TME 5 : Programmation robuste

Objectifs pédagogiques : lecture de code, contrats, Exceptions, tests unitaires, JUnit

L'objectif de ce TME est d'apprendre à rendre robuste des développements en faisant appel aux tests unitaires et à la programmation par contrat. Dans le répertoire de votre TME5, créez un projet « tme5 ». Chargez-y l'archive *agent.jar* fournie sur le site de l'UE.

Vous trouverez dans cette archive un ensemble de classes permettant de représenter un agent qui se déplace dans un labyrinthe. Le déplacement de l'agent est commandé par un contrôleur à base de règles qui indiquent à l'agent comment se déplacer dans le labyrinthe à partir de ce qu'il perçoit autour de lui. Les règles associent des valeurs de capteurs (qui renseignent sur le contenu des cases environnantes) à des actions élémentaires de déplacement (gauche, droite, haut et bas).

1 Lecture de code

L'archive fournie contient deux programmes :

- dans la classe *agent.labyrinthe.interf.LabyBuilder*, en appelant la méthode *main()*, vous lancez une application qui permet de créer un labyrinthe de forme quelconque et de le sauver/charger dans un fichier
- dans la classe *agent.SimuMain*, une application simple crée un agent piloté par des règles aléatoires. Il simule ensuite son évolution dans le labyrinthe (classe Simulation) et lui attribue un score en fonction du nombre de cases qu'il a parcourues. On lui passe en paramètres le nom du fichier où se trouve le labyrinthe et le nombre de pas de simulation permettant d'évaluer un agent. Un fichier exemple *goal.mze* est fourni dans l'archive.

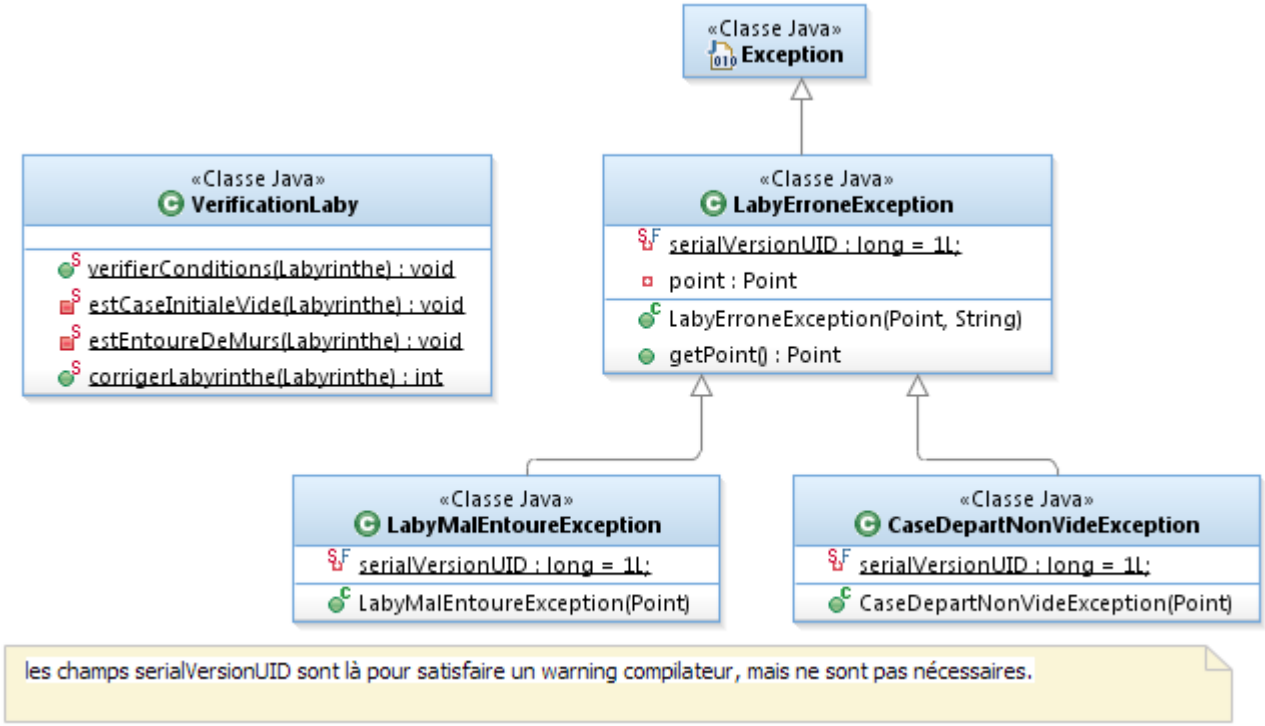
Commencez par lancer l'application *LabyBuilder et* créez un labyrinthe. Faites en sorte que le labyrinthe soit entouré de murs, pour éviter que l'agent puisse se mettre au bord et que ses capteurs pointent en dehors du labyrinthe. Sauvez votre labyrinthe dans un fichier. Examinez le code de sérialisation (classe agent.laby.ChargeurLabyrinthe).

Examinez ensuite le fonctionnement de l'application Simu*Main*. Avant de passer à la suite, explorez le code, générez sa javadoc, et familiarisez-vous avec l'application en lisant l'annexe de ce TME « Guide d'exploration du module Agent ».

2 Exceptions : Labyrinthes contractuels

Pour que l'application fonctionne, il est nécessaire que les labyrinthes vérifient deux conditions :

- d'une part, ils doivent être entourés d'une enceinte de murs afin que l'agent ne puisse pas atteindre une case « au bord » du labyrinthe. Si ce n'est pas le cas, ses capteurs environnants vont chercher la valeur de cases dont l'index est -1 ou supérieur à la taille limite.
- d'autre part, il faut que la case dans laquelle apparaît l'agent au lancement de l'application soit bien vide.



Cependant, les labyrinthes sont chargés depuis un fichier, donc il n'y a aucun moyen de garantir a priori qu'ils vérifient bien ces deux conditions avant de faire une simulation. Vous allez donc effectuer les vérifications nécessaires après avoir chargé un labyrinthe et lever une exception si le labyrinthe chargé ne vérifie pas le contrat qui lui est imposé. Cette gestion à base d'exceptions est essentiellement introduite pour vous faire manipuler les exceptions, on pourrait gérer ce problème de diverses autres façons.

Les contrôles seront introduits dans une nouvelle classe *VerificationLaby*, dans son opération «*public static void verifierConditions (Labyrinthe l) throws LabyErroneException* ».

Ecrivez d'abord une méthode qui vérifie qu'un labyrinthe est entouré de murs : `private static void estEntoureDeMurs(Labyrinthe l) throws LabyMalEntoureException`. Si cette méthode constate que ce n'est pas le cas, elle lève une occurrence de l'exception *LabyMalEntoureException* qui hérite de *LabyErroneException.* La classe *LabyErroneException* portera un *Point* donnant les coordonnées de la case présentant l'erreur. Pour une *LabyMalEntoureException* ce devrait être une case au bord du labyrinthe qui ne contient pas de mur, pour *CaseDepartNonVideException* ce sera la position de la case de départ.

Ecrivez une autre méthode « *private* » qui vérifie si la case de position initiale du labyrinthe est bien vide. De même, si ce n'est pas le cas, levez une exception *CaseDepartNonVideException* qui hérite de *LabyErroneException.*

Enfin, écrivez la méthode « *verifierConditions* » dont la signature est donnée ci-dessus.

Ajoutez à présent à la classe *VerificationLaby* une méthode «*public static int corrigerConditions (Labyrinthe l)* » qui corrige les éventuelles erreurs détectées. Cette méthode doit invoquer *verifierConditions()* et traiter les exceptions levées une par une. Elle rendra le nombre de corrections effectuées (donc 0 si pas de problèmes détectés). NB: comme *verifierConditions()* ne peut lever qu'une exception à la fois, *corrigerConditions()* doit invoquer *verifierConditions()* jusqu'à ce qu'il n'y ait plus d'exceptions levées.

Les codes de correction sont assez simples : si le labyrinthe est mal entouré, vous ajouterez un mur dans la case pointée par l'exception. De même, vous écrirez une méthode de réparation qui met le contenu de la case de départ à vide si nécessaire.

Avec l'application *LabyBuilder,* créez un labyrinthe qui ne vérifie aucune des deux conditions exprimées ci-dessus et vérifier que la réparation fonctionne correctement.

3 Installation de JUnit

Ajoutez *JUnit* dans votre projet. Pour cela, sélectionner votre projet dans la liste de gauche, activez le menu avec un clic droit, choisissez « *Build Path* » dans ce menu, puis choisissez l'action « *Add Library* », puis JUnit (version 3 plutôt que 4).

4 Tests unitaires sur les contrôleurs d'agents

Pour créer une classe de test, désignez la classe que vous voulez tester puis sélectionnez « nouveau » puis « *Junit Test Case* ». Dans la fenêtre qui apparaît, ajoutez « *.test* » au nom du package (nous rangeons les classes de test dans des packages séparés). Créez les classes de test dont vous avez besoin au fur et à mesure pour répondre aux questions ci-dessous.

Vous utiliserez *assertTrue()* et éventuellement *assertFalse()* pour vérifier que vos méthodes passent correctement les tests.

Le test que vous allez réaliser est le suivant : charger le labyrinthe *goal.mze* fourni sur le site de l'UE, placer un agent dans la case de coordonnées (1,1), créer un contrôleur qui contient une règle unique qui dit d'aller vers la droite quelle que soit la perception. Lorsque la simulation se termine, vous vérifierez que le score de l'agent correspond bien au nombre de cases qu'il peut parcourir en se déplaçant toujours vers la droite.

Le package *agent.control*, contient déjà dans la *Factory* une opération : « *IControleur creerControleurDroitier ()*». Pour cela, il crée une règle qui indique d'aller à droite quelles que soient les conditions (examinez les classes *Regle, ContenuCase* et *Direction*). Il crée ensuite un contrôleur et lui ajoute cette unique règle (voir la classe *Controleur*). Le constructeur de la règle prend en paramètre une chaîne de caractères. Les conditions correspondent au contenu des huit cases qui entourent l'agent, la condition 0 correspond à la case au-dessus de l'agent et on tourne dans le sens horaire (voir la classe *Agent*). 0 = Nord, 1 = NE, 2 = E, 3 = SE, 4 = S, 5 = SO, 6 = O, 7 = NO. Vous pouvez donc imaginer d'autres contrôleurs plus performants assez facilement. Lisez aussi l'annexe descriptive fournie à la fin de cet énoncé.

Effectuez donc la suite d'opérations suivante :

- avec l'interface de création de tests JUnit, créez une classe de test pour *Agent* que vous appellerez *AgentTest*. Pour cela, sélectionnez la classe *Agent* dans la liste de gauche, faites « *New → Junit Test Case* ». Indiquez que vous voulez mettre cette classe dans le package *test.agent*, cochez la génération de la méthode *setUp()*, faites « *Next* » et indiquez (en cochant) que vous voulez tester le constructeur et la méthode *mesurePerf()*,
- ajoutez à la classe *AgentTest* un attribut de type *Agent*, que vous appellerez *simuTest*,
- dans la méthode *setUp()* de la classe *AgentTest*, effectuez les opérations suivantes :
  - chargez le labyrinthe *goal.mze*
  - initialisez l'attribut *simuTest* avec une simulation qui contient ce labyrinthe et qui dure 20 pas de temps

- dans la méthode *testAgent()*, vous vous contenterez d'afficher le score associé à l'attribut *simuTest*;

- Pour pouvoir écrire la méthode *testMesurePerf()*, lancez une simulation de déplacement d'un agent dirigé par ce contrôleur « droitier » et vérifiez que le score renvoyé par la mesure de performance (cf. *Simulation*) est bien égal au nombre de cases libres à partir de la case de coordonnées (1,1) et en allant à droite.

- On pourra créer d'autres labyrinthes permettant de s'assurer du fonctionnement du score (mettez moins de points, un mur sur le chemin...).

Pour déclencher les tests, lancez l'opération « *Run as Junit Test* » sur la classe *AgentTest.* Vous exécuterez ces tests et corrigerez votre code jusqu'à ce que la barre de résultat des tests soit de couleur verte.

Ecrivez ensuite dans la *Factory* un contrôleur « *IControleur createSmartController()* » qui fait faire à l'agent un score élevé dans votre labyrinthe, et créez un nouveau test pour vérifier son score.

5. Remise du TME

Question : copiez-collez le code de votre méthode de contrôle « smart » du labyrinthe ainsi que le code de la méthode qui déclenche les réparations en cas de problème. Copiez-collez le code de votre classe de test *AgentTest* et ajoutez une trace d'exécution.

Annexe : Guide d'exploration du module « agent.jar »

Le module « agent.jar » est constitué de 3 packages : *agent, agent.laby* et *agent.control*

Diagramme du package *agent*

Le package *agent* contient 4 classes, il dépend des packages *agent.laby* et *agent.control*. Attention, l'éxécution d'une Simulation (à l'aide de *mesurePerf(int nbPas)*) modifie le labyrinthe. On peut cepen

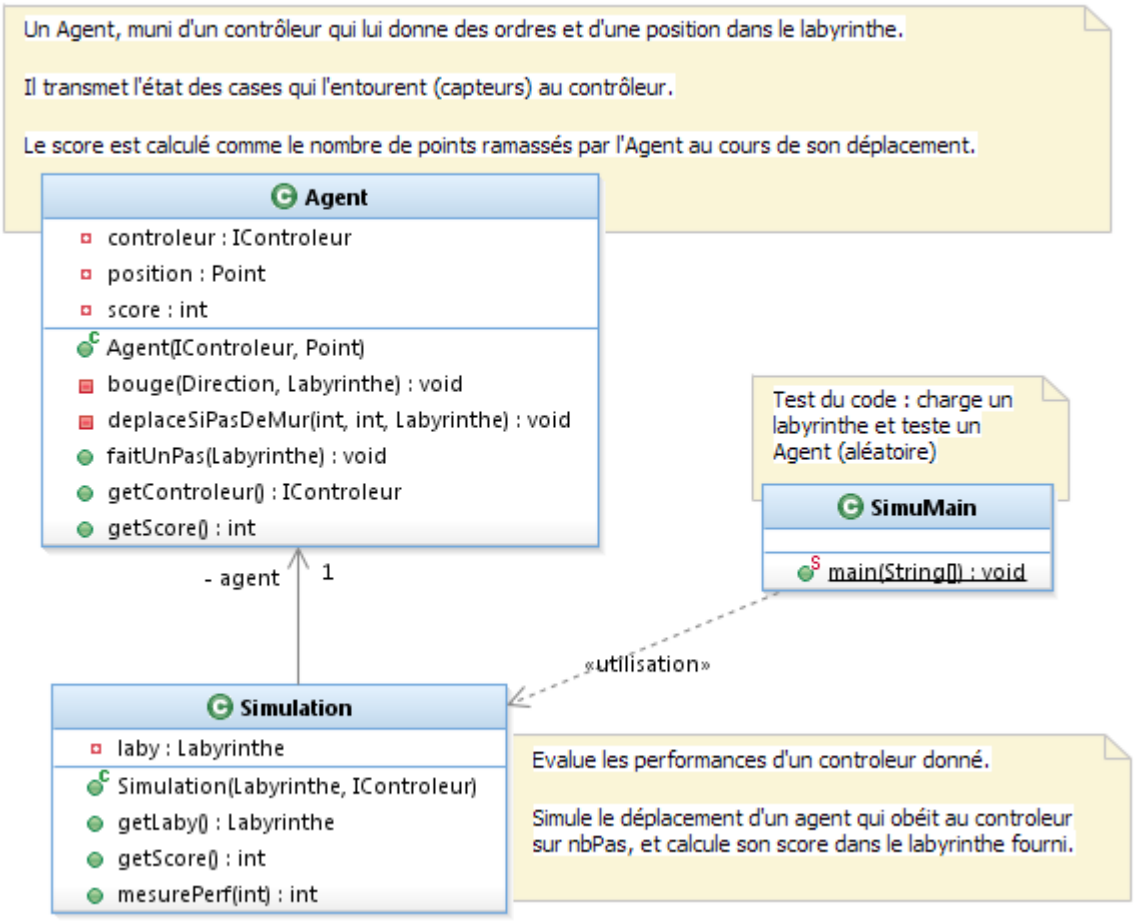


Diagramme du package agent.laby

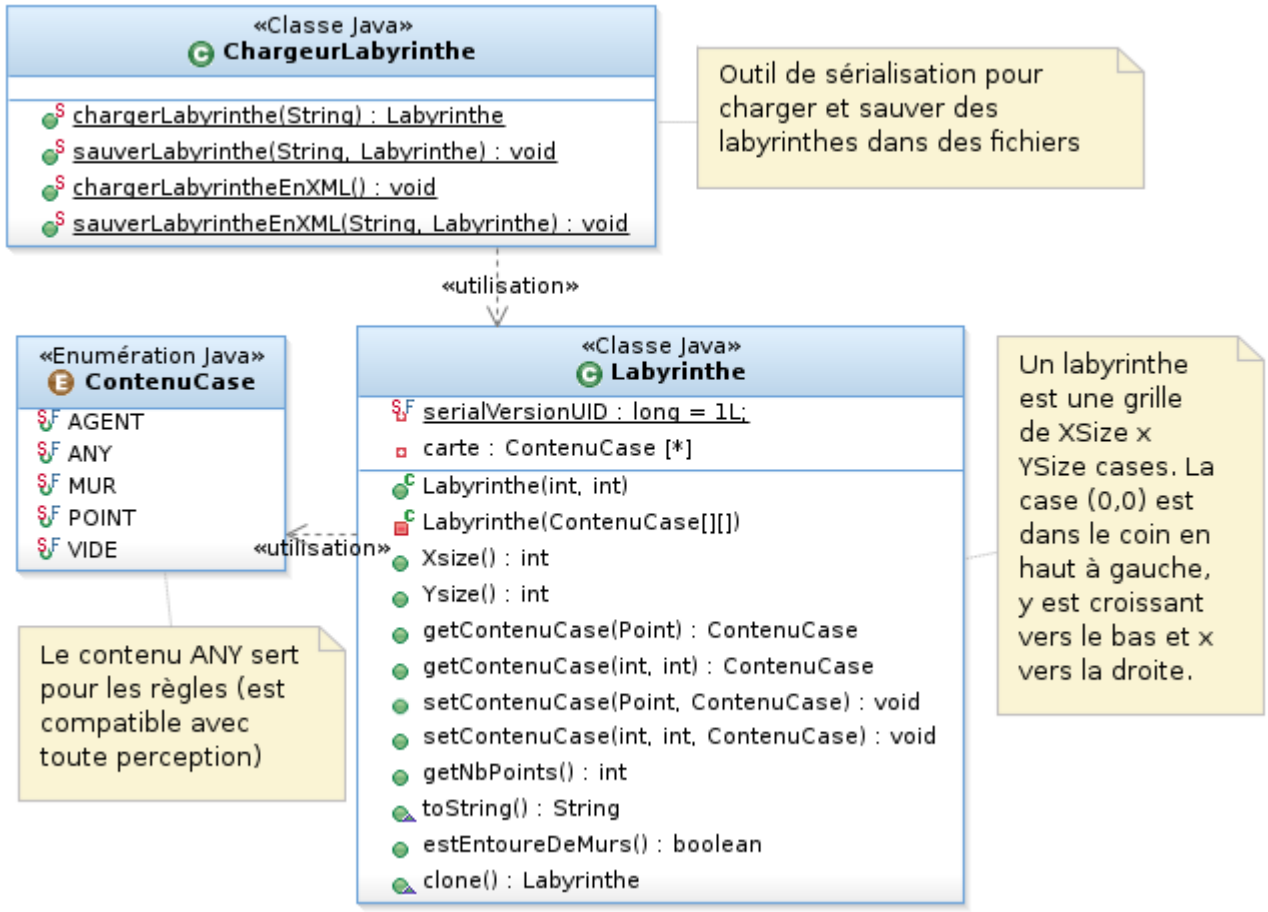
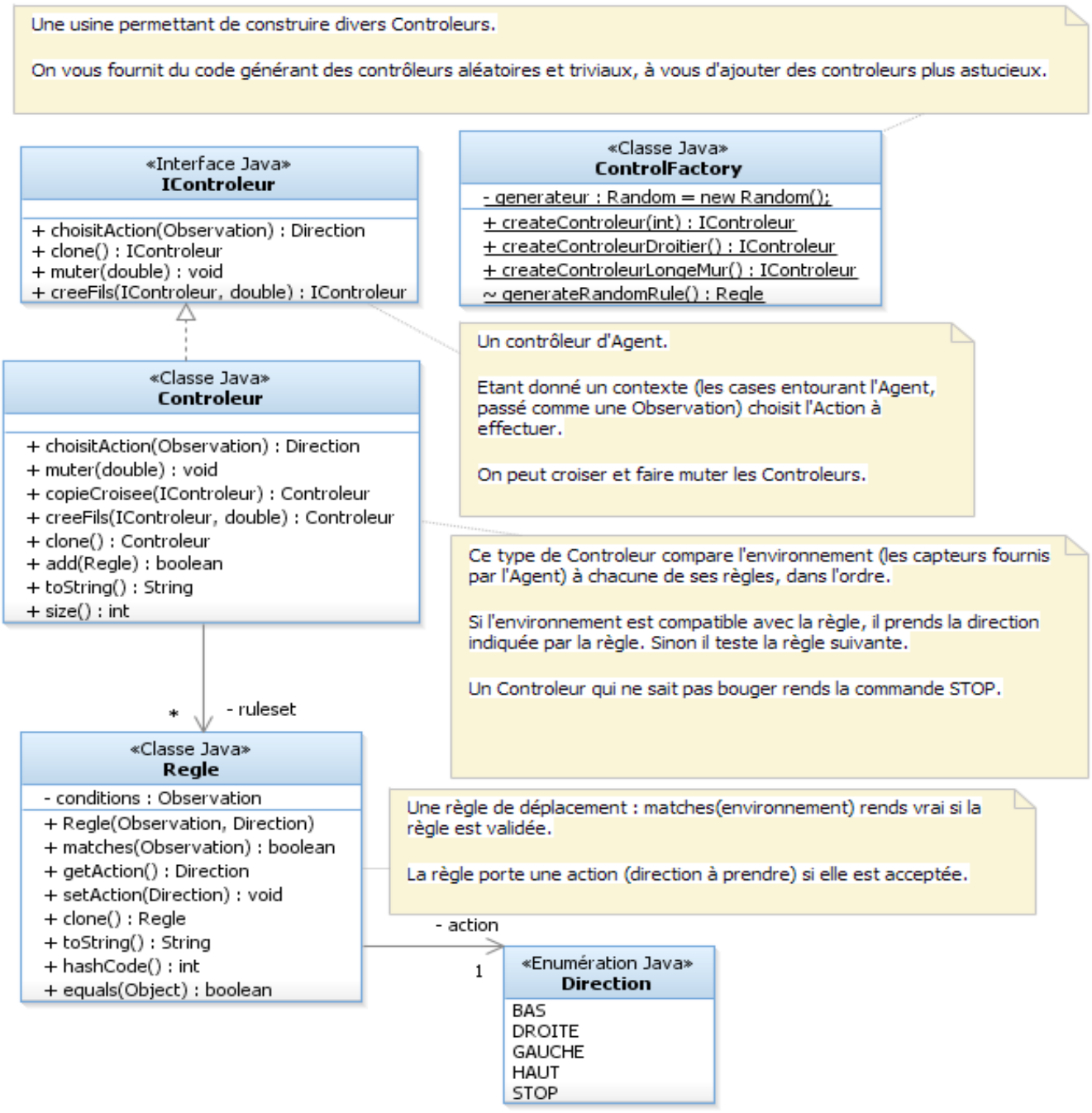
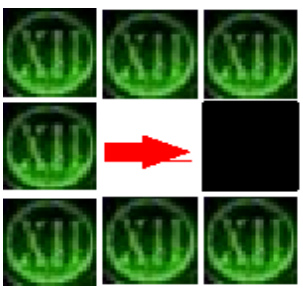
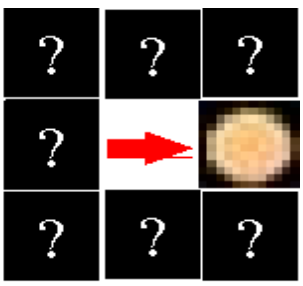


Diagramme du package agent.control

Le comportement d'un agent dans un labyrinthe est dirigé par un contrôleur. Ce contrôleur contient un ensemble de règles de déplacement. A chaque pas de temps, il compare la situation perçue par l'agent à la partie condition de chacune des règles. La première règle dont la partie condition est compatible avec la situation courante (méthode *matches()*) est déclenchée. Dans la représentation graphique ci-dessous, la case centrale de chaque règle indique la direction de déplacement par une flèche orientée dans une des 4 directions possibles et les 8 cases environnantes spécifient le contenu de la partie condition avec des symboles correspondant aux perceptions de l'agent dans les 8 directions.


Une usine permettant de construire divers Controleurs.
On vous fournit du code générant des contrôleurs aléatoires et triviaux, à vous d'ajouter des controleurs plus astucieux.
«Interface Java»
IControleur
+ choisitAction(Observation) : Direction
+ clone() : IControleur
+ muter(double) : void
+ creeFils(IControleur, double) : IControleur
«Classe Java»
ControlFactory
- generateur : Random = new Random();
+ createControleur(int) : IControleur
+ createControleurDroitier() : IControleur
+ createControleurLongeMur() : IControleur
~ generateRandomRule() : Regle
Un contrôleur d'Agent.
Etant donné un contexte (les cases entourant l'Agent, passé comme une Observation) choisit l'Action à effectuer.
On peut croiser et faire muter les Controleurs.
«Classe Java»
Controleur
+ choisitAction(Observation) : Direction
+ muter(double) : void
+ copieCroisee(IControleur) : Controleur
+ creeFils(IControleur, double) : Controleur
+ clone() : Controleur
+ add(Regle) : boolean
+ toString() : String
+ size() : int
Ce type de Controleur compare l'environnement (les capteurs fournis par l'Agent) à chacune de ses règles, dans l'ordre.
Si l'environnement est compatible avec la règle, il prends la direction indiquée par la règle. Sinon il teste la règle suivante.
Un Controleur qui ne sait pas bouger rends la commande STOP.
*
- ruleset
«Classe Java»
Regle
- conditions : Observation
+ Regle(Observation, Direction)
+ matches(Observation) : boolean
+ getAction() : Direction
+ setAction(Direction) : void
+ clone() : Regle
+ toString() : String
+ hashCode() : int
+ equals(Object) : boolean
Une règle de déplacement : matches(environnement) rends vrai si la règle est validée.
La règle porte une action (direction à prendre) si elle est acceptée.
- action
1
«Enumération Java»
Direction
BAS
DROITE
GAUCHE
HAUT
STOP





LI314 : Programmation par Objets

Exemples de représentations graphiques de règles : la règle de gauche (codée « ## ##### ») indique de se déplacer à droite quand c'est la seule direction possible, celle de droite (codée « ??. ????? ») indique d'aller vers un « point » à droite quel que soit le contenu des autres cases environnantes.