TME 9 : Evolution d'agents

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

Au TME 4, vous avez appliqué un algorithme évolutionniste à des expressions arithmétiques en recherchant une expression cible qui passe en un point donné. Vous allez à présent réutiliser vos classes d'évolution génétique pour rechercher par évolution un contrôleur d'agent qui maximise le nombre de cases visitées par l'agent.

Pour atteindre cet objectif, vous allez réutiliser la couche logicielle abstraite du TME4 pour pouvoir appliquer l'évolution génétique aux contrôleurs d'agents.

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 «tme9». Dans ce projet importez :

- le contenu de votre TME 7 (labyrinthes),
- le contenu de votre TME 4.

9.1 Introduction de génériques

A l'issue de la première partie du TME 4, on a obtenu les classes suivantes:

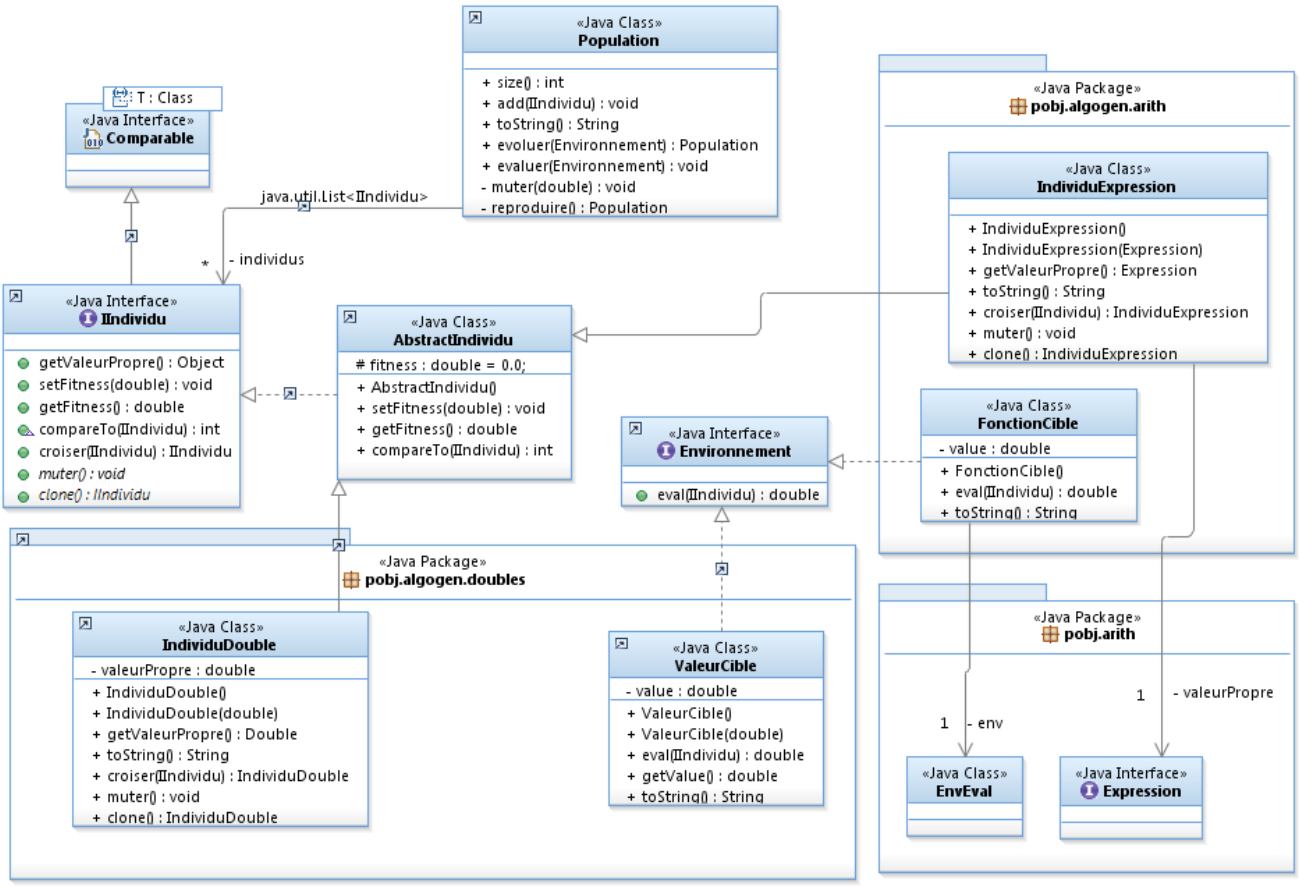

Figure 1: Situation fin TME 4

Cette solution abstraite pose néanmoins un souci de typage de la valeur propre : on a utilisé le type `Object`, ce qui nécessite des transtypages (cast) dans l'environnement et au moment de croiser les individus.

⇒ Transformez l'interface `IIndividu` pour qu'elle prenne un paramètre générique `T`. La valeur de retour de getValeurPropre() sera `T`. Le typage de tous les `IIndividu` mentionnés dans les signatures sera aussi un `IIndividu<T>`.

⇒ Propagez la modification dans les sous-classes de `IIndividu`. On a donc un `AbstractIndividu<T>`. La classe `IndividuExpression` étend `AbstractIndividu<Expression>` alors que la classe `IndividuDouble` étend `AbstractIndividu<Double>`.

⇒ `les Environnement<T>` évaluent des `IIndividu<T>`. La classe `ValeurCible` implémente `Environnement<Double>` et `FonctionCible` implémente `Environnement<Expression>`.

⇒ Débarrassez-vous des transtypages et tests `instanceof` existants dans ces classes.

⇒ Propagez la modification : la `Population<T>` porte des `IIndividu<T>`

⇒ Avec cette modification, remontez le stockage de la valeur propre d'un individu concret sur la classe `AbstractIndividu<T>`. Ajoutez-y un constructeur prenant la valeur propre typée `T` en argument et une opération `void setValeurPropre(T vp)` de visibilité protégée.

⇒ Assurez-vous que tout votre code compile, sans warnings de typage générique mal renseigné.

9.2 Adapter le package `agent` pour `algogen.generic`

Créez un nouveau package `pobj.algogen.agent` qui va servir à « brancher ensemble » les déplacements de l'agent et l'évolution génétique. En vous inspirant du package `pobj.algogen.arith` créé au TME 4, adaptez la manipulation des agents au problème.

⇒ Créez un package `pobj.algogen.agent`.

⇒ Créez une classe d'adaptation `IndividuControleur` qui étend `AbstractIndividu<IControleur>`.

Pour croiser deux contrôleurs, utilisez l'opération fournie `creeFils()` qui crée un descendant du contrôleur. La mutation est déjà prévue également, mais il faut adapter sa signature pour passer les objets pertinents en paramètre. Par ailleurs, la classe `agent.control.ControlFactory` permet de créer des contrôleurs.

⇒ Créez une classe `SimulationCible` qui implémente `Environnement<IControleur>`.

Elle portera en attributs un `Labyrinthe` et un nombre de pas (int). Pour évaluer un individu, elle crée une `Simulation` positionnée sur une copie du labyrinthe et sur le contrôleur de l'individu. Le nombre de points obtenu (calculé par la méthode `mesurePerfs()` de la simulation) correspond à la fitness.

⇒ 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...).

Cette classe 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. Ajoutez des opérations `static` qui cachent le type concret utilisé : les signatures de retour seront des `Population`, `Environnement`, `IIndividu` avec un type générique `<IControleur>`.

9.3 Visualisation du meilleur agent

⇒ Intégrez ces modifications pour écrire une méthode `main()`, basée sur celle des expressions arithmétiques du TME 4.

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.

⇒ En utilisant le code d'interface graphique développé au TME 7, faites le nécessaire pour afficher le comportement du meilleur agent trouvé par l'algorithme génétique.

9.4 Remise du TME

Copiez-collez le code de votre classe `IndividuControleur`. 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.