# Algorithme Génétique

Youssef ZIYYAT

Guillaume COURTIN

Nathan DUBERNARD

Steve DESPRES

**EI5 IHMRV**

## Introduction

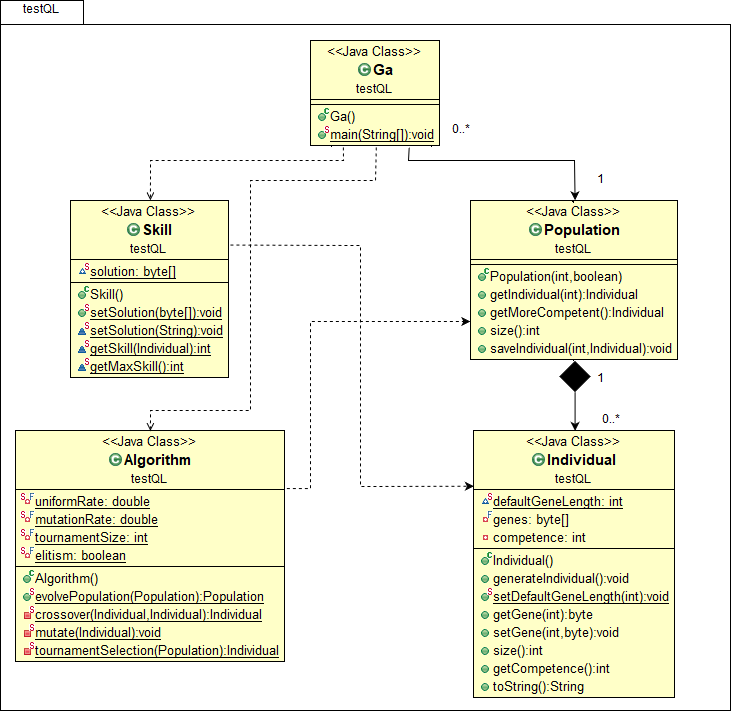
Pour le développement d’une librairie d’algorithme génétique, nous nous sommes basés sur un exemple d’implémentation disponible sur :

[https://www.slauncha.com/index.php?article63/creation-d-un-algorithme-genetique](https://www.slauncha.com/index.php?article63%2Fcreation-d-un-algorithme-genetique&fbclid=IwAR1QT5sXL1bxfwGH6dZbUGQJaB1jupfQ3CEw7BCs6uNKNzTh7ChtKyakCOQ)

Nous avons décidé de développer notre librairie en mettant en pratique différents mécanismes architecturaux et bonnes pratiques vues en cours, et en implémentant l’exemple précédant avec notre librairie développée.

## Exemple d’implémentation d’algorithme génétique

UML :



Les classes :

* **Ga.java** : c’est le *main* de l’application où est exécuté l’algorithme.
* **Algorithm.java** : contient les fonctions relatives à l’algorithme génétique (mutation, croisement
* **Individual.java** : représente un individu
* **Population.java** : représente une population d’individus
* **Skill.java** : permet d’évaluer les individus

Fonctionnement

Dans cet exemple, la population est composée de 50 individus. Les individus sont représentés par un tableau de 64 gènes (tableau de 64 octets) ayant pour valeur 0 ou 1.

L’évaluation des individus est faite par rapport à un objectif (appelé *solution*) : "1111000000000000000000000000001111100000000000000000000000001111"

Une variable *skill* comptabilise les gènes similaires d’un individu par rapport à cet objectif. Elle est calculée par la fonction d’évaluation.

La fonction de mutation est aléatoire.

La fonction de croisement mélange aléatoirement les gènes des parents pour les donner à un individu enfant.

La Sélection des parents se fait aléatoirement ; 5 individus sont tirés aléatoirement dans la population. Parmi ces 5 individus, celui ayant la meilleur compétence (*skill*) est sélectionné pour être un parent.

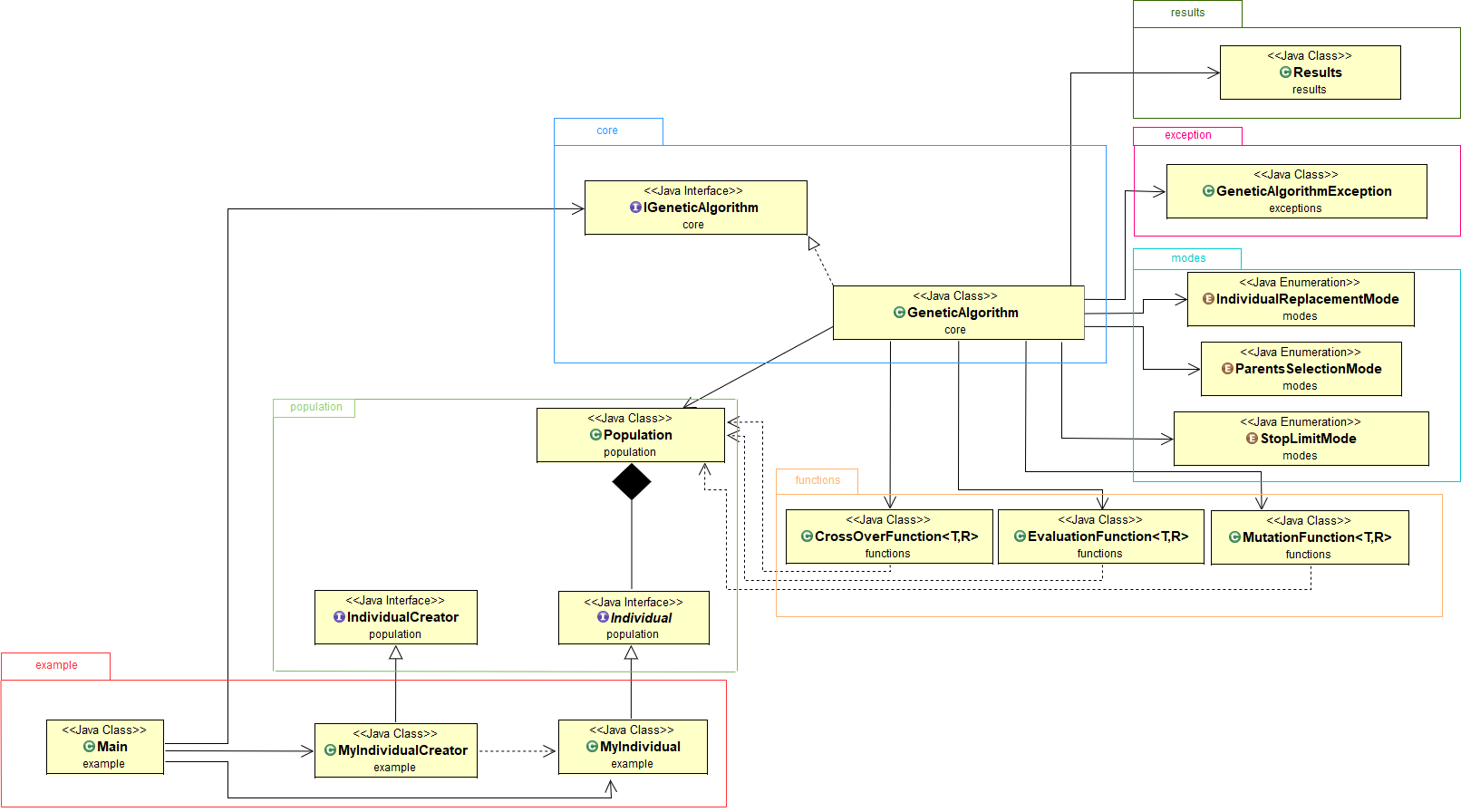
L’algorithme s’arrête lorsqu’un individu présente les mêmes gènes que la solution.

Pour plus de précision, voir le code de l’exemple.

## Librairie développée

UML

Voir l’UML complet en annexe.

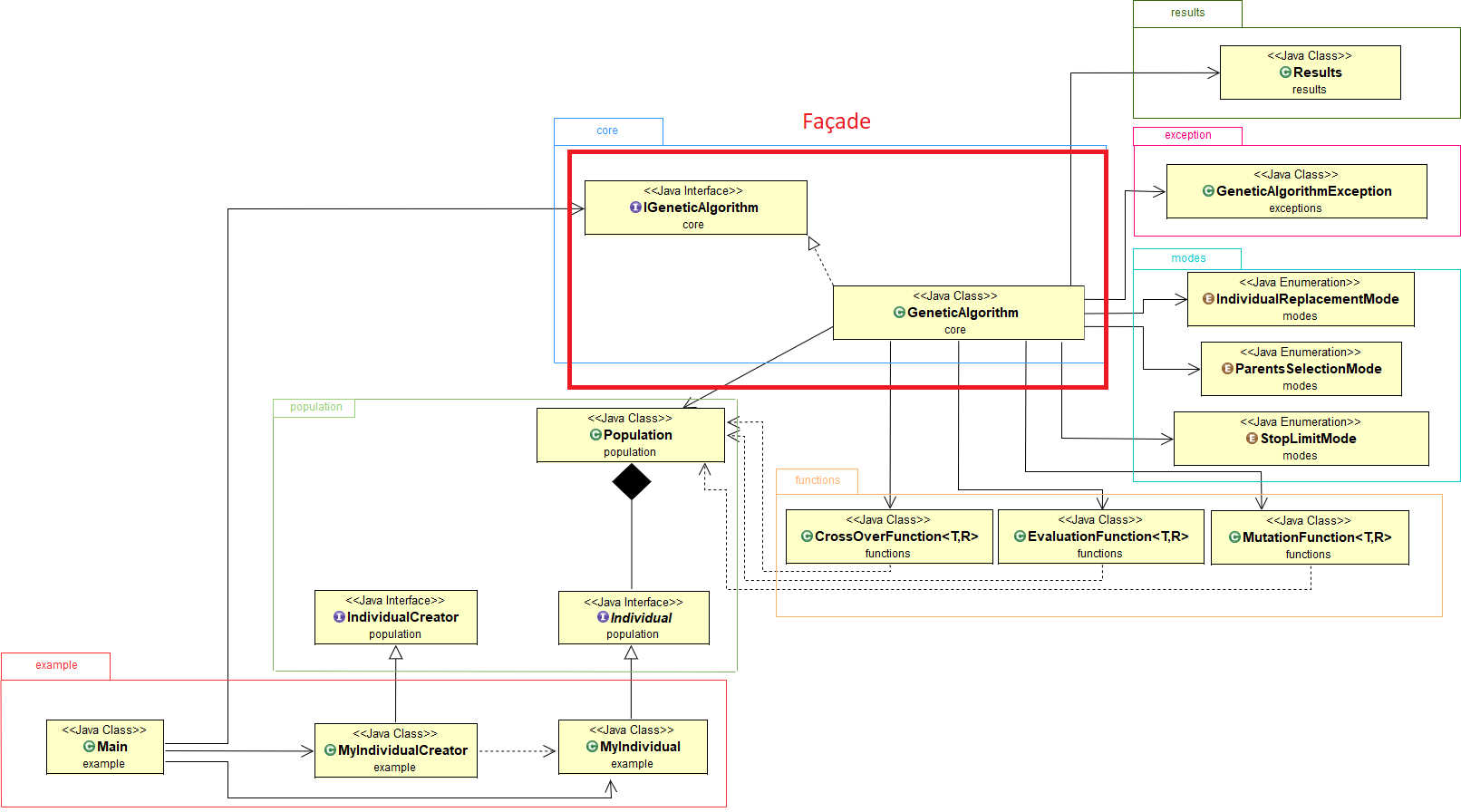


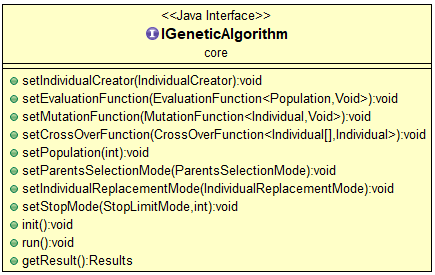
Les classes

* Package **core**
  + *IGeneticAlgorithm*
    - Interface de notre librairie. Permet à l’utilisateur d’initialiser tous les paramètres nécessaires et de lancer l’algorithme.
  + *GeneticAlgorithm*
    - Implémentation de l’interface de notre librairie.
* Package**population**
  + *Population*
    - Représente la population d’individus.
  + *Individual*
    - Interface représentant un individu.
  + *IndividualCreator*
    - Interface représentant le créateur d’individus.
* Package**functions**
  + *MutationFunction*
    - Fonction de mutation : l’utilisateur doit l’implémenter à l’aide d’expressions lambda.
  + *CrossOverFunction*
    - Fonction de croisement : l’utilisateur doit l’implémenter à l’aide d’expressions lambda.
  + *EvaluationFunction*
    - Fonction d’évaluation : l’utilisateur doit l’implémenter à l’aide d’expressions lambda.
* Package**exception**
  + *GeneticAlgorithmException*
    - Permet de gérer les exceptions relatives à la librairie.
* Package**results**
  + *Results*
    - Représente les résultats de l’algorithme génétique.
* Package**example**
  + *MyIndividual*
    - Exemple d’implémentation d’un individu par l’utilisateur.
  + *MyIndividualCreator*
    - Exemple d’implémentation du créateur d’individus par l’utilisateur.
  + *Main*
    - Exemple d’utilisation de la librairie par l’utilisateur

Interface IGeneticAlgorithm : Façade

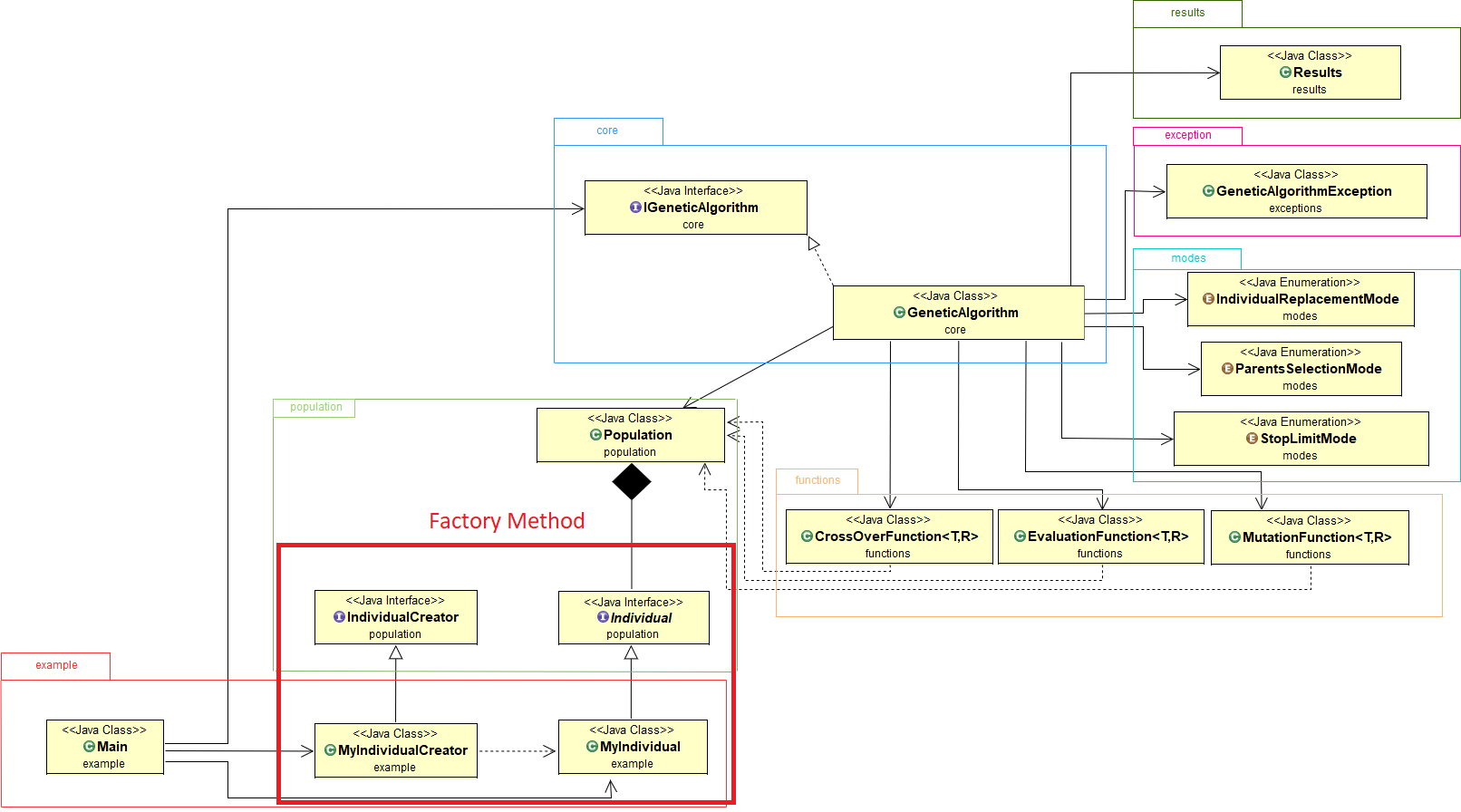
Nous avons mis en place le Design Pattern **Façade** pour notre librairie. L’interface IGeneticAlgorithm permet à l’utilisateur de définir tous les paramètres nécessaires (setters), d’initialiser l’algorithme, de le lancer et de récupérer les résultats.





Création d’individus : Factory Method

Pour la création d’individus, nous avons mis en place le Design Pattern **Factory Method** ; l’utilisateur doit implémenter les interfaces *Individual* et *IndividualCreator* pour définir le modèle de son individu.



Point de vue utilisateur

Pour utiliser la librairie, l’utilisateur doit fournir ses propres fonctions de mutation, de croisement et d’évaluation grâce à des expressions lambda.

Pour définir son modèle d’individu, il doit implémenter l’interface *Individual* pour représenter son individu, et l’interface *IndividualCreator* pour définir la création d’individu.

Pour définir les paramètres de l’algorithme, il doit utiliser les setters disponibles via l’interface *IGeneticAlgorithm*.

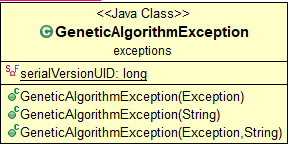
Une fois que tous les paramètres sont définis, l’utilisateur doit initialiser l’algorithme via la fonction **init()** et le lancer via ma fonction **run().**

Une fois l’algorithme executé, il peut récupérer les résultats via la méthode **getResults()** .

Voir le fichier Readme pour plus d’information sur l’utilisation.

Gestion des exceptions

Pour gérer les erreurs possibles dans l’algorithme, par exemple un paramètre non initialisé, nous levons une exception *GeneticAlgorithmException*. Cette classe hérite de la classe java *Exception.*

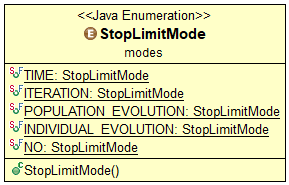
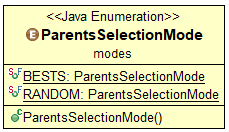
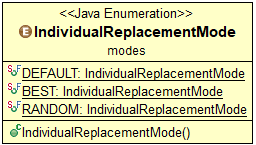
**

Gestions des modes de fonctionnement

Il est possible de définir différents modes de fonctionnement.

* Mode de remplacement des individus
* Mode de sélection des parents
* Mode d’arrêt de l’algorithme

Nous avons représenté ces différents modes grâce à des énumérations. L’utilisateur peut définir chaque mode à l’aide de setters (accessible via *IGeneticAlgorithm*).



(Pour le moment, tous les modes ne sont pas fonctionnels).

Découpage des fonctions : SRP

Nous avons essayé au mieux de respecter le principe de SRP (une fonction = une responsabilité) en séparent les différents rôles des fonctions au sein des classes.

Voir en particulier les fonctions *run()*, *evolvePopulation()* et *selectParent()* dans la classe *GeneticAlgorithm.*

Autres bonnes pratqiues

Pour le développement de cette librairie, nous avons utilisé ces différentes « bonnes pratiques » :

* Nommage homogène et lisibles des packages, classes, attributs, etc…
* Commentaires sur les classes, fonctions et où cela nous semblait utiles. (Javadocs).
* Utilisation de git (GitLab) pour la gestion des versions et le travail en équipe.
* Utilisation d’un logger.
* Utilisation du polymorphisme
* Analyse avec SonarLint

Travail restant

* Parallélisation
* Terminer la gestion des modes
* Tests unitaires avec JUnit et Mockito
* Revue de code entre les membres de l’équipe
* Contraintes sur les entrées utilisateurs (pour plus de sécurité et fiabilité)
* Tests avec différentes entrées utilisateurs (fonctions mutation/croisement/evaluation, Individual et IndividualCreator)

## Annexe

UML Complet de notre librairie.