# **Final Project Report**

**Genetic Algorithm using Strategy, Singleton and Factory design patterns.**

Name: Sakina Fatima

Student Number: 18251051

**Introduction:**

In this project we have used Genetic Algorithm to find an optimal solution for generating an optimal solution. This report will explain this technique thoroughly along with important terms which make this algorithm. Also it will demonstrate how I have implemented this technique with Object Oriented Programming principles and different design patterns. I will also show code snippets where necessary.

**Genetic Algorithm:**

A genetic algorithm is a search heuristic that is inspired by Charles Darwin’s theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. [2]

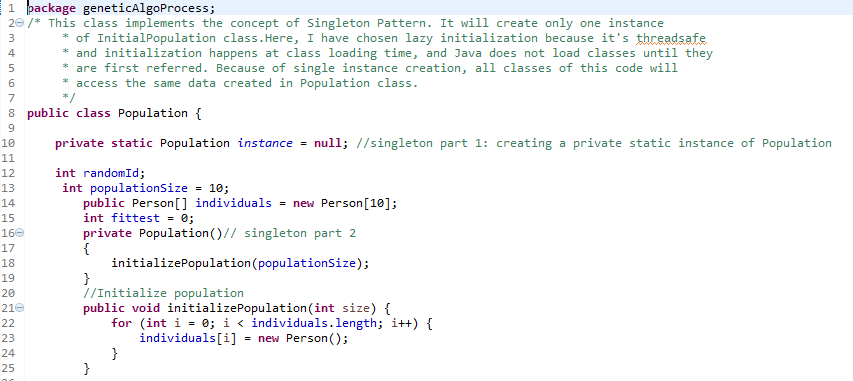
Genetic algorithms are a type of optimization algorithm, meaning they are used to find the optimal solution(s) to a given computational problem that maximizes or minimizes a particular function. [1]

Following are the different important terms in a Genetic Algorithm:

1. **Initial Population**

The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve. [2]

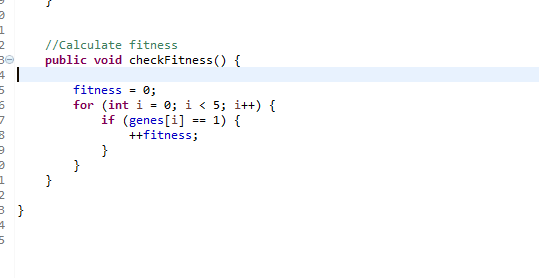
An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution). In a genetic algorithm, the set of genes of an individual is represented using a string. [2] In my program, a chromosome is made up of five binary numbers that are randomly generated.



1. **Fitness Function**

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score. [1]

In our program, from the initial population, each individual’s chromosome is compared with the **target chromosome 11111** and then based on this, a fitness score is calculated.



**3. Selection**

In this phase the fittest individuals will be selected and their genes will be passed to the next generation.

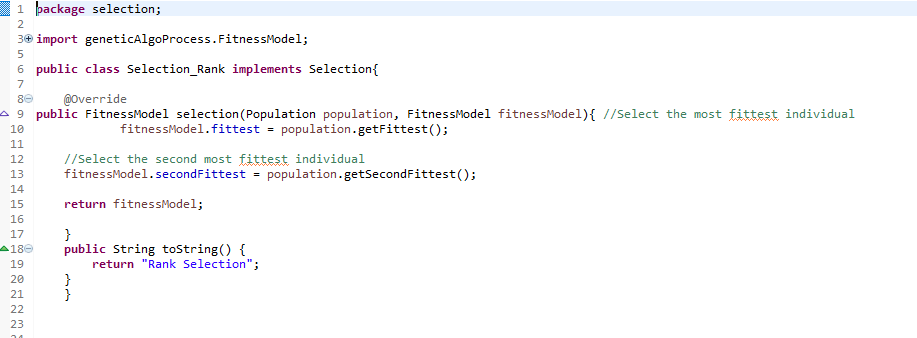
Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction. [2]

In my program there are two types of selection methods are used. i.e. Rank and Tournament Selection which are inherited by Selection\_Interface Class.

**Rank Selection:**

Here all individuals are ranked on the basis of their fitness score. And from them, the two best parents are selected to perform further operations.

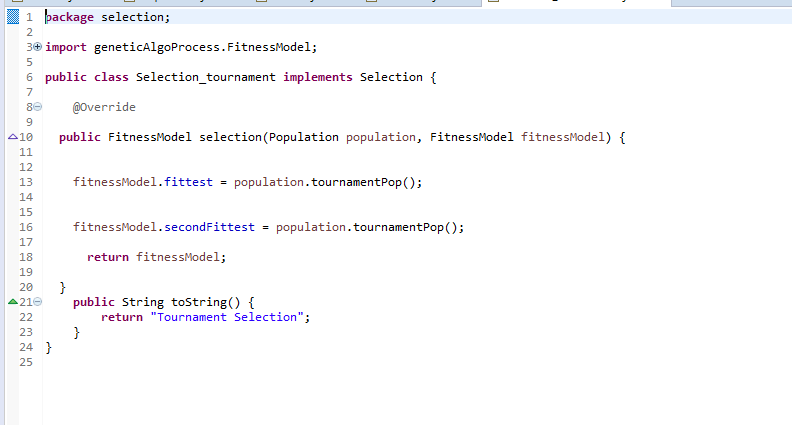
Rank selection first ranks the population and then every chromosome receives fitness from this ranking. The worst will have fitness 1 and the best will have fitness N (number of chromosomes in population). After this all the chromosomes have a chance to be selected. But this method can lead to slower convergence, because the best chromosomes do not differ so much from other ones. [3]

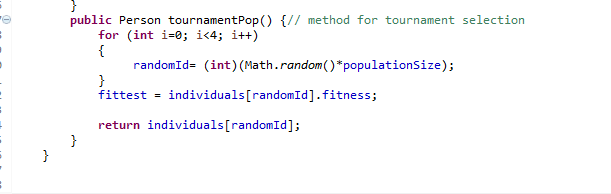


**Tournament Selection:**

Tournament selection returns the fittest individual of some t individuals picked at random, with replacement, from the population. First choose t (the tournament size) individuals from the population at random. Then choose the best individual from tournament with probability p, choose the second best individual with probability p\*(1-p), choose the third best individual with probability p\*((1-p)^2), and so on...[3]

In the program 5 tournaments are conducted.





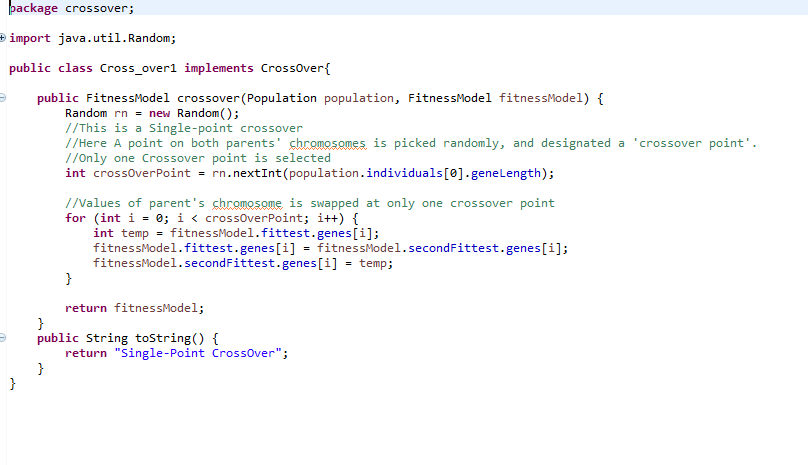
**4. Crossover**

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. [2]

In the program, two types of Crossover techniques are used i.e. Single-point crossover and two-point crossover.

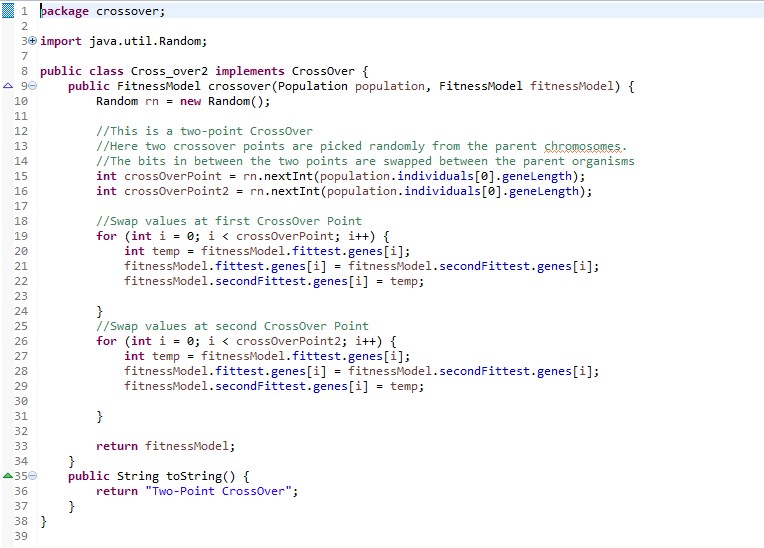
**Single-point crossover**

A point on both parents' chromosomes is picked randomly, and designated a 'crossover point'. Bits to the right of that point are swapped between the two parent chromosomes. This results in two offspring, each carrying some genetic information from both parents. [4]



**Two-point crossover**

In two-point crossover, two crossover points are picked randomly from the parent chromosomes. The bits in between the two points are swapped between the parent organisms. [4]



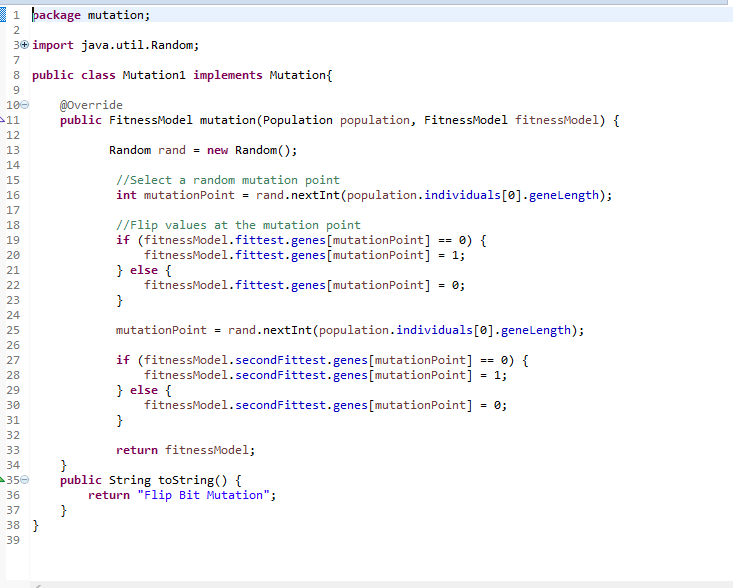
**5. Mutation:**

Last step of this process is Mutation.

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped [2]. In the code two types of mutations are used that are Flip bit Mutation and Boundary Mutation.

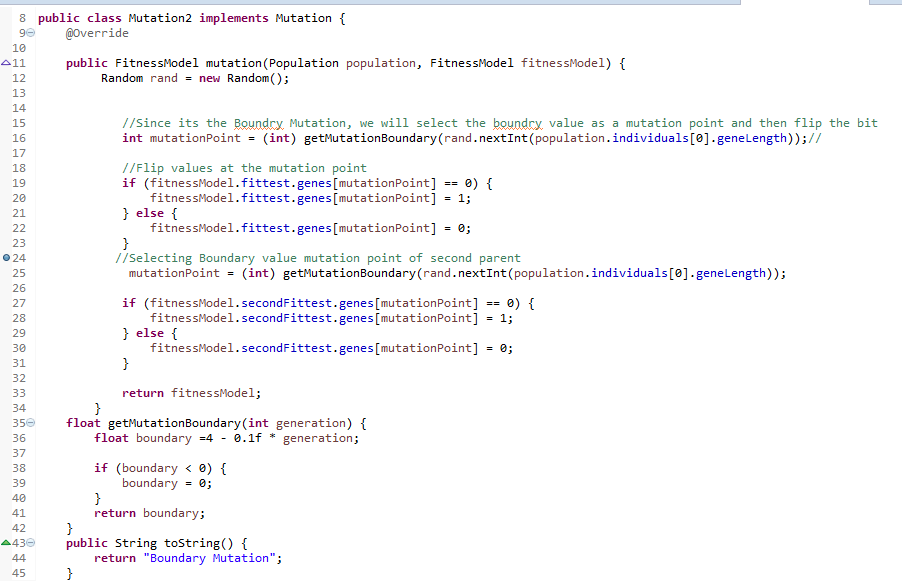
**Flip Bit Mutation:**

This mutation operator takes the chosen genome and inverts the bits (i.e. if the genome bit is 1, it is changed to 0 and vice versa). [5]



**Boundary Mutation:**

This mutation operator replaces the genome with either lower or upper bound randomly. This can be used for integer and float genes.



**Patterns Implemented:**

In this code three design patterns are implemented to optimize the performance of the code and to provide maximum encapsulation and data abstraction. ( All these patterns are clearly mentioned in comments in code)

**1. Singleton Pattern:**

The singleton design pattern is used to restrict the instantiation of a class and ensures that only one instance of the class exists in the JVM. In other words, a singleton class is a class that can have only one object (an instance of the class) at a time per JVM instance. There are various ways to design/code a singleton class. [6].

In the code I am using Lazy Initialization Singleton Method on my population class. So that at one time only one instance of population is created. (Note: It is also commented out in the code that with singleton all instances of population point towards same address in memory).

**2. Strategy Pattern:**

Strategy is a behavioural design pattern that turns a set of behaviours into objects and makes them interchangeable inside original context object. [7]

The original object, called context, holds a reference to a strategy object and delegates it executing the behaviour. In order to change the way the context performs its work, other objects may replace currently linked strategy object with another one. [7]

Now in my program, I have created two Strategies Strategy 1 and Strategy 2; both of them are working on different random values for the selection of the factory to complete the process of genetics. The strategy class is accessed by the context variable **strategy\_selected** (in my runner class) to get the strategy at run time. Currently strategy is being selected randomly but since the client has the control, this strategy can be selected according to the user’s choice as well. This is one the main reason for implementing the strategy pattern so that the selection of the strategy can be changed at run time without affecting the code structure or working. With this pattern, users will be able to play with behaviour of a class without extending it from other classes through inheritance that may lead to the case of class explosion.

**3. Abstract Pattern:**

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. [9]

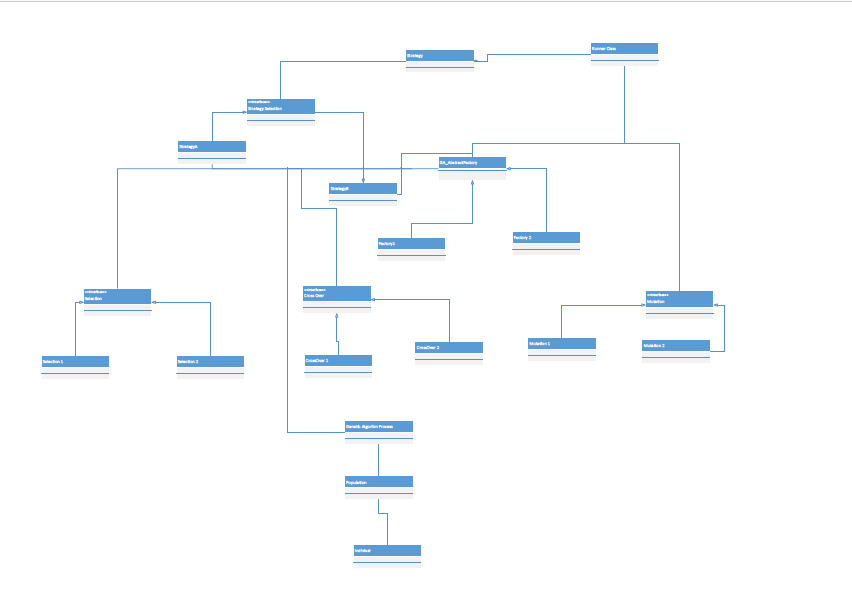
In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern. [8]

In my code GA\_AbstractFactory class serves as my abstract factory which is the parent factory of factory 1 and factory 2. Referring to the example of pizza store from [8] the purpose of GA\_AbstractFactory is to return the objects of selection, mutation, crossover created by its sub factories and then passing it to the GeneticAlgorithmProcess class to further initiate the process.

Now the reason for using abstract factory in my class is that, it can be easily extended to accommodate more products, for example if in future there is more addition to the code, we can add another sub-class like factory 3 without making changes in rest of the code (hence, supporting open-closed principle).

Another general benefit of factory pattern is that it hides the code creation and code logic from user so encouraging code abstraction and encapsulation. As you can see that in my code, objects of mutation, selection and crossover are created inside the factories not at client side and in this way client will not know how these objects are created or what’s the inside logic of the code.

**Code Structure:** (Through Following Diagram, I just want to show a brief structure of the code with different classes, interfaces and the relation between different classes.)



**References:**

[1] Jenna Carr, “*An Introduction to Genetic Algorithms*”, [Online]. Available: <https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf>. [Accessed: 17-11-2018]

[2] “*Introduction to Genetic Algorithms — Including Example Code”,* [Online]. Available: <https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>. [Accessed: 17-11-2018]

[3] “*Enum GeneticAlgorithm.Selection*” [Online]. Available: https://haifengl.github.io/smile/api/java/smile/gap/GeneticAlgorithm.Selection.html#TOURNAMENT [Accessed: 04-12-2018]

[4] “Crossover (genetic algorithm)” [Online]. Available: https://en.wikipedia.org/wiki/Crossover\_(genetic\_algorithm) [Accessed: 22-11-2018]

[5] “Mutation (genetic algorithm)” [Online]. Available: https://en.wikipedia.org/wiki/Mutation\_(genetic\_algorithm) [Accessed: 04-12-2018]

[6] “How to Use Singleton Design Pattern in Java” [Online]. Available: https://dzone.com/articles/singleton-in-java [Accessed: 05-12-2018]

[7] “Java: Strategy” [Online]. Available: <https://refactoring.guru/design-patterns/strategy/java/example> [Accessed: 05-12-2018]

[8] Eric Freeman, HEAD FIRST DESIGN PATTERNS, 05 November 2004

[9] “Design Pattern - Abstract Factory Pattern” [Online] Available: <https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm> [Accessed: 06-12-2018]