TME 2 : Évolution d'une Population sous Éclipse

Objectifs pédagogiques : découverte de l'environnement de programmation eclipse, hiérarchie simple de classes.

Lors du TME 2, nous avons créé aléatoirement une population d'individus. Chaque individu est doté d'une valeur propre représentant ses caractéristiques et d'une valeur de fitness (initialisée à 0) représentant son niveau de compétence relativement à un but donné.

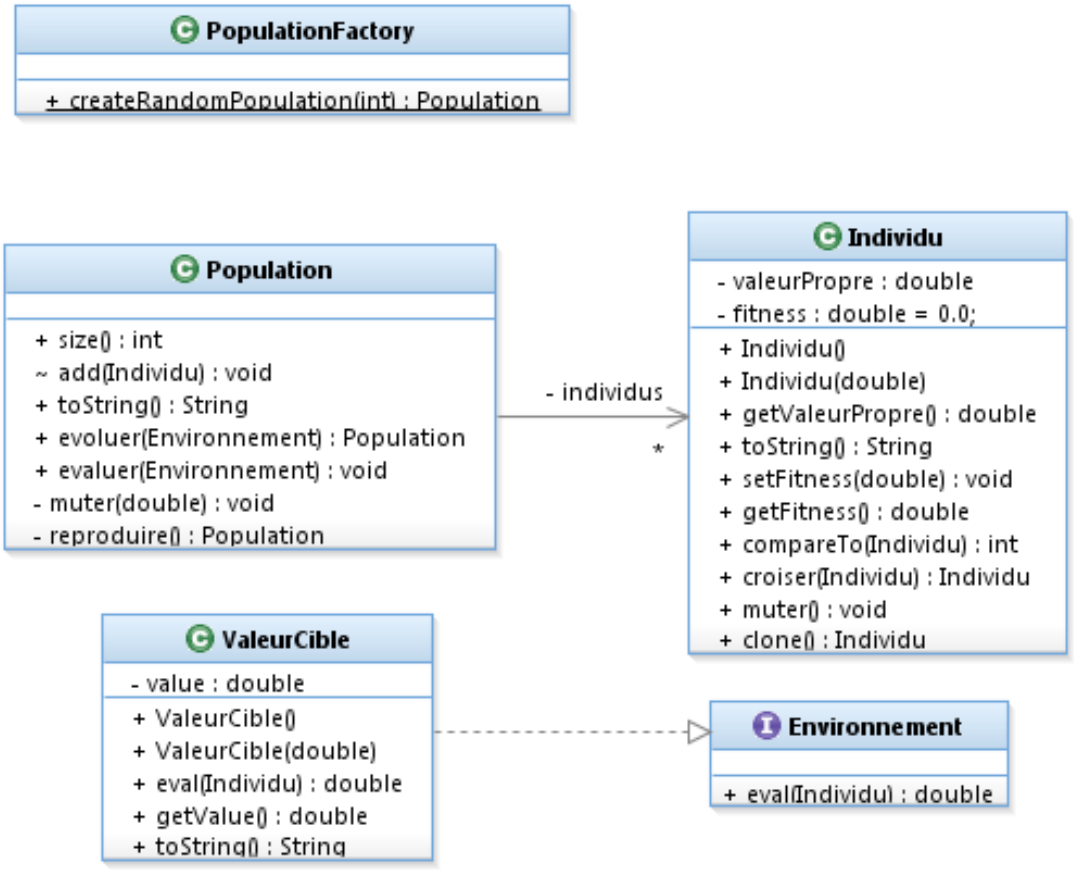

Figure 1: Diagramme UML des classes.

Dans ce TME, l'objectif est d'implémenter les classes et méthodes nécessaires à l'évaluation d'une population relativement à un but donné. Il sera ainsi possible d'identifier les meilleurs individus et de ne conserver que ceux-ci pour engendrer une nouvelle génération (que l'on espère plus performante). L'objectif global du TME est donc d'ajouter à la classe `Population` la méthode :

Population.java

```
/**
 * Permet de faire évoluer la Population en produisant une nouvelle génération.
 * La fonction primordiale de la Population est de pouvoir evoluer.
 * On passe un Environnement qui permettra de calculer le fitness des individus,
 * afin de décider lesquels sont les plus aptes (survival of the fittest).
 * On garde ici les 20% meilleurs et on les fait se reproduire pour générer la
    prochaine génération.
 * @param cible l'objectif/probléme à résoudre/environnement conditionnant l'évolution
    .
```

```
     * @return une nouvelle Population, dont les membres sont tous nouveaux (aucun
         individu de cette Population n'appartient à la Population "this").
     */
    public Population evoluer (Environnement cible) {
```

2.1 Création de projet

⇒ Lancez l'environnement de développement eclipse (tapez « `eclipse &` » en ligne de commande ou cherchez cette application dans le menu des applications).

⇒ Créez un nouveau « projet java » que vous appellerez « tme2 » à partir du menu Fichier/ nouveau/...

Le paramétrage par défaut pour créer un projet java est normalement suffisant pour vos besoins, mais passez quelques instants à examiner les options qui vous sont proposées.

Important :

- Séparez les sources de votre programme des autres répertoires de sortie : cf. dernière option du second écran de création de projet,
- Vérifiez bien que le JRE (Java Runtime Environment) 1.6 ou 1.7 est sélectionné.

Pour importer les fichiers développés lors de la première séance, le plus simple est de les copier dans le projet nouvellement créé (dans le répertoire `src/`), puis de forcer eclipse à rafraîchir (Menu *File* → *Refresh*, ou F5 lorsque vous avez sélectionné le projet).

Remarques

- La position de votre projet dans le système de fichier est visible par un clic droit sur le projet puis menu *Properties*.
- Eclipse compile les sources à la volée. Lorsqu'il n'y a plus d'erreur de compilation (en rouge dans votre code), vous pouvez lancer l'exécution de votre méthode `main()`. Pour cela, positionnez-vous sur la classe qui contient cette méthode et faites *bouton-droit / « Exécuter en tant qu'application Java »*. Vous pouvez également cliquer sur le bouton vert *« play »*. On doit vous proposer de l'exécuter en tant qu'application Java. Pour ajouter des arguments, entrez-les dans l'onglet arguments de la fenêtre *Run → Run Configurations*.

2.2 Ajout d'un environnement

L'évaluation d'une population vis-à-vis d'un problème donné est dépendante de l'environnement dans lequel la population évolue. Ce dernier conditionne en effet l'évaluation de la *fitness* des individus.

A titre d'exemple, l'évaluation des qualités de déplacement d'une population nécessite de disposer d'un environnement où les individus peuvent se déplacer et où l'on est en mesure d'évaluer la distance parcourue. Si l'on passe d'un environnement terrestre à un environnement aquatique, les résultats des individus ne seront pas les mêmes, bien que la mesure (la distance parcourue par exemple) reste inchangée. Ainsi l'évaluation de la *fitness* d'un individu dépend de l'environnement ciblé.

Dans un premier temps, l'environnement considéré portera simplement une valeur cible comprise entre 0 et 1. La mesure de fitness d'un individu vis-à-vis de l'objectif considéré est une mesure de distance entre la valeur cible de l'environnement et la valeur propre de l'individu :

$$fitness(indiv, env) = \frac{1}{(cible.valeur - indiv.valeurPropre)^2} \quad (1)$$

⇒ Écrivez une interface `Environnement`, avec son opération `public double eval(Individu i)`.

Cette opération doit rendre la fitness de l'individu « i » dans l'environnement courant. Pour rappel, une interface n'implémente aucune méthode, elle définit simplement un contrat (un ensemble

de méthodes) que devront respecter (implémenter) toutes les classes qui l'implémentent (mot clé `implements`).

⇒ Écrivez la classe `ValeurCible` qui implémente l'interface `Environnement`.

Remarque : Appuyez-vous sur les opérations (`static`) de la classe `Math`, pour le calcul de la fitness au sein de la méthode `eval(Individu i)`. cf. la documentation de l'API.

2.3 Evaluation d'une population

Vous allez à présent pouvoir évaluer une population.

⇒ Ajoutez à la classe `Population` une opération: `public void evaluer(Environnement cible)`. Cette opération doit calculer et mettre à jour les fitness de tous les individus par l'appel de la méthode `eval()` précédemment implémentée. **NB: cette opération va donc modifier les individus constituant la population sur laquelle on l'invoque, en mettant à jour leur fitness.**

⇒ Utilisez la classe `PopulationMain` pour tester les interactions entre l'environnement et la population. Dans le `main()`, créez donc une population aléatoire et évaluez-la par rapport à un environnement.

2.4 Tri d'une Population

Pour réaliser l'algorithme génétique, il faut pouvoir trier la population par fitness. Pour ce faire, vous allez utiliser les mécanismes standards offerts en Java.

Lisez la documentation de `Collections.sort()`. Cette méthode de nature `static` permet de trier toute collection, en particulier les listes utilisées pour stocker la population. Notez que son équivalent pour les simples tableaux existe sous plusieurs variantes dans `Arrays.sort()`. La documentation nous précise que l'ordre utilisé pour trier les objets est l'ordre dit « naturel » de Java, donné par la fonction standard `compareTo()`.

⇒ Ajoutez à la classe `Individu` le fait qu'un individu soit `Comparable` à un autre individu en ajoutant la mention `implements Comparable<Individu>` à la déclaration de la classe. On vous force alors à ajouter la méthode `public int compareTo(Individu o)`.

⇒ Implémentez la méthode `compareTo()`. Celle-ci doit rendre -1, 0 ou 1 selon que l'objet courant est plus petit, égal ou plus grand que celui passé en paramètre. NB: Les types Java standard (`String`, `Double`...) sont déjà `Comparable`, pour les types simples (`int`, `float`...) on utilisera les méthodes static « `compare` » de la classe correspondante (`Integer`, `Float`...) par exemple. Par exemple `Integer` porte l'opération « `static int compare( int x, int y)` » qui compare x et y numériquement.

⇒ Mettez à jour la méthode `evaluer()` de la population pour qu'elle trie la population par fitness et testez-la dans votre méthode `main()`. On affichera pour cela les quelques premiers individus de la population, pour s'assurer qu'ils sont bien rangés par fitness décroissantes.

2.5 Évolution des individus

Vous êtes maintenant capables d'évaluer les individus afin d'identifier les plus performants. Vous allez à présent faire évoluer la population d'individus de génération en génération comme présenté dans la page de présentation du projet. Vous allez donc programmer des mécanismes de croisement et de mutation (inspirés des opérations de recombinaison qui opèrent dans les chromosomes des êtres vivants) au sein de la population en sélectionnant les individus en fonction de leur fitness.

⇒ Implémentez l'opération de mutation: `public void muter()` et testez-la sur un individu au sein du `main()`. Elle consiste à remplacer un objet d'un certain type par un objet du même type, mais doté d'une autre valeur. Cette opération suppose de connaître l'ensemble des valeurs possibles. Dans le cas présent, la mutation doit mettre à jour la valeur propre de l'individu de façon aléatoire. On pourra choisir de simplement tirer une nouvelle valeur propre pour l'individu, ou de tempérer la mise à jour de la valeur propre en prenant en compte sa valeur actuelle (par exemple, une variation

de plus ou moins dix pour cent de la valeur actuelle...). **NB: Cette opération modifie l'individu sur laquelle on l'invoque.**

⇒ Implémentez le croisement entre deux individus : `public Individu croiser(Individu autre)` et testez celui-ci avec deux individus dans le `main()`. Cette opération est plus complexe que la mutation, car elle permet de créer un nouvel individu à partir de deux individus existants. Plus précisément, étant donné deux individus, le nouvel individu combine une partie des éléments du premier avec une partie des éléments du second. **NB: à la différence de « muter », croiser crée un NOUVEL `Individu`, sans modifier aucun des deux parents (« this » dans le contexte de la méthode et « autre »).**

Cette opération ne doit modifier aucun des deux parents. On pourra par exemple créer un individu dont la valeur propre est la moyenne des valeurs propres des parents, ou une variante sur ce thème.

2.6 Copie des individus

On souhaite également pouvoir copier les individus d'une population d'origine dans une nouvelle population.

⇒ Ajoutez une méthode `public Individu clone ()` à la classe Individu qui doit rendre une nouvelle occurrence d'Individu identique à l'objet courant `this`.

2.7 Évolution des populations

Vous allez à présent réaliser l'opération de reproduction d'une population pour créer la génération suivante. Les différentes générations devront comporter autant d'individus (taille de la population identique).

⇒ Commencez par implémenter une méthode `private void muter(double prob)` dans la classe Population qui fait muter les individus la composant avec une probabilité de `prob` (i.e. chaque individu a `prob` chances de muter). Le premier individu de la population (supposé le meilleur) sera systématiquement épargné; on ne veut jamais s'écarter de la meilleure solution trouvée.

⇒ Programmez ensuite la méthode `private Population reproduire()` qui crée une nouvelle population à partir de la population courante. En supposant que la population est déjà triée (via évaluer)

- On crée une nouvelle occurrence de Population pour loger le résultat
- On y ajoute des clones des 20 % meilleurs individus de la population actuelle.
- Pour les 80% restant on crée de nouveaux individus en croisant des individus choisis aléatoirement parmi les 20 % meilleurs.
- Le résultat est une nouvelle Population qui comporte autant d'individus que la précédente.

Attention à bien réaliser des copies des individus de la population initiale, deux populations différentes ne doivent pas contenir des références sur le même individu en mémoire.

⇒ Compléter enfin le code de la méthode `evoluer()` décrite au début de cet énoncé de TME. Elle doit évaluer la population, puis se reproduire et enfin faire muter le résultat avec une probabilité assez faible (5 à 15 %). On pourra relancer une évaluation sur le résultat pour maintenir la Population triée vis à vis de l'Environnement.

⇒ Testez l'ensemble, en générant une population aléatoire et en choisissant une valeur cible particulière. Produisez un affichage des générations successives (ou au moins des quelques meilleurs individus de chaque génération). Vous devez observer une convergence rapide des valeurs propres des individus vers la valeur choisie.

⇒ Comparez votre code au diagramme de classes au début de cet énoncé.

2.8 Rappel : création d'une archive et remise du TME (Impératif)

Le TME que vous venez de réaliser va resservir par la suite. Il est donc impératif de le terminer.

Chaque semaine, il est obligatoire de rendre une version par email à votre chargé de TME avant la fin de la séance. Si vous le souhaitez, vous pouvez aussi rendre une seconde version améliorée avant le début du TME suivant.

Pour rendre votre TME mais aussi pour le réutiliser dans des TME ultérieurs, il faut créer une archive `jar`. Pour créer l'archive sous eclipse, il suffit d'exporter le projet développé (clic droit sur le projet→Export→Java→JAR file) sous forme d'un fichier jar. Utilisez le format indiqué : `pobj.algogen-Nom1-Nom2.jar`.

NB : il suffit d'exporter les fichiers sources java, sans les binaires ni le javadoc éventuellement créé, ces derniers éléments peuvent être générés à l'ouverture de votre projet.

`NB :` pour générer le javadoc sous eclipse: clic droit sur le projet→Export→Java→Javadoc.

Par ailleurs, chaque TME comporte une question à laquelle il vous est demandé de répondre par écrit dans le mail que vous envoyez à votre encadrant.

Cette règle de remise des TME ne sera plus rappelée sur les sujets suivants, pourtant elle reste valable.

Copiez-collez dans votre mail le code de votre méthode de croisement entre deux individus. Donnez une trace d'évolution de la fitness au sein de la population sur 10 générations. Ajoutez une trace d'exécution.