TME 6 : Evolution d'agents

Objectifs pédagogiques : montée en abstraction par fusion avec un autre cadre d'application, pattern *adapter*

Au TME4, vous avez appliqué un algorithme évolutionniste à des expressions arithmétiques en recherchant une expression cible via une approximation de fonction à partir de points. Vous allez à présent réutiliser vos classes d'évolution génétique pour rechercher par évolution un contrôleur d'agent (vu au TME5) qui maximise le nombre de cases visitées par l'agent.

Pour atteindre cet objectif, vous allez adapter la couche logicielle permettant l'évolution génétique pour pouvoir l'appliquer indifféremment aux expressions arithmétiques ou aux contrôleurs d'agents. En particulier, vous ferez en sorte de ne pas toucher au code fourni au TME5. L'objectif applicatif est d'engendrer par évolution un contrôleur qui permette à l'agent de parcourir un maximum de cases du labyrinthe.

Dans votre répertoire de travail habituel, lancez eclipse et créez un nouveau projet que vous appellerez «tme6». Chargez l'archive résultant de votre TME5.

2. Fusion de code par montée en abstraction

Vous allez maintenant retravailler la structure du package *pobj.algogen* pour permettre la réalisation de ce nouveau scénario (parmi d'autres). Pour cela, il va falloir monter en abstraction. Comparez les solutions obtenues aux TME2 et TME4 (à travers les diagrammes schématisant la solution de l'énoncé).

Il y a clairement une redondance entre les solutions : on a copié-collé une grande partie du code. Il y a cependant des points satisfaisants, ainsi le package *pobj.arith* ne dépend pas du package *pobj.algogen*.

Votre objectif est donc d'identifier les points communs entre le TME2 et le TME4 afin de modifier la structure du package *algogen* de façon à pouvoir utiliser le même code de manipulation génétique (essentiellement l'évolution d'une population) pour manipuler un *double* comme au TME2, une *Expression* comme au TME4, ou un *agent* dans un labyrinthe comme au TME5. Un diagramme UML complet de la solution envisagée se trouve à la fin de cet énoncé.

2.1 Première étape : ajout d'interfaces et abstractions.

Il faut généraliser votre solution pour permettre sa réutilisation. Pour cela, vous allez découper la fonctionnalité pour avoir une partie générale (tout ce qu'on a identifié comme points communs entre la version du TME2 et celle du TME4) et des parties spécialisées pour traiter les *double*, les *Expression* puis enfin les agents du labyrinthe.

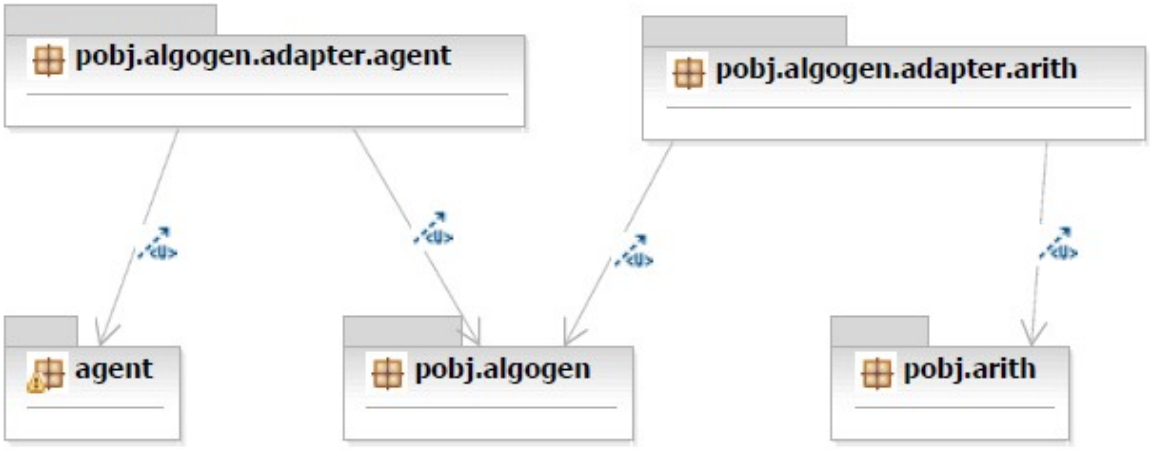


Illustration 1: Découpage des fonctionnalités

On veut centraliser dans le package *algogen* uniquement les parties génériques et laisser le moins de code possible à écrire dans les package *adapter*. D'autre part, comme le montrent les dépendances indiquées sur le diagramme (illustration 1), le mécanisme doit permettre de poser un algorithme génétique sur un package existant sans le modifier : les packages *agent* ou *arith* ne dépendent pas du package *algogen*.

Dans le package *algogen*, créez une interface *Individu* et faites-lui porter toutes les opérations nécessaires au fonctionnement de la population.

Pour cela, vous pourrez vous aider du menu *Refactor* d'eclipse. En partant de la classe *Individu,* essayez la séquence : **Refactor → Extract Interface**. Vous pouvez aussi y aller progressivement en vous aidant des autres entrées.

Refactor: Use Super Type where possible vous permet de propager l'introduction de l'interface dans les classes qui s'en servent.

Refactor: pull up permet de remonter la déclaration et/ou l'implémentation d'une opération sur les parents d'une classe (interface implémentée ou classe parente).

Suite au *refactoring* (littéralement ré-ingéniérie) du code, on cherche à obtenir le diagramme ci-après (Illustration 2). Vous devez donc :

- Créer une interface *Environnement*, de façon à ce que la classe *ValeurCible* implémente cette interface. Remontez toutes les opérations utiles dans cette interface pour que la population puisse réaliser des évaluations avec cet environnement (cf. illustration UML).

- Déclarer dans la classe *Population* la nature « conteneur d'individu » de la population, en lui faisant implémenter *Iterable<Individu>*, si vous savez le faire. Il suffit de rendre par délégation un itérateur sur la List<Individu> sous-jacente. Sinon, vous définirez au moins la taille et un accesseur type *Individu get(int index)*.

- Créer une classe *AbstractIndividu* représentant un individu abstrait, qui porte seulement la responsabilité de la fitness. L'idée est qu'un *IndividuDouble* ou un *IndividuExpression* pourraient ainsi en dériver et factoriser leur code. Implémentez l'interface *Comparable* dans cette classe, en utilisant la fitness comme critère de comparaison entre les individus.

- Créer un package *pobj.algogen.adapter.arith* et y déplacez la méthode *main()* et vos classes *IndividuExpression* et *ValeurCible*. Placez-y également la *Factory* statique, qui portera des signatures abstraites: *Environnement createEnvironnement()* avec un paramètre *EnvEval* et une valeur à

préciser, etc. La méthode *main()* ne devra dépendre **que** de cette *Factory* et des classes et interfaces *Population, Individu, Environnement* du package *algogen*.

3. Evolution des Agents

Créez un nouveau package *pobj.algogen.adapter.agent* qui va servir à « brancher ensemble » les déplacements de l'agent et l'évolution génétique. En vous inspirant du package *adapter.arith* créé plus haut, adaptez la manipulation des agents au problème.

Dans le package *pobj.algogen.adapter,agent,* vous allez créer deux classes d'adaptation : une classe *ControleurIndividuAdapter*, qui permet de voir les contrôleurs des agents comme des individus de la population qu'on fait évoluer, et une classe Laby*EnvironnementAdapter*, qui permet de voir la simulation d'agent dans le labyrinthe comme un environnement d'évaluation pour l'évolution des contrôleurs. Ces classes implémentent respectivement les interfaces *Individu* et *Environnement*.

Développez aussi une version modifiée de la *Factory*, présentant des arguments de configuration adaptés aux agents (nombre de règles, nombre de pas de simulation, labyrinthe...).

Enfin, vous adapterez la méthode *main()* qui devrait simplement charger le labyrinthe puis se comporter en tout point comme la version du package *adapter.arith*.

3.1 Classe ControleurIndividuAdapter

Un *ControleurIndividuAdapter* est un *AbstractIndividu* (il dispose donc d'une fitness) et contient un contrôleur (sa valeur propre). Déduisez-en la ou les interface(s) implémentée(s) par la classe et ses attributs. Un certain nombre de méthodes en découlent directement, ajoutez-les.

Pour croiser deux contrôleurs, on utilisera l'opération fournie *creeFils()* qui crée un descendant du contrôleur. La mutation est déjà prévue également, mais il faudra en adapter la signature.

Par ailleurs, la classe *agent.control.ControlFactory* permet de créer des contrôleurs basés sur des règles.

3.2 Classe *MazeSimuEnvironnementAdapter*

Un *AgentEnvironnementAdapter* est vu comme un *Environnement* et contient un *Labyrinthe*. Pour évaluer le fitness d'un individu porteur d'un contrôleur, il lance une nouvelle simulation avec ce contrôleur et **une copie** du labyrinthe. Le score obtenu par l'agent à l'issue de cette simulation (configurée par *nbSteps*) représente sa fitness.

Déduisez-en la ou les interface(s) implémentée(s) par la classe et ses attributs. Un certain nombre de méthodes en découlent directement, ajoutez-les.

Pour évaluer un agent, sa fitness est égale à son score, c'est-à-dire au nombre de cases qu'il a parcourues. Ainsi, plus la fitness est élevée, donc plus l'agent a parcouru de cases, meilleur est l'agent.

3.3 Classe *PopulationFactory et main*

La classe *PopulationFactory* doit permettre de configurer une population et un problème. Une population est définie par sa taille, et le nombre de règles que chaque contrôleur (*Individu*) utilise. L'environnement est défini par un labyrinthe donné et le nombre de pas de simulation à réaliser avant de donner un score. On se donnera des opérations *static* qui cachent le type concret utilisé : les signatures de retour seront des *Population, Environnement, Individu*.

Ces modifications introduites, la méthode *main()* sera quasiment identique à celle des expressions arithmétiques. Elle prend en arguments le nom du fichier contenant le labyrinthe sur lequel on travaille, la taille de la population, le nombre de générations, le nombre de règles par contrôleur et le nombre de pas de simulation pour chaque évaluation. Elle instancie un *ChargeurLabyrinthe* pour

charger un labyrinthe, invoque la *Factory* pour créer une population, puis fait évoluer la population sur *n* générations et affiche le meilleur individu de la dernière génération.

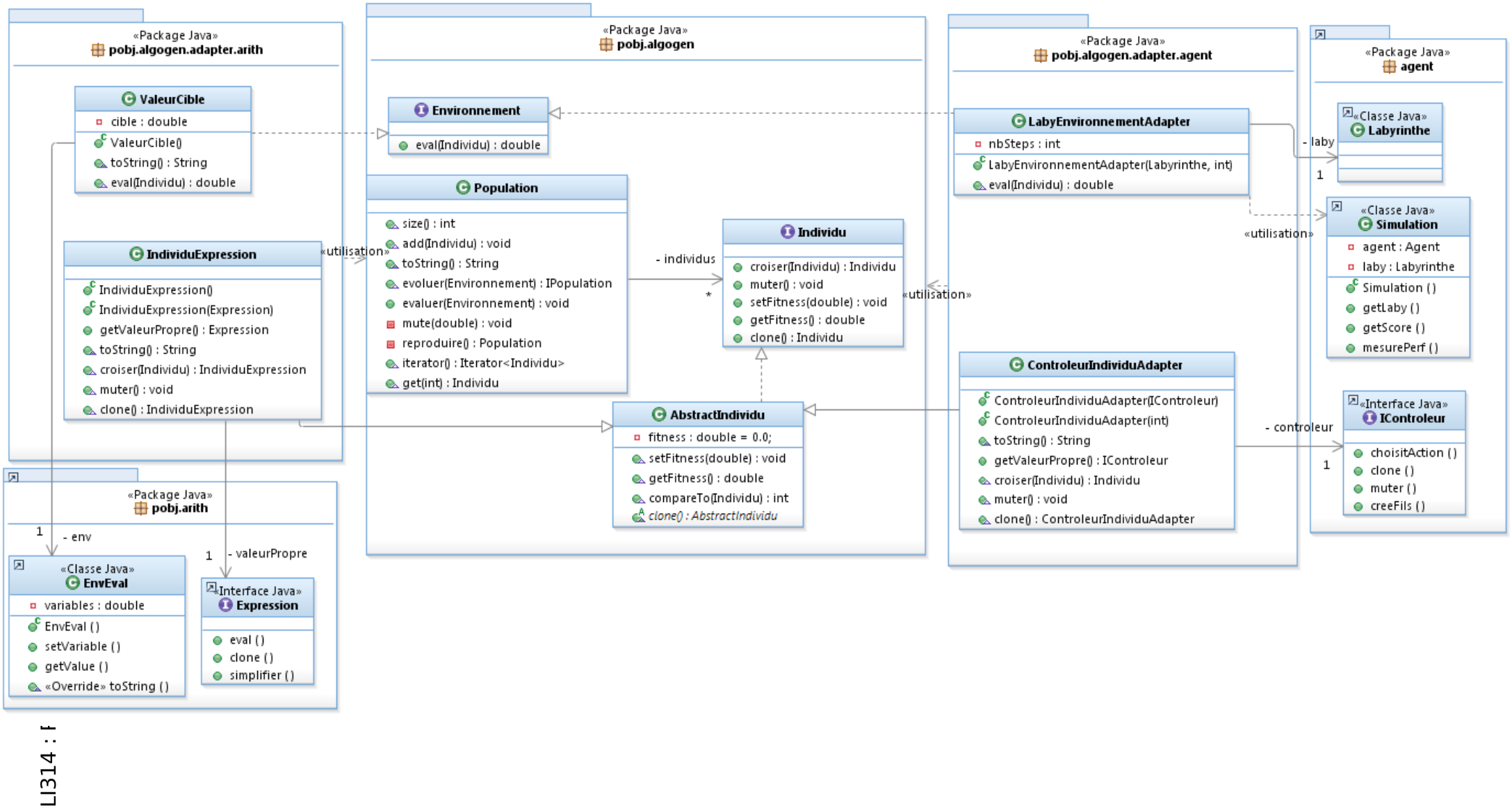
4. Tests unitaires

Vous allez passer un ensemble de tests unitaires sur le code que vous avez développé. Récupérez le fichier « tme6_tests.jar » sur le site de l'UE. Installez-le dans votre projet, ainsi que Junit 3.8 (cf. TME5). Déclenchez tous les tests unitaires fournis. Corrigez votre code jusqu'à ce que tous les tests passent correctement.

5. Remise du TME

Question : Copiez-collez le code de votre classe *ControleurIndividuAdapter.* Après avoir spécifié les valeurs que vous avez choisies pour tous les paramètres, indiquez dans votre mail le nombre de cases visitables que contient le labyrinthe que vous avez créé et le score obtenu par le meilleur individu à la fin de l'évolution. Copiez-collez une trace d'exécution.

LI314 : Programmation par Objets


«Package Java»
pobj.algogen.adapter.arith
ValeurCible
cible : double
ValeurCible()
toString() : String
eval(Individu) : double
IndividuExpression
IndividuExpression()
IndividuExpression(Expression)
getValeurPropre() : Expression
toString() : String
croiser(Individu) : IndividuExpression
muter() : void
clone() : IndividuExpression
«utilisation»
«Package Java»
pobj.algogen
Environnement
eval(Individu) : double
Population
size() : int
add(Individu) : void
toString() : String
evoluer(Environnement) : IPopulation
evaluer(Environnement) : void
mute(double) : void
reproduire() : Population
iterator() : Iterator<Individu>
get(int) : Individu
- individus
*
Individu
croiser(Individu) : Individu
muter() : void
setFitness(double) : void
getFitness() : double
clone() : Individu
AbstractIndividu
fitness : double = 0.0;
setFitness(double) : void
getFitness() : double
compareTo(Individu) : int
clone() : AbstractIndividu
«Package Java»
pobj.algogen.adapter.agent
LabyEnvironnementAdapter
nbSteps : int
LabyEnvironnementAdapter(Labyrinthe, int)
eval(Individu) : double
«utilisation»
ControleurIndividuAdapter
ControleurIndividuAdapter(IControleur)
ControleurIndividuAdapter(int)
toString() : String
getValeurPropre() : IControleur
croiser(Individu) : Individu
muter() : void
clone() : ControleurIndividuAdapter
- controleur
1
«Package Java»
agent
- laby
1
«Classe Java»
Labyrinthe
«Classe Java»
Simulation
agent : Agent
laby : Labyrinthe
Simulation ()
getLaby ()
getScore ()
mesurePerf ()
«Interface Java»
IControleur
choisitAction ()
clone ()
muter ()
creeFils ()
«Package Java»
pobj.arith
1
- env
1
- valeurPropre
«Classe Java»
EnvEval
variables : double
EnvEval ()
setVariable ()
getValue ()
«Override» toString ()
«Interface Java»
Expression
eval ()
clone ()
simplifier ()

LI314 : I