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 dans les huit directions environnantes) à des actions élémentaires de déplacement (gauche, droite, haut et bas). Deux exemples de règles sont illustrés sur la figure 5.

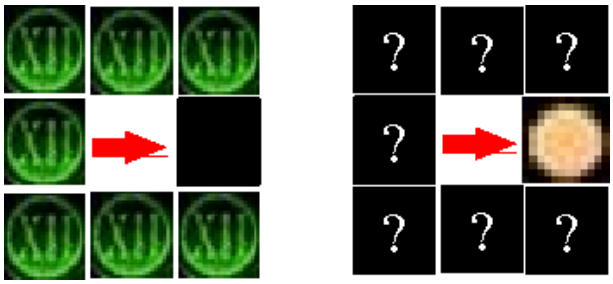

Figure 1: Représentation graphique des règles de déplacement

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 rouge à droite quel que soit le contenu des autres cases environnantes.

5.1 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 et 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 `SimuMain`**. 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 ».

5.2 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.

⇒ Créez une nouvelle classe `VerificationLaby`.

⇒ **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`.

⇒ Créez les classes représentant ces exceptions. `LabyErroneException` est une Exception qui porte en attribut un `Point` donnant les coordonnées de la case présentant l'erreur. Ajoutez deux exceptions spécialisées: pour une `LabyMalEntoureException` le point désignera une case au bord du labyrinthe qui ne contient pas de mur; pour une `CaseDepartNonVideException` le point désignera la case initiale du labyrinthe.

⇒ Ecrivez une autre méthode « private » qui vérifie si la case de position initiale du labyrinthe est bien vide. `private static void estCaseInitialeVide(Labyrinthe l) throws CaseDepartNonVideException`

⇒ Enfin, **écrivez la méthode `verifierConditions`** dont la signature est `public static void verifierConditions (Labyrinthe l) throws LabyErroneException` et qui teste les deux conditions.

⇒ **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, ajoutez un mur dans la case pointée par l'exception. ⇒ écrivez 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 vérifiez que la réparation fonctionne correctement.

5.2.1 Installation de JUnit

⇒ **Ajoutez `JUnit` dans votre projet**. Pour cela, sélectionnez 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 Libraries...*». Dans le choix de la version de `JUnit` sélectionnez `JUnit 4`.

5.2.2 Tests unitaires sur les contrôleurs d'agents

Pour créer une classe de test, désignez la classe que vous voulez tester puis sélectionnez «*New*» 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.

Pour définir les conditions de réussite des tests, vous utiliserez les méthodes statiques de la classe `Assert` du package `org.junit`, comme `Assert.assertTrue`, `Assert.fail` ou `Assert.assertEquals`.

Le test que vous allez réaliser est le suivant :

⇒ chargez le labyrinthe `goal.mze` fourni sur le site de l'UE,

⇒ placez un agent dans la case de coordonnées (1,1),

⇒ créez un contrôleur qui contient une règle unique qui dit d'aller vers la droite quelle que soit la perception.

⇒ lorsque la simulation se termine, vérifiez 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 classe `ControlFactory` 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 séquence d'opérations suivante :

- avec l'interface de création de tests `JUnit` dans Eclipse, créez une classe de test pour `Simulation` que vous appellerez `SimulationTest`. Pour cela :
  - sélectionnez la classe `Simulation` dans la liste de gauche, faites un clic droit, puis «*New* → *JUnit Test Case*» ;
  - spécifiez la version de `JUnit` utilisée en sélectionnant «*New JUnit 4 test*» ;
  - indiquez que vous voulez mettre cette classe dans le package `test.simulation` ;
  - cochez la génération de la méthode `setUp()` et décochez la génération de la méthode `tearDown()`, inutile dans notre cas ;
  - faites «*Next*» et indiquez (en cochant) que vous voulez tester la méthode `mesurePerf()`, puis faites «*Finish*» ;
  - si vous n'avez pas ajouté la bibliothèque `JUnit 4` au projet, Eclipse proposera à ce moment de l'ajouter (si c'est le cas, acceptez) ;
- dans la méthode `setUp()` de la classe `SimulationTest`, chargez le labyrinthe `goal.mze` en prenant en compte les éventuelles exceptions ;
- dans la méthode `testMesurePerf()` de la classe `SimulationTest`, lancez une simulation de déplacement d'un agent dirigé par un contrôleur «droitier» et vérifiez que le score renvoyé par la mesure de performance est bien égal au nombre de cases libres à partir de la case de coordonnées (1,1) et en allant à droite.

Pour déclencher les tests, lancez l'opération «*Run as JUnit Test*» sur la classe `SimulationTest` (dans le menu du clic droit).

⇒ Exécutez ces tests et corrigez votre code jusqu'à ce que la barre de résultat des tests soit de couleur verte, ce qui signifie que tous les tests passent correctement.

⇒ Ecrivez ensuite dans la `Factory` un contrôleur `IControleur createSmartController()` qui fait faire à l'agent le tour du labyrinthe en suivant les murs et créez un nouveau test pour vérifier son score.

5.3 Remise du TME

Copiez-collez le code de votre méthode de chargement 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 `SimulationTest`

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'exécution d'une simulation (à l'aide de `mesurePerf(int nbPas)`) modifie le labyrinthe. On peut cependant cloner les labyrinthes si nécessaire avant de créer une simulation.

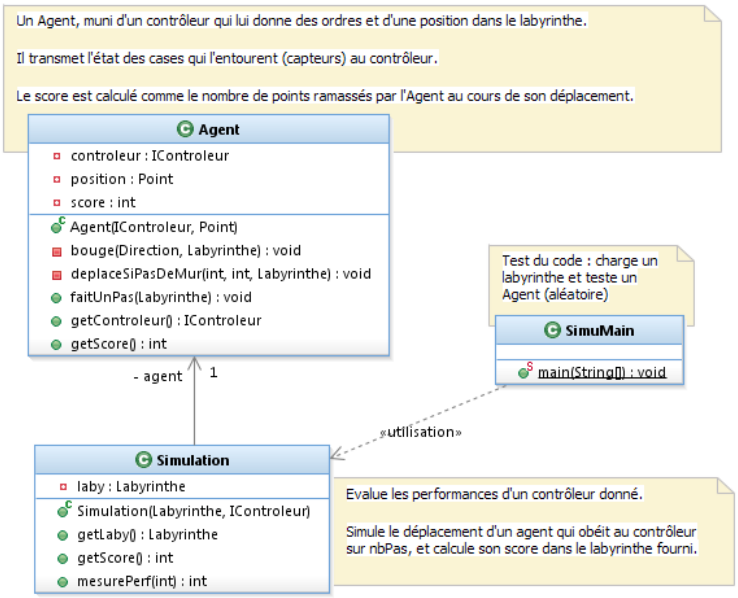

Figure 2: Diagramme du package agent

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.

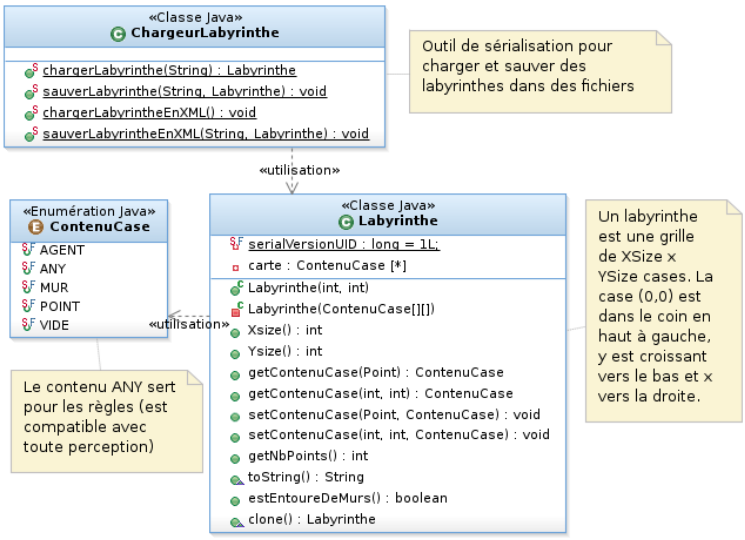

Figure 3: Diagramme du package `agent.laby`

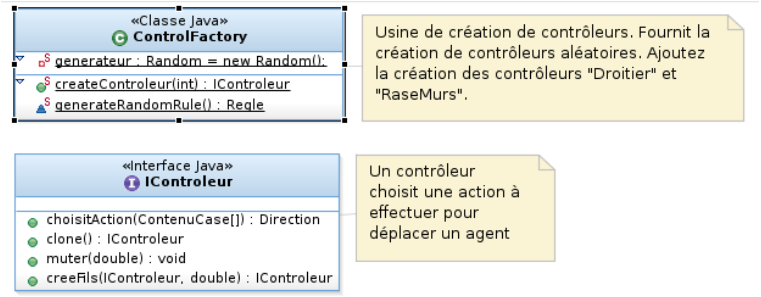

Figure 4: Diagramme du package `agent.control`(1)

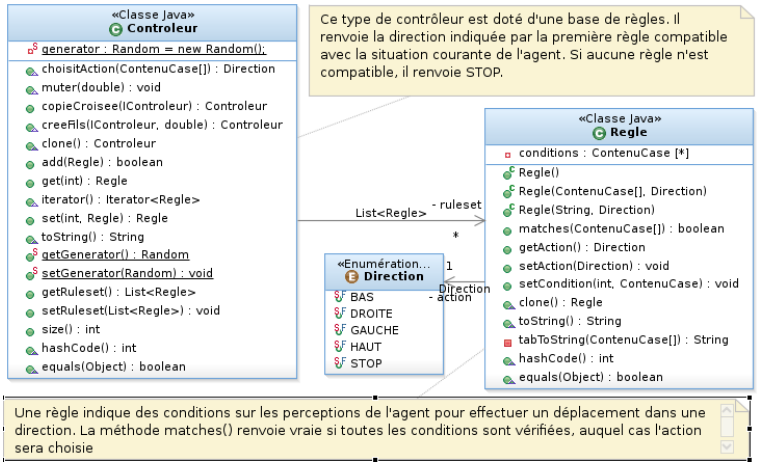

Figure 5: Diagramme du package `agent.control`(2)