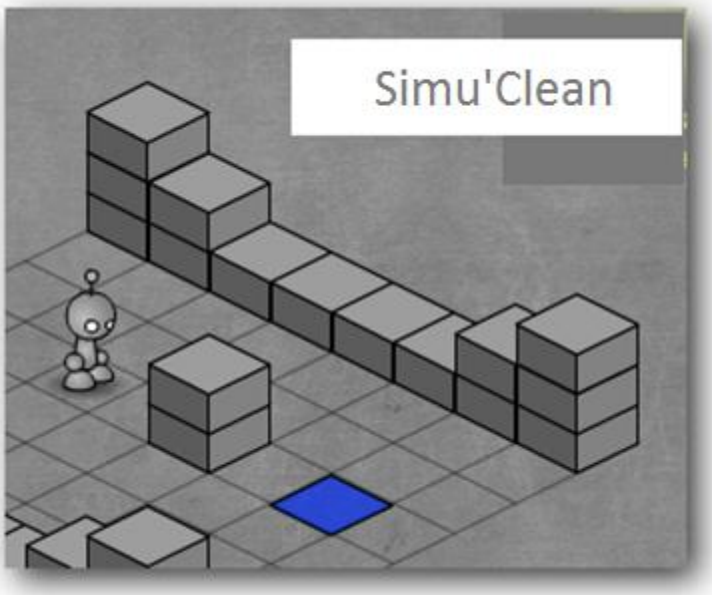
POLYTECH GRENOBLE

2010

Manuel développeur



BARRADOUANE Ilham
BARRIAL Geoffrey
DEWULF Mathieu
DUPESSEY Xavier
El BAKKOURI Nysrine
ODUL Jonathan

Table des matières

Introduction

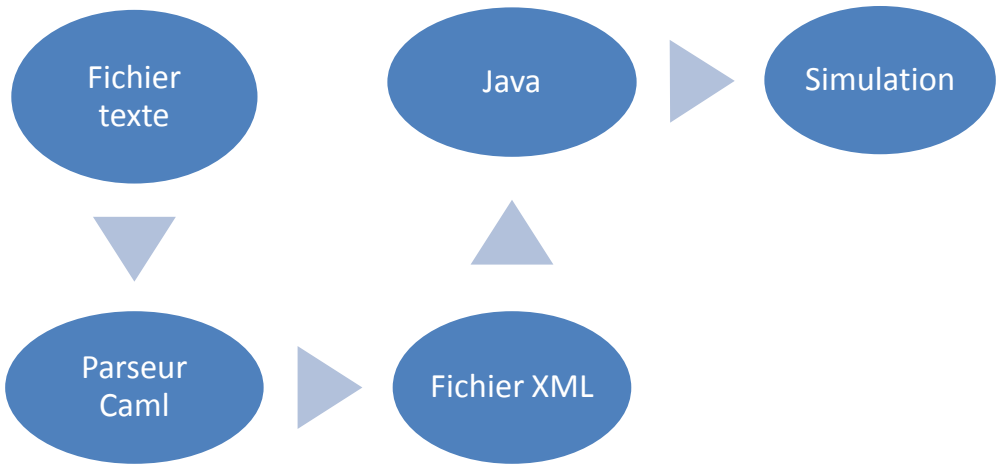
Ce manuel donne des informations de développement aux futurs développeurs du projet Simu'Clean. Ces informations sont en parties utiles pour la mise en place des conventions de codages utilisées dans ce projet.

Le logiciel Simu'Clean a pour but de simuler le comportement de robots autonomes et plus particulièrement des aspirateurs intelligents pour nettoyer une pièce.

Pour ce faire l'utilisateur entre des données dans un fichier texte afin de composer la pièce que l'on voudra nettoyer.

Par la suite l'évolution du comportement de ces robots se fera à partir d'un algorithme implémenté en Java. Elle pourra être visualisée par l'utilisateur via une interface graphique.

L'architecture globale de notre programme est décrite par le graphe suivant :



Partie 1 : Analyseur

I. Structure du fichier texte

Un fichier texte doit être rempli par l'utilisateur, respectant la grammaire définie dans la partie II.1, intitulée « Grammaire du format d'entrée »

Le fichier texte d'entrée doit vérifier les instructions suivantes :

1. Avoir le caractère « # » suivi du nom de la salle puis de ses dimensions
2. Avoir des points virgules à la fin de chaque ligne
3. Avoir un caractère de fin « ~ »

Le fichier d'entrée est un fichier texte au format suivant :

```
# nom_salle dimension_x dimension_y;
Mur(coordonnees_x1,coordonnees_y1,coordonnees_x2,coordonnees_y2);
Mur(coordonnees_x1,coordonnees_y1,coordonnees_x2,coordonnees_y2);
Mur(coordonnees_x1,coordonnees_y1,coordonnees_x2,coordonnees_y2);
Mur(coordonnees_x1,coordonnees_y1,coordonnees_x2,coordonnees_y2);
RobotExplorateur(nom_robot_Explorateur,coordonnees_x,coordonnees_y,vitesse,direction,adresseIP,batteriemax,
batterieactuelle,
masse,coordonnees_x_base,coordonnees_y_base);
BaseRobotExplorateur(coordonnees_x,coordonnees_y);
Sol(coordonnees_x,coordonnees_y,taux_salete,nom_objet,direction);
Machine(coordonnees_x,coordonnees_y);
RobotAspirateur(nom_robot_Aspirateur,coordonnees_x,coordonnees_y,vitesse,direction,adresseIP,batteriemax,
batterieactuelle, masse,bacmax,bacactuelle,coordonnees_x_base,coordonnees_y_base);
BaseRobotAspirateur(coordonnees_x,coordonnees_y);
~
```

Où « nom_salle », « nom_robot_Aspirateur », « nom_robot_Explorateur », « nom_objet » sont respectivement les noms de la salle, du robot aspirateur, du robot explorateur et nom de l'objet dynamique à choisir par l'utilisateur. Les « coordonnees_x », « coordonnees_y », « coordonnees_x_base », « coordonnees_y_base », « coordonnees_x1 », « coordonnees_y1», « coordonnees_x2 », « coordonnees_y2 », « dimension_x » , « dimension_x », adresseIP », « batteriemax », « batterieactuelle », « masse », « taux_salete » sont des entiers et « vitesse », « direction », « bacmax » , « bacactuelle » sont des float.

Les mots « Mur », «BaseRobotExplorateur », « RobotExplorateur », « RobotAspirateur », « BaseRobotAspirateur » , « Sol », « Machine » doivent être écrits ainsi.

Ci-dessous un exemple de fichier d'entrée simple :

```
# TestDeBase 7 7;
Mur(0,0,0,6);
Mur(0,0,6,0);
Mur(6,0,6,6);
Mur(0,6,6,6);
RobotExplorateur(Nono,1,1,1.,DROITE,42,1200,600,1000,1,1);
BaseRobotExplorateur(1,1);
Sol(3,1,100,Pingouin,BAS);
Machine(3,3);
RobotAspirateur(XiouXiou,4,3,1.,HAUT,123,800,800,1000,2.,1.,4,4);
BaseRobotAspirateur(4,4);
~
```

II. Réalisation du lexeur

Le fichier texte d'entrée est parsé par notre programme en langage Ocaml :

- Analyse du fichier texte pour le transformer en un flot de lexèmes reconnus.
- Analyse du flot de lexèmes pour la vérification syntaxique.
- Génération de messages d'erreur précis pour une correction rapide et efficace.

1. Grammaire du format d'entrée

```
type typ = string
type nom = string
type salete = int
type direction = string
type vitesse = float
type adresse = int
type batteriemax = int
type batterieactuelle = int
type masse = int
type bacmax = float
type bacact = float
type coordxbase = int
type coordybase = int
type lexeme = PARO | PARF | LIGNE | VIRG | PVIR | EOF | INIT | STR of string | ENT of int | FLOAT of float
type noeudsol = int*int*int*int*salete
type noeudsolperso = int*int*salete*nom*direction
type noeudsolalea = int*int*int*int*salete*salete
type noeudobstacle = int*int*int*int
type noeudobstacledynamique = nom * int * int * vitesse  * direction
type noeudrobote = nom * int * int * vitesse  * direction * adresse * batteriemax * batterieactuelle * masse * coordxbase * coordybase
type noeudrobota = nom * int * int * vitesse  * direction * adresse * batteriemax * batterieactuelle * masse * bacmax * bacact * coordxbase *
coordybase
type noeudbaserobote = int *int
type noeudbaserobota = int *int
type noeudtable = typ*nom*salete*vitesse*direction*adresse*batteriemax*batterieactuelle*masse*bacmax*bacact*coordxbase*coordybase
type syntaxe = ObjetFixe of noeudobstacle | Personne of noeudobstacledynamique | BaseRobotExplorateur of noeudbaserobote | BaseRobotAspirateur of
noeudbaserobota | RobotExplorateur of noeudrobote | RobotAspirateur of noeudrobota | Sol of noeudsol | Solbis of noeudsolperso | Solter of
noeudsolalea | Piece of piece | Eof
```

Où :

- le type typ est le type du nœud
- le type nom est le nom du robot ou de la personne
- le type salete est le taux de saleté variant de 0 à 1000
- le type direction est soit GAUCHE ou DROITE ou BAS ou alors HAUT
- le type vitesse est une valeur variant de 0.1 à 1.
- le type adresse est l'adresse IP des robots de 0 à 255
- le type batteriemax est une valeur entre 100 et 7000
- le type batterieactuelle est une valeur entre 0 et la valeur de batteriemax
- le type masse est la masse du robot en grammes entre 50 et 3000
- le type bacmax est une valeur entre 0.2 et 4.
- le type bacact est une valeur entre 0. et la valeur de bacmax
- le type coordxbase est la coordonnée x de la base
- le type coordybase est la coordonnée y de la base
- le type lexeme est soit une parenthèse ouvrante ou parenthèse fermante ou une ligne ou une virgule ou un point virgule ou un end of file ou un caractère de début "#" ou un string ou un entier ou un float
- le type noeudsol est coordonnée x1* coordonnée y1 * coordonnée x2* coordonnée y2 * taux de saleté
- le type noeudsolperso est coordonnée x* coordonnée y * taux de saleté * nom de la personne * direction de la personne
- le type noeudsolalea est coordonnée x1* coordonnée y1 * coordonnée x2* coordonnée y2* taux de saleté minimum *taux de saleté maximum
- le type noeudobstacle est coordonnée x1* coordonnée y1 * coordonnée x2* coordonnée y2
- le type noeudobstacledynamique est nom de l'obstacle* coordonnée x* coordonnée y* vitesse * direction
- le type noeudrobote est nom du robot explorateur * sa coordonnée x ,* sa coordonnée y* sa vitesse *sa direction *son adresse *sa batterie maximum *sa batterie actuelle * sa masse * coordonnée x de sa base * coordonnée y de sa base
- le type noeudrobota est nom du robot aspirateur * sa coordonnée x * sa coordonnée y * sa vitesse *sa direction * son adresse * sa batterie maximum * sa batterie actuelle * sa masse * coordonnée x de sa base * coordonnée y de sa base
- le type noeudbaserobote est coordonnée x de la base robot explorateur* coordonnée y de la base robot explorateur

- le type noeudbaserobota est coordonnée x de la base robot aspirateur* coordonnée y de la base robot aspirateur
- le type noeudtable est type du nœud * nom de la personne ou du robot * taux de saleté du nœud * vitesse* direction de la personne ou du robot* adresse * batterie maximum *batterie actuelle*masse * maximum du bac * coordonnée x de la base du robot *coordonnée y de la base du robot

2. Flot de lexème

A partir du fichier texte d'entrée transformée en flots, il est nécessaire de créer un flot de lexème à partir de notre grammaire. Pour cela la fonction « parseur » est à votre disposition. Celle-ci permet de repérer un lexème, afin de le mémoriser dans un flot. Cette fonction vérifie que la suite de lexème crée précédemment respecte la syntaxe voulue. La fonction « parseur » permet de reconnaître l'ensemble des expressions sous forme rationnelle pour ensuite les rajouter à la table de nœuds.

3. Vérifications

a) Les fonctions de vérifications grammaticales

La fonction « lexeur » permet de faire les **vérifications lexicales** pour s'assurer que le fichier en entrée est correctement écrit et correspond à des éléments de la grammaire.

```
val lexeur : char Stream.t -> lexeme Stream.t = <fun>
```

Les fonctions « parseur » et « parseur2 » s'occupent des **vérifications syntaxiques**, afin qu'il n'y a pas de caractères parasites et nous renvoie l'erreur qui correspond si nécessaire.

La première fonction vérifie la syntaxe de la déclaration du nom de la pièce du fichier donné en entrée, la deuxième fonction vérifie la syntaxe après la déclaration du nom de la pièce du fichier donné en entrée.

```
Val parseur : lexeme Stream.t ->syntaxe Stream.t = <fun>

Val parseur2 :  lexeme Stream.t ->syntaxe Stream.t = <fun>
```

b) Les fonctions de vérifications contextuelles

Afin de vérifier que les valeurs entrées ont du sens et sont conformes, nous avons réalisé la fonction « verification »vérifiant si les valeurs suivantes entrées en attributs sont cohérents :

- Les coordonnées x et y
- Le taux de saleté
- La vitesse
- La direction
- L'adresse des robots
- La batterie des robots
- La masse des robots
- La taille du bac des robots

III. Ecriture du fichier XML

La fonction « generation_xml » permet d'écrire dans le fichier les informations contenues dans les structures créées.

Val generation_xml : string -> unit = <fun>

Cette fonction va donc charger le fichier texte en mémoire, le transformer en flot de caractère, et faire un flot de string. On va alors transformer cela en flot de lexème afin de pouvoir procéder à toutes les vérifications. Si celles si sont concluantes, on écrira alors notre fichier final XML.

Extrait d'un fichier XML généré :

```
<Piece name="TestDeBase">
	<Noeud Type="MUR">
		<Coordx Coordx="0" />
		<Coordy Coordy="0" />
	</Noeud>
	<Noeud Type="SOL">
		<Coordx Coordx="4" />
		<Coordy Coordy="3" />
		<Salete Taux="0" />
		<Robot Robot="ROBOTRA" />
		<Nom Nom="XiouXiou" />
		<Vitesse Vitesse="1." />
		<Direction Direction="HAUT" />
		<Adresse Adresse="123" />
		<BatterieMax BatterieMax="800" />
		<BatterieAct BatterieAct="800" />
		<Masse Masse="1000" />
		<BacMax CapaciteMax="2." />
		<BacAct CapaciteAct="1." />
		<CoordxBase Coordx="4" />
		<CoordyBase Coordy="4" />
	</Noeud>
	<Noeud Type="BASERA">
		<Coordx Coordx="4" />
		<Coordy Coordy="4" />
	</Noeud>
	<Noeud Type="MUR">
		<Coordx Coordx="2" />
		<Coordy Coordy="0" />
	</Noeud>
	<Noeud Type="SOL">
		<Coordx Coordx="2" />
		<Coordy Coordy="1" />
		<Salete Taux="0" />
	</Noeud>
	<Noeud Type="SOL">
		<Coordx Coordx="2" />
		<Coordy Coordy="2" />
		<Salete Taux="0" />
	</Noeud>
</Piece>
```

Partie 2 : Organisation du code

Dans cette partie, nous allons détailler la structure du code. Nous expliquerons le rôle des différents packages et classes dans le fonctionnement final de l'application.

Notre code est organisé autour de quatre packages:

- **Simulateur :** gère l'interface avec l'utilisateur.
- **simulateur.chargeur :** permet de décoder le fichier xml généré par la partie caml.
- **simulateur.donneesGraphiques :** gère l'interface avec l'utilisateur.
- **simulateur.donneesSimulation :** gère le comportement des robots et des obstacles dynamiques dans la pièce.

Package simulateur

a) classe Interface.java

Cette classe permet de créer notre interface graphique. On s'est servi de la librairie graphique swing. On a défini différents panels grâce à un BorderLayout dans notre frame afin de positionner les éléments et les informations facilement.

b) classe Simulator.java

Cette classe permet de définir toutes les méthodes permettant l'exécution de la simulation.

c) classe SimulatorViewer.java

Cette classe permet d'afficher les éléments dessinables sur l'interface graphique.

Package simulateur.chargeur

a) classe DecodageXML.java

Cette classe prend les informations du fichier XML et les rend accessibles à java.

b) classe Element.java

Un élément peut être soit un mur, soit un sol, soit une base robot aspirateur, soit une base robot explorateur.

c) classe Voisin.java

Cette classe permet de calculer les noeuds voisins.

d) classe XML.java

Cette classe permet de parser un fichier XML créé dans la phase Caml.

Package simulateur.donneesGraphiques

- classe D_BaseRobot.java

Cette classe permet de dessiner une base de robot.

- classe D_Noeud.java

Cette classe permet de dessiner un noeud.

- classe D_Obstacle.java

Cette clase permet de dessiner un obstacle.

- classe D_Personne.java

Cette classe permet de dessiner une personne.

- classe D_RobotAspirateur.java

Cette classe permet de dessiner un robot aspirateur.

- classe D_RobotExplorateur.java

Cette classe permet de dessiner un robot explorateur.

- classe D_Sol.java

Cette classe permet de dessiner un sol.

- classe Dessinable.java

Cette classe permet de dessiner tous les éléments pouvant être affichées sur l'interface.

Package simulateur.donneesGraphiques.ressources

Ce package contient toutes les ressources graphiques utiles pour notre interface.

Package simulateur.donneesSimulation

- classe Direction.java

Cette classe permet d'énumérer toutes les directions possibles pour un noeud

(HAUT, BAS, GAUCHE, DROITE)

- classe Printable.java

Cette classe permet d'afficher tous les éléments affichables par le simulateur.

- classe Simulable.java

Cette classe permet de simuler tous les éléments simulables par le simulateur.

Package simulateur.donneesSimulation.objets.composants

- classe Bac.java

Chaque robot aspirateur contient un bac. Cette classe contient tous les paramètres qui définissent un bac de robot comme sa capacité totale et la capacité disponible restante en litres.

- classe Batterie.java

Chaque robot a besoin d'une batterie d'alimentaion. Cette classe définit tous les paramètres qui définissent la batterie d'un robot comme sa capacité totale et la capacité disponible restante.

Package simulateur.donneesSimulation.objets.taches

- classe ZoneAnettoyer.java

Le robot explorateur doit localiser les zones sales afin de communiquer leur position aux robots aspirateurs qui se chargeront de les nettoyer.

Package simulateur.donneesSimulation.reseau

- classe MediumCommunication.java

Le signal emis par les robot circule sur le sol .Donc, notre medium de communication est le sol.

- classe Message.java

Cette classe énumère toutes les données définissant un message:

(DEBUT_DIALOGUE, FIN_DIALOGUE, POSITION_COURANTE, ACCEPTER, REFUSER,

CARTE.)

- classe ObjetCommuniquant.java

Un objet communiquant peut être soit un robot aspirateur, soit un robot explorateur.

- classe Signal.java

Cette classe définit le signal pouvant être émis et reçu par les robots.

Package simulateur.donneesSimulation.terrain

- classe BaseRobot.java

La base d'un robot est un noeud sur lequel le robot peut revenir pour se recharger.

- classe BaseRobotAspirateur.java

La base d'un robot aspirateur est un noeud sur lequel le robot aspirateur peut revenir pour se recharger.

- classe BaseRobotExplorateur.java

La base d'un robot explorateur est un noeud sur lequel le robot explorateur peut revenir pour se recharger.

- classe Noeud.java

Le sol de la pièce est défini comme un ensemble de noeuds. Chaque noeud peut être accessible par ses coordonnées, ou par le noeud voisin.

- classe Obstacle.java

Un obstacle peut être soit un mur, soit un obstacle au milieu de la pièce.

- classe Sol.java

un sol est un ensemble de noeuds délimités par des murs.

Package simulateur.util

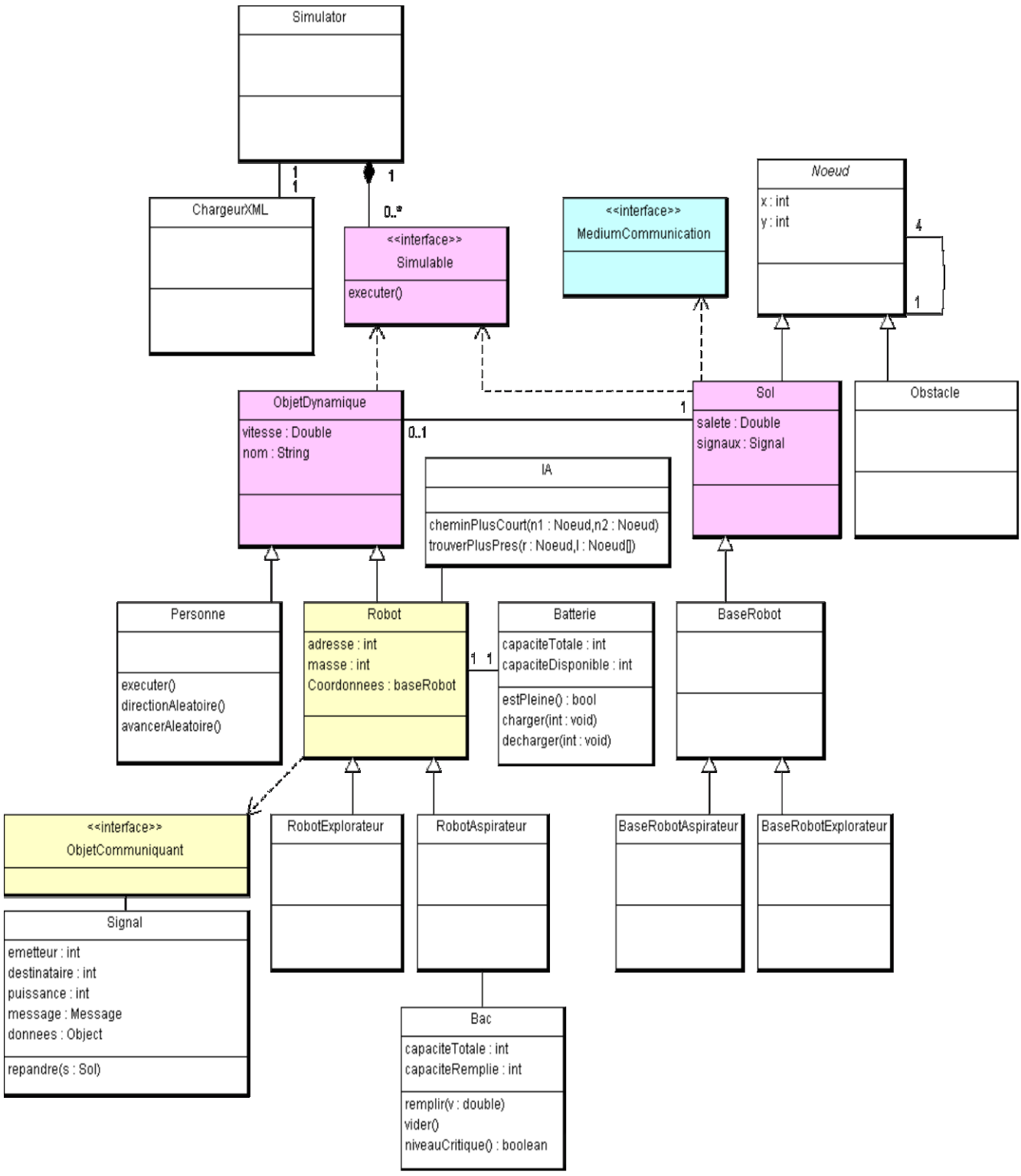
- classe Coordonnees.java

Chaque noeud a besoin de coordonnees qui permettent de le repérer dans la pièce.

- classe Loader.java

Cette classe permet de charger des images pour le dessin des objets à l'écran.

Diagramme de classes



Principales difficultés

1. Algorithme du chemin le plus court

Nous prenons une liste initiale de Nœud, où nous insérons dedans le nœud de départ. Ainsi, on boucle tant qu'il reste des éléments dans la file; puis pour chaque nœud, on insère tout ses voisins dans la file si ceux-ci n'ont pas déjà été traités auparavant. Ainsi de suite, par effet de profondeur, on arrivera jusqu'au nœud d'arrivée, et nous pouvons ainsi connaitre le chemin le plus court à prendre pour arriver jusque celui-ci.

Notre code peut donc être présenté sous cette forme:

-Initialisation d'une file

-Ajout du nœud de départ dans la file

-Tant que nœud présent dans la file

Si nœud = nœud d'arrivée alors

On retourne les directions prises jusqu'arriver en ce nœud

Sinon

On ajoute dans la file les voisins des nœuds s'ils n'ont pas déjà été traités auparavant

2. Algorithme de parcours de la pièce

-On fournit la direction du regard

-Selon direction du regard:

Si pas d'obstacle alors:

Le robot avance

On range les nœuds voisins au fur est à mesure dans un tableau

Si obstacle alors:

Le robot contourne le mur

Il se déplace en forme de carré

-Si tous les nœuds voisins connus, on cherche le nœud le plus proche dans le tableau des nœuds restant à traiter

-Si aucun nœud restant: on applique l'algorithme du chemin le plus court pour revenir à la base.

Principe de simulation

Le simulateur (un ActionListener) connait la liste des éléments simulables. Ces derniers doivent implémenter les interfaces Simulable et Printable.

L'interface ElementSimulable oblige l'implémentation des méthode preExecuter(), executer() et postExecuter(), indispensables à la simulation. L'interface printable permet d'afficher les informations de simulation que l'on peut observer en bas à droite de l'interface graphique.

Une boucle principale exécute donc le code des 3 niveaux d'exécution de chaque élément simulable. En effet, notre simulation se passe sur 3 niveaux car certaines actions doivent être simulées avant d'autres. Pour la communication entre robots par exemple, la méthode preExecuter permet d'effacer tous les signaux déjà existants, ensuite la méthode « Executer » permet d'ajouter les bons signaux, en fin la méthode postExecuter permet de vérifier si les signaux ont bien été ajoutés.