TME 9 : Pattern Singleton

Objectifs pédagogiques : pattern Singleton, configuration d'un programme, sérialisation, chargement de fichiers

Créez un projet « tme9 ». Importez l'archive de votre TME7.

1. Générateur aléatoire unique

Quand on utilise des méthodes adaptatives, on est souvent content de pouvoir rejouer exactement la même exécution que précédemment en maîtrisant la séquence des nombres aléatoires qui ont été générés par le programme.

A l'heure actuelle, si on relance le programme deux fois de suite, on obtient deux résultats différents.

Pour maîtriser la séquence des nombres aléatoires, il est possible de passer à un objet de la classe *Random* (avec la méthode *setSeed(long graine)*) une « graine » à partir de laquelle sera engendrée la séquence des nombres suivants. Si on lui passe deux fois la même graine, on obtient deux fois la même séquence de nombres. Le vrai aléatoire n'existe pas vraiment en informatique, tout est toujours reproductible.

Mais, dans l'état actuel du programme, nous avons de nombreuses instances distinctes de générateurs de nombres aléatoires répartis dans diverses classes, ce qui complique la génération d'une séquence unique à l'aide d'une graine passée en paramètre au programme.

Pour résoudre ce problème, vous allez faire appel au pattern *Singleton*. Vous allez créer notre propre classe de génération de nombres aléatoires qui ne pourra être instanciée qu'une seule fois et qui porte un attribut de type *Random*. Vous utiliserez la délégation pour que votre générateur dispose des méthodes *nextInt()* et *nextDouble()* usuelles. Vous passerez la graine à votre générateur via un accesseur que vous appellerez dans la méthode *main()* de la classe principale de votre logiciel.

Dans le package *pobj.util*, créez une classe *Generateur* qui réalise ce principe. Eliminez toutes les instances de *Random* de votre programme en invoquant à la place des opérations du singleton *Generateur* (attention à aussi remplacer les invocations à *Math.random()* qui utilise son propre singleton*)*. Utilisez une graine et vérifiez que votre programme donne deux fois de suite le même résultat.

2. Fichier de configuration

Le jeu de paramètres qu'accepte la méthode *main()* de votre logiciel commence à être conséquent. On va créer un mécanisme de configuration flexible, s'appuyant sur des fichiers.

Votre fichier de configuration contiendra les informations suivantes :

TypeIndividu : Agent

Labyrinthe : big.txt

taillePop : 100

nbGens : 70

etc.

Implantez une classe de configuration munie des opérations : *String getParameterValue(String param), void setParameterValue(String param, String value).* Vous la munirez d'une

Map<String,String> pour assurer le stockage des valeurs de paramètres. Déclarez une interface *AlgoGenParameter* et ajoutez-y des constantes de type *String*, pour loger les noms des paramètres légaux dans votre application (e.g. *public static final String TAILLE_POP = « TaillePopulation »;*).

Utilisation : modifiez votre méthode *main()* pour qu'elle lise ses arguments depuis une instance de la classe *Configuration*. Vous créerez cette instance manuellement.

3. Sauvegarde/Chargement

Ajoutez des opérations permettant la sauvegarde et le chargement d'une configuration. Vous vous appuierez sur la sérialisation. Pour cela, vous pouvez vous inspirer des méthodes fournies dans *agent.laby.ChargeurLabyrinthe.java*

Pour éviter d'avoir à spécifier ce jeu de paramètres manuellement en ligne de commande à chaque fois, créez un fichier de configuration dont le nom sera l'unique paramètre passé en ligne de commande à votre application.

Déportez le code de votre méthode *main()* dans une fonction qui prend une configuration en paramètre. La méthode *main()* créera la configuration et la passera à cette fonction.

4. La configuration comme Singleton

On souhaite à présent pouvoir accéder à la configuration de tout point du code de l'application. On suppose qu'à un instant donné il n'existe qu'une seule configuration active.

Implémentez le *Design Pattern* Singleton, pour offrir l'unique instance de la classe *Configuration* dans l'application via une méthode *static* de la classe *Configuration : Configuration getInstance()*;

Vous pourrez à ce stade ajouter à votre classe *Configuration* divers éléments : stratégie de contrôle des agents, algorithme élitiste (on garde les 20% meilleurs de chaque génération) ou pas, ...

Ces points de variation/configuration de votre application peuvent s'appuyer sur la classe *Configuration* via le Singleton, sans remettre en cause l'architecture de votre application (signatures actuelles préservées...). Mais on a ainsi une façon de positionner finement des paramètres de configuration en profondeur dans l'application, sans nécessairement en informer les couches hautes.

En règle générale, chaque package peut définir sa propre interface *MesConstantesParameter* avec des constantes désignant les noms des paramètres de configuration qu'il utilise.

Optionnel: Vous pourrez développer une classe *Swing* pour afficher ces paramètres et les mettre à jour. Vous réaliserez pour cela un *ConfigurationPanel*, muni de paires *nomParametre:JLabel* et *valeurParam:JText*. On pourra s'inspirer du code de *LabyPanel* et on utilisera une *HashMap<String, JtextField>* pour stocker les références aux champs de saisie.

5. Ajout de génériques : QUESTION BONUS

Le souci qui persiste après la montée en abstraction du TME5 est le lien qui existe nécessairement entre un environnement qui évalue les individus et les individus eux-mêmes. Cela se voit par le problème de la signature de la méthode *getValeurPropre()* utilisée par l'environnement pour obtenir un *double* au TME2 et une *Expression* au TME4.

Une solution consiste dans le *eval(Individu i)* d'un environnement à raffiner le type d'*Individu* (down-cast) via un test *instanceof*.

Par exemple :

```
/** Mesure l'écart entre la valeur propre de l'individu et la valeur cible*
```

```
public double eval(Individu i) {

      if (i instanceof IndividuExpression) {

            Expression e = ((IndividuExpression)i).getValeurPropreExpression();
```

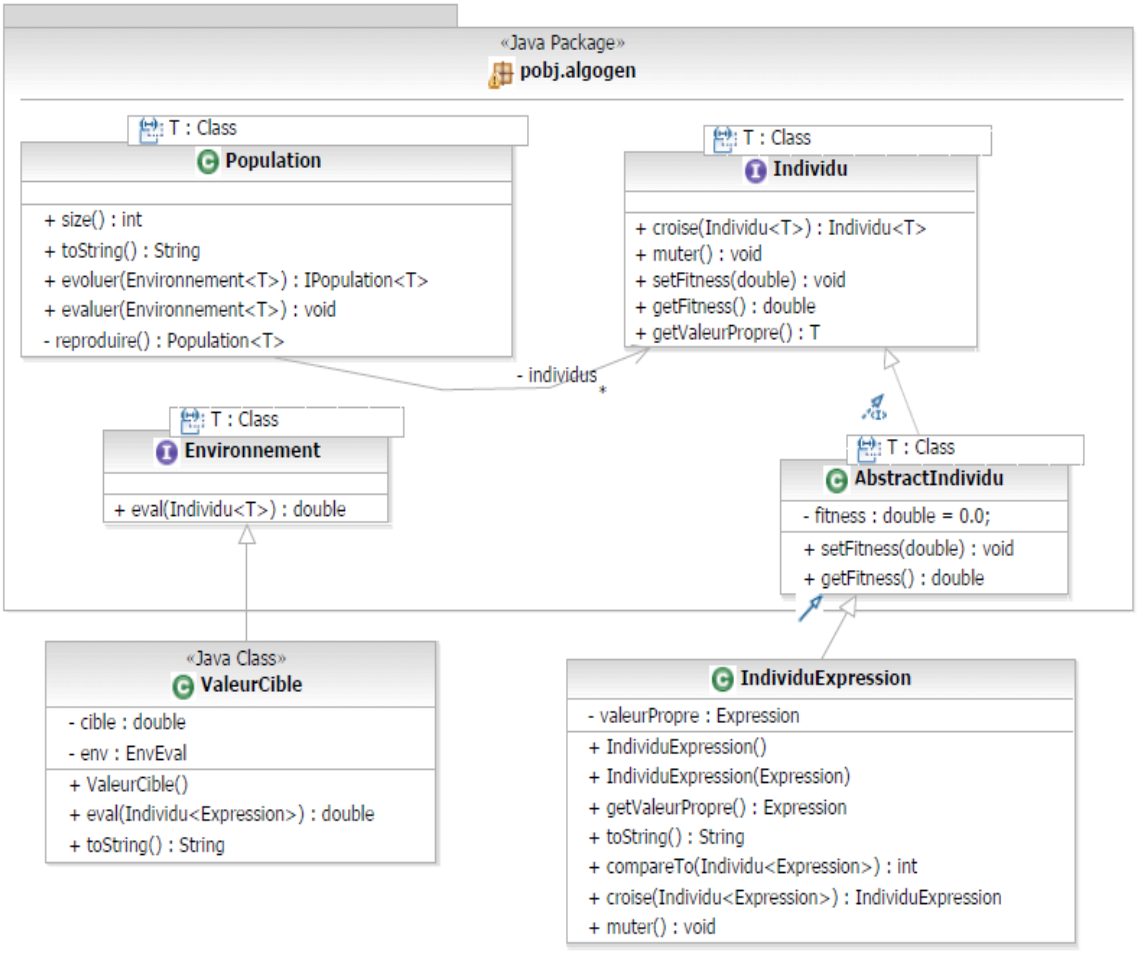
On peut se contenter de cette solution, certes inélégante, mais fonctionnelle. Il faut juste veiller à bien choisir les individus et les environnements de façon congruente. Cela dit, ce type d'expression est dangereuse et désagréable à manipuler. Mais les génériques permettent de lutter contre ce problème de typage.

Ajoutez un paramètre générique *T* à l'interface *Individu*. Ainsi on aura des *Individu<Expression>* des *Individu<Double>* ou des *Individu<Controleur>*. De fait, ce paramètre donne le type de retour de la méthode *getValeurPropre()*.

Attention, il faut propager cette modification dans *Population*, qui devient une *Population<T>*, contenant une *List<Individu<T>>* afin de correctement typer les opérations d'ajout d'individus. De même, il faut que la classe *Individu<T>* hérite de *Comparable<Individu<T>>* et que *compareTo()* soit implémentée dans la classe *AbstractIndividu<T>*. Enfin, la classe *IndividuExpression* qui porte l'expression arithmétique déclarera *extends AbstractIndividu<Expression>*, fixant ainsi ses signatures.

L'environnement est lui-même conditionné par un type *T*. Pour vous guider, la figure ci-dessous représente le schéma qu'on cherche à obtenir avec les signatures des opérations appropriées. Comme on le voit, la partie liée aux expressions se découple de la partie évolution. Pour pouvoir faire un algorithme génétique, il suffit de savoir implémenter les interfaces *Environnement* et *Individu*.

Notons que cela est déjà le cas sans les génériques, aux problèmes de typage faible près. Cette solution avec génériques permet même de monter la valeur propre (typée *T*) dans la classe *AbstractIndividu* (avec ses accesseurs *get/set*), ne laissant plus que la responsabilité d'implémenter *mute()* et *croise()* à l'utilisateur du package qui souhaite traiter un nouveau type de problème.



6. Remise du TME

Question : Mesurez le temps d'exécution avant et après utilisation de la classe *Generateur*. Constatez-vous une perte d'efficacité significative ? Copiez-collez dans votre mail le code de votre classe *Generateur*. Copiez-collez un fichier de configuration rempli et la trace d'exécution correspondante.