TME 4: Algorithmes génétiques sur les expressions arithmétiques

Objectifs pédagogiques : récupération de projets sous eclipse, fusion de projets (sensibilisation aux problèmes d'architecture que cela pose), application multi-package, héritage, utilisation de collections simples tirées de l'API, *equals()* et copie profonde, manipulation d'*ArrayList*, utilisation des interfaces

1. Récupération de projets

Lors des TME précédents, vous êtes partis « de rien ». Cette semaine, vous allez réutiliser et fusionner les projets des deux semaines précédentes.

L'objectif est d'utiliser l'algorithme génétique du TME 2 pour chercher une expression arithmétique (code développé au TME3) qui passe par un point donné. Par exemple, on cherche une fonction f(x1,...,xn) telle que f(x1=0.3,x2=0.8, ... xn=0.12)= 0.37.

Pour cela, vous allez construire et évaluer une population d'expressions arithmétiques, en mettant en place les mécanismes d'évolution permettant de rechercher une expression qui passe par le point ciblé.

Dans votre répertoire de travail habituel, créez un répertoire TME4 et lancez eclipse avec ce répertoire comme origine.

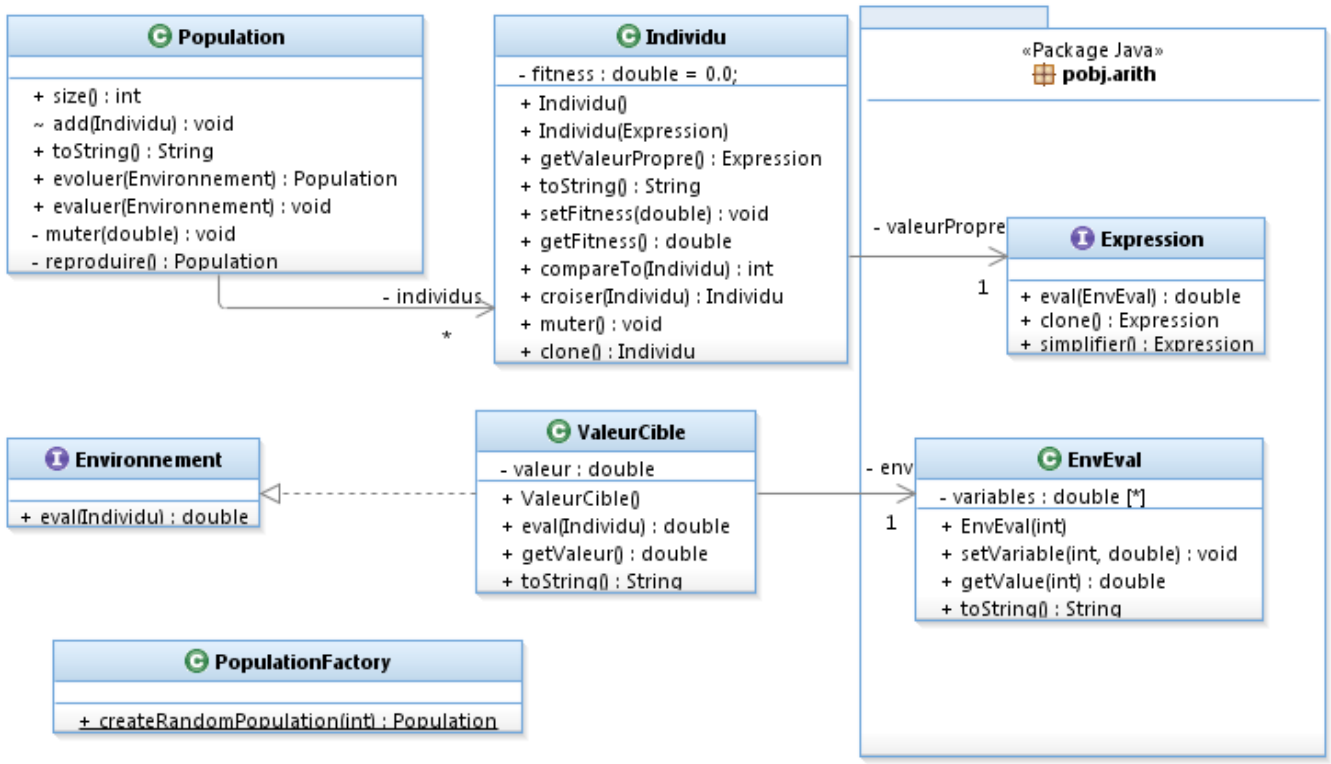
Sous eclipse, créez un nouveau projet que vous appellerez « tme4 » et importez successivement les deux archives correspondant à vos projets des deux semaines précédentes. Une fois importés, les fichiers sources apparaissent dans des répertoires externes, vous n'avez plus qu'à les déplacer dans l'arborescence de votre projet pour qu'ils soient pris en compte en tant que packages du nouveau projet.

2. Fusion des projets

L'objectif du TME est de faire en sorte que les individus évalués dans la population créée lors du TME2 soient les expressions créées lors du TME3. Le mécanisme d'évaluation est celui de la comparaison avec une fonction cible tel que vous l'avez réalisé au TME3.

Pour réaliser cette fusion, vous avez deux problèmes à résoudre, que vous allez traiter successivement :

- faire en sorte que les expressions soient vues comme des individus par la population,
- faire en sorte que l'environnement d'évaluation des individus soit adapté à votre objectif : trouver une fonction qui passe par un point.



_

2.1 Expressions et Individus

Nous aimerions faire en sorte que les expressions soient manipulées au sein de l'algorithmique génétique du TME 2. Essentiellement il faut donc donner un sens aux concepts génétiques quand les individus constituant une population sont des expressions.

On aimerait que le type *Expression* étende le type *Individu*. Mais *Expression* est une interface alors qu'*Individu* est une classe qui contient un attribut *fitness*. Il faudrait donc transformer *Individu* en interface (ce qui est une bonne idée) et déporter les attributs et comportements correspondants dans les classes qui implémentent *Expression*. Cette approche peut fonctionner, mais elle présente deux inconvénients : on duplique le code de gestion de la fitness dans toutes les classes qui implémentent *Expression* et, surtout, tous les atomes qui composent une expression complexe se voient attribuer une fitness, ce qui n'a pas de sens du point de vue de l'application. D'autre part, on peut considérer que ce serait intrusif : cela forcerait à remettre en cause la base de code établie dans le TME3, alors qu'elle remplissait pleinement son rôle : gérer des expressions arithmétiques.

Une autre approche consiste à conserver *Individu* en tant que classe portant la fitness et dire que les classes qui implémentent *Expression* étendent cette classe. De nouveau cela nécessite de remettre en cause le package gérant les expressions arithmétiques.

Vous allez au contraire vous appuyer sur de la délégation, en introduisant une expression arithmétique dans les individus. On souhaite au maximum NE PAS MODIFIER le package gérant les expressions arithmétiques développé au TME 3. On s'appuiera donc uniquement sur l'API publique du package arithmétique, c'est-à-dire l'interface *Expression*, la classe *ExpressionFactory* pour construire des expressions et la classe *EnvEval* pour gérer les valeurs des variables et évaluer les fonctions.

Sous cette contrainte, vous allez donc plutôt améliorer votre package p*obj.algogen*, sachant qu'il a été construit de façon très brute (pas d'héritage, pas d'interfaces...). Dans un premier temps, vous allez introduire les expressions dans les individus de la population.

Modifiez la classe *Individu* pour que sa valeur propre soit une expression au lieu d'un simple *double* comme au TME 2. Les points critiques qu'il faut implémenter sont les deux méthodes d'évolution: *muter()* et *croiser()*.

Pour la mutation, vous pourrez simplement générer une nouvelle expression aléatoirement.

Pour le croisement, par analogie à l'approche du TME 2, vous produirez comme descendant de deux expressions e1 et e2 une nouvelle expression e3 qui est leur moyenne : *e3 = (e1 + e2) / 2*. Cette expression sera construite explicitement, en utilisant à la racine l'opérateur binaire « / » avec, comme fils gauche un opérateur « + » dont les fils sont e1 et e2 et comme fils droit la constante 2.

Pour les deux méthodes *muter()* et *croiser(),* vous ferez attention à la gestion des références ou des copies sur les objets : il ne faudrait pas que, en mutant un descendant, vous fassiez aussi muter simultanément un de ses parents (voir la question 3.1 à ce sujet).

2.2 Environnement d'évaluation

Pour évaluer les expressions, il faut mettre à jour l'environnement d'évaluation mis en place au TME 2. Il faut donc implémenter la fonction : *double eval(Individu i).*

Vous allez munir l'environnement d'un *EnvEval* qui donne la valeur des variables au point cherché et une valeur (un *double*) qui donne la valeur cible de la fonction en ce point. Mettez à jour le code de la classe *ValeurCible* pour qu'elle rende l'inverse du carré de la différence entre la valeur de la fonction au point considéré et la valeur cible. Cette valeur retournée sera utilisée comme fitness pour les expressions (plus la valeur de l'expression est proche de la valeur cible, plus la fitness est élevée).

2.3 Exécution

Reprenez le *main()* élaboré au TME 2 et mettez-le à jour pour manipuler des expressions. Pour cela, vous pourrez si nécessaire mettre à jour l'opération de la *Factory* du TME 2.

Testez votre programme. Vous devez observer une certaine convergence et, si vous laissez trop de générations passer, une explosion de la taille des expressions. Ce deuxième point sera traité dans la suite.

3 Simplification des Expressions

Pour éviter un accroissement trop rapide de la taille des expressions, vous allez les simplifier au fur et à mesure de l'évolution.

Pour cela, vous allez ajouter une fonction `public Expression simplifier()` à l'interface *Expression* et ses sous-classes.

Une règle de simplification élémentaire consiste à simplifier les sous-arbres constants. Pour cela, on simplifie les deux opérandes d'un opérateur binaire puis, si l'on obtient des constantes, on opère le calcul et on remplace l'opérateur binaire par la constante résultante. Les parties contenant des variables ne sont pas simplifiées. Ecrivez cette fonction récursive, réfléchissez pour déterminer où il faut la brancher dans l'architecture et testez-la.

Attention, on souhaite qu'il n'y ait pas de dépendances en mémoire entre l'expression d'origine et sa version simplifiée, pour que d'éventuelles modifications de l'expression simplifiée ne modifient pas son parent. En effet, cela aurait des effets fâcheux sur la population, si des modifications d'instances de type *Expression* étaient permises. Vous ferez donc en sorte que les expressions soient

immutables, ce qui implique que tous leurs attributs soient déclarés *final*. Utilisez systématiquement la *Factory* pour produire de nouvelles expressions plutôt que des invocations directes à *new*.

Une deuxième règle de simplification consiste à éviter de construire la moyenne de deux expressions égales : on peut remplacer *(e1 +e1) /2* par *e1*. Pour cela, il faut tester dans la méthode *muter()* si les expressions sont égales. Quelles méthodes standards doivent implémenter les expressions pour qu'on puisse comparer leur égalité ? Réalisez la simplification décrite.

Question 3.1

La classe *Expression* est immutable. En d'autres termes, il n'existe pas de moyen de modifier une expression à travers son API publique. Si l'on n'avait pas fait ce choix, il faudrait ajouter une opération de copie profonde d'une expression, pour assurer la cohérence de la population. A quel moment faudrait-il copier les expressions si elles n'étaient pas immutables ? Comment déclarer et implémenter une telle opération ?

Remise du TME

Question : donnez la réponse aux questions ci-dessus. Par ailleurs, copiez-coller le code de votre classe principale et une trace d'exécution de votre programme pour le cas où vous évaluez une population de dix individus constitués d'expressions à deux variables.