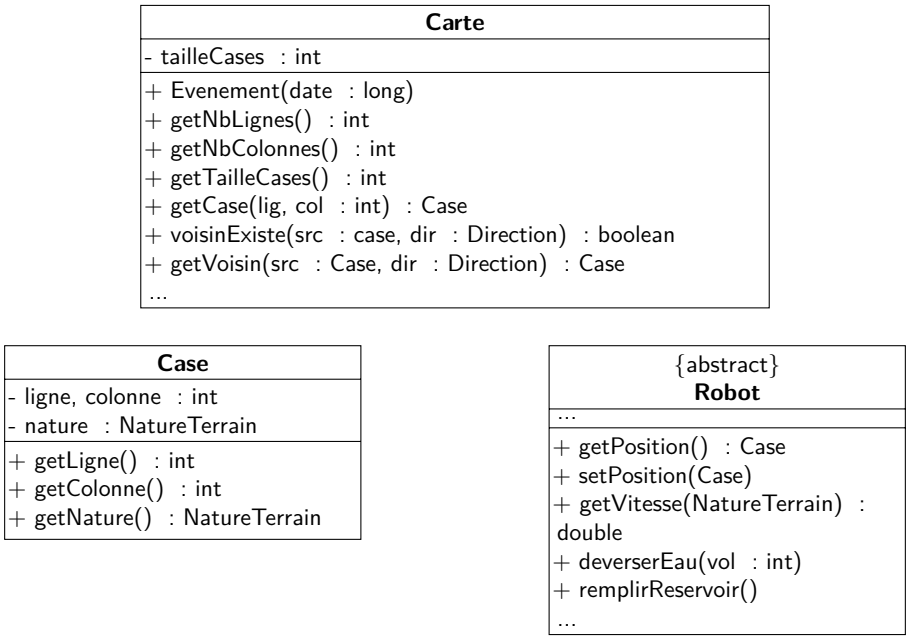


FIGURE 2 – Diagramme présentant les principaux attributs et méthodes des classes `Carte`, `Case` et `Robot`. Tout n'est pas spécifié ici, et vous pouvez bien sûr ajouter tous les éléments nécessaires à vos choix d'implémentation. Dans `Robot`, certaines méthodes peuvent être abstraites.

— La classe `LecteurDonnees` fournit une unique méthode (de classe) qui construit une instance de `DonneesSimulation` à partir d'un fichier texte au format décrit en annexe A. Une version simplifiée de cette classe est donnée par les enseignants, qui permet de lire puis afficher toutes les données. Vous devrez la modifier pour créer effectivement les différents objets, en fonction de la manière dont vous aurez écrit vos classes et constructeurs.

Les robots possèdent des propriétés communes et spécifiques. Il vous revient d'appliquer le processus de *généralisation* pour créer une *hiérarchie de classe* des robots, afin d'éviter la duplication inutile de code. Par contre c'est bien la classe `Robot` qui sera utilisée dans les algorithmes de haut niveau (calcul de chemins, simulateur, ...) grâce au processus de *polymorphisme* et de *liaison dynamique*.

Pour faciliter l'organisation de votre code et en augmenter la lisibilité, il est très fortement recommandé de regrouper l'ensemble des classes de données dans un même `package Java`.

1.3 Interface graphique et programme principal

Une interface graphique sommaire vous est fournie, au travers d'une classe `IGSimulateur` qui permet :

1. de créer et d'afficher une fenêtre graphique composé de $n \times m$ éléments carrés, qui peuvent être décorés avec une couleur, un texte ou une image. Ceci vous permettra d'afficher toutes les données du problème : les cases de la carte, les robots et les indendies.
2. de contrôler une simulation, via différents boutons : `Début`, `Lecture`, `Suivant`, `Quitter`.

La relation entre l'interface graphique et votre simulateur se fera avec le mécanisme d'héritage. A sa création, la classe `IGSimulateur` est associée à un objet de type `Simulable`, qui déclare deux méthodes `next()` et `restart()`. Ces méthodes sont automatiquement exécutées en réponse aux actions de l'utilisateur sur les boutons :

— `void next()` est invoquée suite à un clic sur le bouton `Suivant`, ou bien à intervalles réguliers si la lecture a été démarrée (le pas de temps entre deux évènements `next()` est paramétrable).

— `void restart()` est invoquée suite à un clic sur le bouton `Début`. La lecture est alors arrêté, et le simulateur doit revenir dans l'état initial (par exemple en relisant les données).

Pour utiliser cette interface, vous devez créer une classe (par exemple `Simulateur`) qui *réalise* l'interface `Simulable`, c'est-à-dire qui définit concrètement les méthodes de l'interface pour traiter les évènements de

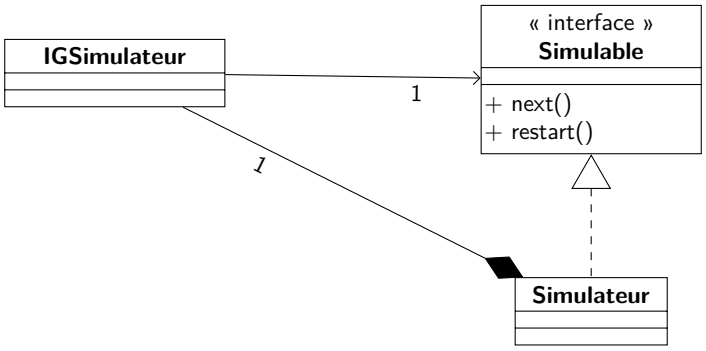


FIGURE 3 – Relation entre l'interface de simulation fournie et le simulateur

manière adéquate en fonction des données et de l'état de la simulation. La classe `Simulateur` possède donc un attribut de type `IGSimulateur`, auquel elle est associée pour réagir aux clics de l'utilisateur. Elle devra aussi utiliser cet objet `IGSimulateur` pour mettre à jour l'affichage en fonction de l'état des données.

Les figures de l'annexe C fournissent un exemple de fenêtre graphique et l'extrait de code correspondant à sa création, ainsi qu'un squelette de classe `Simulateur` permettant d'utiliser l'interface fournie.

L'interface graphique vous est fournie sous forme d'une archive `ihm.jar` contenant le *bytecode* Java des classes disponibles (mais nous n'avez pas accès au code source). La documentation (API) de ces classes, principalement `IGSimulateur` et `Simulable` sera disponible sur le `kiosk`.

Travail demandé

Le travail demandé pour cette première partie est d'implémenter toutes les classes décrites ci-dessus, ainsi qu'un programme de test qui charge un fichier de données et affiche la carte correspondante, les robots et les incendies à l'aide de l'interface graphique fournie.
A ce stade, les données initiales sont simplement affichées et votre simulateur ne fait rien en réponse aux évènements `next()` et `restart()` envoyés par l'interface graphique.

2 Deuxième partie : simulation de scénarios

La suite du travail consiste à compléter le projet pour pouvoir déplacer et faire intervenir des robots. Pour l'instant, le but est de simuler des scénarios définis à l'avance, par exemple déplacer un robot vers une case, puis une autre, puis verser de l'eau sur la case où il se trouve. Il s'agit en fait de mettre en place le cœur du simulateur, et de le tester avec des suites d'évènements prédéfinies.

2.1 Simulateur à évènements discrets

Même si d'autres solutions seraient possibles (par exemple avec des *threads* Java, chacun gérant les actions d'un robot), il est ici proposé de centraliser le problème en utilisant un *simulateur à événements discrets*. Ce simulateur possède une séquence ordonnée d'événements datés (par un entier par exemple, ou une classe `Date`). A chaque évènement est associée une action à réaliser : déplacer un robot sur une case, prévenir qu'un robot est arrivé, verser de l'eau, lancer le processus décisionnel compte tenu de la stratégie adoptée, *etc.* Le simulateur parcourt la liste d'événements dans l'ordre et exécute les opérations associées au fur et à mesure.

En pratique, le simulateur maintient une « date courante » de simulation. Lorsque la méthode `next()` est invoquée, en réponse aux actions de l'utilisateur sur l'interface graphique, une méthode `incrementeDate()` incrémente alors la date courante puis exécute dans l'ordre tous les évènements non encore exécutés jusqu'à cette date. `simulationTerminee()` retourne `true` si plus aucun évènement n'est en attente d'exécution.

Un diagramme de classes est proposé figure 4. La classe `Evenement` est abstraite, elle devra être héritée par des sous-classes qui représenteront des évènements réels avec leurs propres proriétés (par exemple un

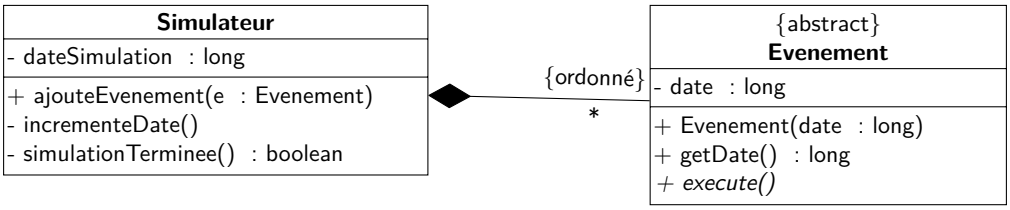


FIGURE 4 – Diagramme de classes d'un simulateur à événements discrets

robot et une case destination pour un évènement de déplacement) et définiront la méthode `execute()` de manière adéquate.

Un exemple d'utilisation d'un tel gestionnaire d'évènements discrets est présenté en annexe D.

2.2 Génération d'évènements : le *manager*

La question suivante est de savoir générer les évènements à simuler. Le simulateur délègue ceci à un *« manager »*, dont le seul rôle est de décider des évènements à exécuter, de les créer et de les ajouter au simulateur. Il peut aussi être intéressant de disposer de différents managers, par exemple pour tester différentes stratégies de répartition des robots sur les incendies afin d'éteindre le feu au plus vite.

Le mécanisme utilisé ici consiste à associer un attribut de type `Manager`, une classe *abstraite*, au `Simulateur`. Différentes stratégies de management pourront être implantées dans des classes concrètes filles de *Manager*. Du point de vue de l'application, le manager associé au simulateur peut être changé à volonté, et même dynamiquement [3]. La figure 5 présente les classes à mettre en œuvre pour l'application de ce mécanisme.

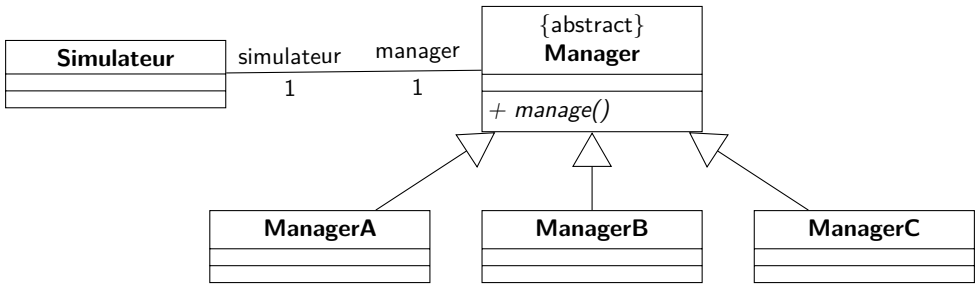


FIGURE 5 – Diagramme de classes de la hiérarchie des managers.

2.3 Scénarios de test

Une première fille concrète possible est `ManagerScenario`. Dans sa méthode `manage()` elle ajoute simplement des évènements *prédéfinis* au simulateur, à des dates contrôlées. Ceci est très utile pour tester le simulateur et le gestionnaire d'évènements, en vérifiant par exemple que des robots se déplacent correctement, aux bons instants, et que l'affichage est bien mis à jour.

Voici quelques exemples de tests possibles avec le fichier `carteSujet.txt` de l'annexe A :

3. Ce mécanisme est le patron de conception *Strategy*.

	Evènements et résultat
Scénario 0 (KO)	Déplacer le 1er robot (drone) vers le nord, quatre fois de suite. *Erreur : le robot est sorti de la carte.*
Scénario 1 (OK)	Déplacer le 2ième robot (à roues) vers le nord, en (5,5). Le faire intervenir sur la case où il se trouve. Déplacer le robot deux fois vers l'ouest. Remplir le robot. Déplacer le robot deux fois vers l'est. Le faire intervenir sur la case où il se trouve. *Le feu de la case en question doit alors être éteint.*

Travail demandé

Ajouter un gestionnaire d'évènements au simulateur, et créer un premier objet « manager » permettant de tester quelques scénarios (simples) prédéfinis. Les choses doivent commencer à bouger...

Important : il n'est pas demandé dans ce TP d'assurer une couverture de tests fonctionnels extensive, mais de simplement valider la fonctionalité générale de votre application (notamment les cas "qui marchent", comme le scénario 1 ci-dessus).

3 Troisième partie : calculs de plus courts chemins

Pour aller plus loin dans la simulation, il est nécessaire de savoir calculer les (meilleurs) itinéraires permettant à un robot de se rendre sur une case quelconque, plus forcément voisine, en fonction de ses propriétés et des caractéristiques du terrain. Cette partie est beaucoup moins guidée que précédemment. Voici néanmoins quelques éléments à prendre en compte :

- un robot doit être capable de calculer le plus court chemin, s'il existe, lui permettant de se rendre sur une case destination appartenant à la carte. Vous devez donc ajouter :
  - une méthode qui retourne le temps nécessaire à un robot pour se rendre sur une case donnée ;
  - une méthode qui dit à un robot de se rendre effectivement sur une case. Le robot devra alors programmer la série d'évènements de déplacements "élémentaires" (de case en case) correspondants au chemin optimal ;
  - pour faire ces actions, le robot devra avoir accès à la carte (puisque c'est elle qui permet de connaître les voisins d'une case) et au simulateur (pour pouvoir ajouter des évènements). Ils peuvent être passés en paramètres des méthodes le nécessitant, ou bien être connus du robot lui-même ;
  - ce travail peut aussi être délégué par le robot à une classe tierce. Ce modèle peut être intéressant en terme de séparation du code, ou pour facilement changer d'algorithme de calcul (là encore, le patron *stratégie* peut être utilisé).
- à vous de choisir un algorithme de calcul de chemin adéquat ; nombreuses solutions possibles !
- vous pourrez avoir besoin de représenter la notion de chemin, par exemple une suite de cases et les dates auxquelles elles sont atteintes.
- des optimisations sont possibles, pour la gestion du cache de calcul notamment : calculer des chemins vers plusieurs destinations, appels successifs alors que le robot ne s'est pas déplacé, *etc.* Ne les faites (éventuellement) qu'après avoir été au bout du TP. Il vous est d'abord demandé une version fonctionnelle, même simple et non optimale.

Dans toute cette partie vous aurez besoin de stocker des données intermédiaires, par exemple des coûts, cases à explorer, séquences de cases ou d'évènements, ... Les tableaux peuvent être utiles pour des données de taille fixe, mais il vous est surtout conseillé d'utiliser les *collections Java* qui fournissent des implémentations efficaces de nombreuses structures de données : `List`, `HashSet`, `PriorityQueue`, *etc.*

Un document de présentation des principales collections Java et de leur utilisation est disponible sur le `kiosk`.

Travail demandé

Mettre en place les classes permettant de calculer le plus court chemin d'un robot vers une destination, et de le traduire en évènements de déplacement à ajouter au simulateur.
Utiliser les collections Java.
Pout tester, vous pouvez simplement créer une nouvelle classe `ManagerTestChemin` qui essaie d'envoyer un robot sur une case déterminée.

4 Dernière partie : résolution du problème

La dernière partie consiste à définir une stratégie d'affectation des tâches, c'est-à-dire organiser les déplacements et interventions des différents robots afin d'éteindre tous les incendies au plus vite.

Le principe repose ici sur l'utilisation d'un manager central, qui peut échanger des messages avec les divers agents en présence et prend les décisions stratégiques [4]. Dans notre contexte, les robots sont des agents ayant une certaine autonomie, notamment pour trouver leur chemin sur la carte. Le manager peut lui être vu comme un « chef pompier ». Il connaît l'intégralité de la carte, la position des incendies, peut interroger les robots et décide de l'affectation des robots. Il n'a ici qu'un rôle décisionnel.

Le protocole général est le suivant. Le manager envoie une demande à des robots. Cette demande peut être par exemple une spécification de tâche à réaliser. Chaque robot contacté soit refuse la demande, soit envoie une proposition, correspondant par exemple au temps qu'il compte mettre pour remplir la tâche demandée par le manager. Le manager sélectionne ensuite les différentes tâches à affecter aux robots, qui les exécutent.

4.1 Stratégie élémentaire

Une première stratégie, simpliste, consiste à envoyer des robots intervenir sur des incendies sans recherche d'optimisation du processus global. Par exemple, tous les n pas de temps :

1. Le manager propose un incendie non affecté (peu importe lequel) à un robot (peu importe lequel).
2. Si le robot contacté est occupé (à se déplacer, à éteindre un incendie ou à recharger son réservoir en eau) il refuse la proposition. Sinon, il cherche un chemin pour se rendre verts l'incendie. S'il n'en trouve pas, il refuse la proposition.

   Dans ces cas, le manager contacte alors un autre robot.
3. Le robot sélectionné programme la séquence d'évènements nécessaires pour réaliser son déplacement. Arrivé sur place, il verse son eau.
4. Si le réservoir d'un robot est vide, celui ne peut plus être utilisé. Il reste « occupé » vis à vis du manager, et refuse toutes les nouvelles demandes du manager.
5. Le manager recommence les étapes 1, 2 et 3 sur un autre incendie non affecté – et ainsi de suite, jusqu'à avoir parcouru tous les incendies et le cas échéant tous les robots.

4.2 Stratégie un peu plus évoluée

Une approche un peu plus avancée pourrait être :

1. Le manager propose à tous les robots un incendie à éteindre ;
2. Les robots occupés refusent la proposition, les autres robots calculent leur plus court chemin pour y arriver et renvoient le temps nécessaire au manager.
3. Le manager sélectionne le robot le plus proche pour aller éteindre l'incendie.

   Celui-ci programme alors la séquence d'évènements nécessaires pour réaliser son déplacement. Arrivé sur place, il peut vérifier que le feu n'a pas été éteint entre temps avant de verser son eau. Lorsque que le réservoir d'un robot est vide, celui-ci peut de lui-même aller se remplir au point d'eau (ou la berge) le plus proche.

4. Ceci s'inspire du protocole *Contract Net*, de R.G. Smith : *The Contract Net Protocol : High-Level Communication and Control in a Distributed Problem Solver*, IEEE Transactions on Computers, C-29(12) :1104-1113, dec. 1980.

4. Le manager peut demander ceci pour chaque incendie. Si certains restent non affectés, le manager attend un certain laps de temps et repropose les incendies restants.

   D'autre solutions sont possibles, par exemple le manager demande à tous les robot les temps pour aller sur tous les incendies puis envoie chaque robot sur l'incendie le plus proche.

Des versions encore plus fines pourraient prendre en compte l'intensité des incendies, les capacités des réservoirs des robots, les positions des points d'eau par rapport aux incendies, *etc.* [5]

Dans un cadre réel, il s'agirait naturellement de trouver une stratégie la plus efficace possible. L'étude des problèmes de décision dans un contexte incertain n'est cependant pas l'objectif de ce projet, et nous nous contenterons de stratégie(s) beaucoup plus simple(s). Votre but est d'avoir une simulation qui s'exécute et d'arriver si possible à éteindre les feux.

Travail demandé

Créer une nouvelle sous-classe de `Manager` qui organise les différentes activités des robots pour éteindre tous les incendies de la carte.
Objectif minimal : votre objectif pour ce TP est avant tout de mettre en œuvre *une* première stratégie de répartition des robots, même très simple. Si besoin, augmentez la taille des réservoirs des robots.
Optionnel : en fonction du temps disponible, implantez une stratégie plus évoluée.
Comme précédemment, des données intermédiaires pourraient être stockées par les différents objets, par exemple l'ensemble des cases contenant de l'eau, *etc.* Là encore, utilisez les collections `Java`.

5 Livrable attendu

L'application rendue devra répondre aux spécifications des différentes parties ci-dessus. Si toutes les contraintes ne sont pas prises en compte, bien le spécifier dans le rapport. En plus de ceci, quelques exigences non fonctionnelles sont attendues :

- le code rendu devra être propre, bien structuré, et respecter le *coding style* Java (voir le lien sur le `kiosk`... et tout ce qui est fait en cours !) ;
- en plus de noms de variables explicites, les aspects « techniques » de votre code devront être commentés, pour en faciliter la compréhension ;
- l'API des classes doit aussi être renseignée, avec les tags `/** ... */` utilisés par `javadoc` pour générer la documentation en `html` ou `pdf`. Il n'est pas nécessaire de s'étendre plus que de raison sur une méthode type `getNbLignes()`. Par contre la documentation devra être adaptée pour expliquer le comportement ou l'utilisation des classes ou méthodes plus avancées ;
- le principe d'encapsulation devra être respecté : masquage des attributs, garantie de l'intégrité des états des objets, principe de délégation ;
- utilisez l'héritage pour factoriser tout code nécessaire à plusieurs objets, et spécifiez des méthodes abstraites dans les classes de haut niveau [6] ;
- même s'il n'est pas demandé de tests exhaustifs, votre application devra être le plus robuste possible. Vous pourrez à cet effet utiliser le mécanisme d'exceptions.

Le livrable final sera transmis sous la forme d'une archive `tar.gz` contenant :

- le code source de votre application ;
- un document au format `pdf` de 4 pages maximum expliquant et justifiant vos choix de conception, l'utilisation à bon escient des classes et des méthodes les plus adaptées ;
- un second document `pdf` décrivant rapidement les tests effectués et les principaux résultats obtenus.

La date limite de rendu est fixée au **jeudi 20 novembre 2014, 19h**.

Bon travail à vous tous !

5. Il serait également possible de gérer aussi la propagation des incendies sur la carte. Ce n'est pas demandé, mais ce serait très facile avec la structure proposée : il suffit de faire un « manager » qui modélise aussi le comportement des incendies au cours du temps. De jolies « batailles » en perspectives, pour voir quelles stratégies sont gagnantes... ou pas !
6. si vous avez à utiliser `instanceof` en dehors de la redéfinition d'une méthode `equals(Object o)`, il y a généralement un problème de conception objet...

Annexes

A Format de description des conditions initiales

Un exemple de format de description des données dans un fichier texte est proposé ci-dessous. Il est recommandé de le conserver pour faciliter les tests sur des données communes.

```
# Un exemple de carte 8x8
8 8 10000

# l0
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
# l1
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
ROCHE
TERRAIN_LIBRE
ROCHE
TERRAIN_LIBRE
# l2
TERRAIN_LIBRE
TERRAIN_LIBRE
EAU
EAU
TERRAIN_LIBRE
FORET
TERRAIN_LIBRE
TERRAIN_LIBRE
...
...
# l3
TERRAIN_LIBRE
TERRAIN_LIBRE
EAU
EAU
TERRAIN_LIBRE
FORET
FORET
TERRAIN_LIBRE
# l4
TERRAIN_LIBRE
TERRAIN_LIBRE
EAU
TERRAIN_LIBRE
TERRAIN_LIBRE
FORET
FORET
TERRAIN_LIBRE
# l5
TERRAIN_LIBRE
TERRAIN_LIBRE
EAU
TERRAIN_LIBRE
TERRAIN_LIBRE
HABITAT
HABITAT
TERRAIN_LIBRE
# l6
TERRAIN_LIBRE
TERRAIN_LIBRE
...
...
EAU
EAU
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
# l7
TERRAIN_LIBRE
HABITAT
HABITAT
EAU
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE
TERRAIN_LIBRE

# Incendies
6
7 0 50000
7 1 10000
6 0 10000
6 1 20000
5 5 8000
7 7 5000

# Robots
3
3 3 DRONE 150
6 5 ROUES
4 7 PATTES
```

FIGURE 6 – Format des données du simulateur. Ici un exemple de carte 8×8, avec des cases de 10000 (mètres ?) de côté, 3 robots et 6 incendies. Tout ce qui suit un caractère # est un commentaire, et les lignes blanches sont ignorées.

B Propriétés des robots

B.1 Déplacement des robots

Type de robot	Propriétés de déplacement sur une case
Drone	Vitesse par défault de 100 km/h, mais peut être lue dans le fichier de données (sans dépasser 150 km/h) Peut se déplacer sur toutes les cases, quelle que soit leur nature, à vitesse constante.
Robot à roues	Vitesse par défaut de 80 km/h, mais qui peut être lue dans le fichier. Ne peut se déplacer que sur du terrain libre ou habitat.
Robot à chenilles	Vitesse par défaut de 60 km/h, mais qui peut être lue dans le fichier (sans dépasser 80 km/h) La vitesse est diminuée de 50% en forêt. Ne peut pas se rendre sur de l'eau ou du rocher.
Robot à pattes	Vitesse de base de 30 km/h, réduite à 10 km/h sur du rocher. Ne peut pas se rendre sur de l'eau.

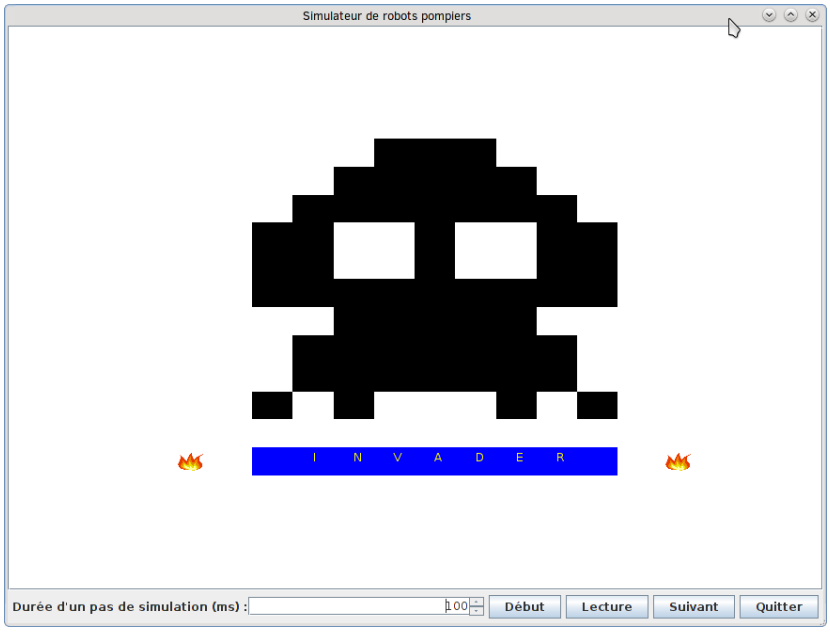
B.2 Capacités d'extinction des robots

Type de robot	Réservoir et remplissage	Extinction
Drone	Réservoir de 10000 litres. Remplissage complet en 30 minutes. Se remplit sur une case contenant de l'eau.	Intervention unitaire : vide la totalité du réservoir en 30 secondes.
Robot à roue	Réservoir de 5000 litres. Remplissage complet en 10 minutes. Se remplit à côté d'une case contenant de l'eau.	Intervention unitaire : 100 litres en 5 sec.
Robot à chenille	Réservoir de 2000 litres. Remplissage complet en 5 minutes. Se remplit à côté d'une case contenant de l'eau.	Intervention unitaire : 100 litres en 8 sec.
Robot à pattes	Utilise de la poudre. Réservoir considéré infini à l'échelle de la simulation. Ne se remplit jamais.	Intervention unitaire : 10 litres en 1 sec.

C Interface graphique

Cette annexe présente un exemple de classe `Invaders` utilisant l'IHM fournie. C'est à cette classe qu'il revient de créer l'instance de `IGSimulateur` qui servira à afficher ses données.

La classe réalise l'interface `Simulable`, et donc doit implémenter les méthodes `next()` et `restart()` recevant les signaux de l'utilisateur. Dans cet exemple, il n'y a pas de modification des données et l'affichage est toujours identique. Dans votre simulateur, `next()` devra bien entendu mettre les données à jour (incrémenter la simulation « d'un pas de temps ») et `restart()` revenir à l'état initial. Dans les deux cas, l'affichage doit être mise à jour pour refléter le nouvel état des données.




```java
import ihm.Simulable;
import ihm.gui.*;
import java.awt.Color;

public class TestIHM {
    public static void main(String[] args) {
        Invaders invaders = new Invaders();
    }
}

class Invaders implements Simulable {
    ...                         // les donnees eventuelles
    private IGSimulateur ihm;  // l'IHM associee a ce simulateur

    public Invaders() {
        ...                     // init des donnees

        // cree l'IHM et l'associe a ce simulateur (this), qui en tant que
        // Simulable recevra les evenements next() et restart()
        // suite aux clics de l'utilisateur
        ihm = new IGSimulateur(20, 20, this);
        dessine();    // mettre a jour l'affichage
    }

    @Override
    public void next() {
        System.out.println("TODO :
            avancer la simulation \"d'un pas de temps\"");
```

```
29          System.out.println("  => faire descendre les invaders !");
30          dessine();    // mettre a jour l'affichage
31      }
32
33      @Override
34      public void restart() {
35          System.out.println("TODO :
36              remettre le simulateur dans son état initial");
37          dessine();    // mettre a jour l'affichage
38      }
39
40
41      private void dessine() {
42          try {
43              ihm.paintBox(9, 4, Color.BLACK);
44              /* [...] Ici il y a le reste du dessin qu'il serait
45              fastidieux de mettre en entier */
46              ihm.paintBox(14, 13, Color.BLACK);
47
48              for (int i = 6; i < 15; i++) {
49                  ihm.paintBox(i, 15, Color.BLUE);
50              }
51              ihm.paintImage(4, 15, "images/feu.png", 0.8, 0.8);
52              ihm.paintString(7, 15, Color.YELLOW, "I");
53              ihm.paintString(8, 15, Color.YELLOW, "N");
54              ihm.paintString(9, 15, Color.YELLOW, "V");
55              ihm.paintString(10, 15, Color.YELLOW, "A");
56              ihm.paintString(11, 15, Color.YELLOW, "D");
57              ihm.paintString(12, 15, Color.YELLOW, "E");
58              ihm.paintString(13, 15, Color.YELLOW, "R");
59              ihm.paintImage(16, 15, "images/feu.png", 0.8, 0.8);
60              ihm.paintBox(14, 13, Color.BLACK);
61
62          } catch (MapIndexOutOfBoundsException e) {
63              e.printStackTrace();
64          }
65      }
66  }
```

D Gestionnaire d'évènements discrets

Les figures suivantes présentent un exemple de classe d'évènement, son utilisation dans un simulateur à évènements discrets et la trace résultante.

```
class EvenementMessage extends Evenement {
    private String message;

    public EvenementMessage(int date, String message) {
        super(date);
        this.message = message;
    }

    public void execute() {
        System.out.println(this.getDate() + this.message);
    }
}
```

FIGURE 7 – Exemple de classe représentant un événement héritant le modèle `Evenement` présenté figure 4. Ici il ne s'agit que d'afficher un message dans la console.

```
public class Test {
    public static void main(String[] args) {
        // On crée un simulateur
        Simulateur simulateur = new Simulateur(...);

        // On poste un événement [PING] tous les deux pas de temps
        for (int i = 2; i <= 10; i += 2) {
            simulateur.ajouteEvenement(new EvenementMessage(i, " [PING]"));
        }
        // On poste un événement [PONG] tous les trois pas de temps
        for (int i = 3; i <= 9; i += 3) {
            simulateur.ajouteEvenement(new EvenementMessage(i, " [PONG]"));
        }

        // et on suppose que la simulation démarre
        ...
    }
}
```

FIGURE 8 – Un exemple de code illustrant le fonctionnement du simulateur à l'aide d'un scénario fixé à l'avance (ici l'ajout d'événement « `[PING]` » tous les deux pas de temps, et d'un « `[PONG]` » tous les trois pas de temps)

```
Next... Current date : 1
Next... Current date : 2
2 [PING]
Next... Current date : 3
3 [PONG]
Next... Current date : 4
4 [PING]
Next... Current date : 5
Next... Current date : 6
6 [PING]
6 [PONG]
Next... Current date : 7
Next... Current date : 8
8 [PING]
Next... Current date : 9
9 [PONG]
Next... Current date : 10
10 [PING]
```

FIGURE 9 – Trace d'exécution de la simulation spécifiée dans le code de la figure 8.