**“LUCIAN BLAGA” UNIVERSITY OF SIBIU  
ENGINEERING FACULTY**

**DEPARTMENT OF COMPUTER SCIENCE, ELECTRICAL**

**AND ELECTRONICS ENGINEERING**

**DISSERTATION**

SCIENTIFIC COORDINATOR : Prof. Dr. Ing., Florea, Adrian

GRADUATE:

Aftenie, Ana-Maria

Specialization ACS

* Sibiu, 2024 –

**“LUCIAN BLAGA” UNIVERSITY OF SIBIU  
ENGINEERING FACULTY**

**DEPARTMENT OF COMPUTER SCIENCE, ELECTRICAL**

**AND ELECTRONICS ENGINEERING**

**Implementation of superposition methods for Pareto multi-objective algorithms and inspired by nature algorithms**

SCIENTIFIC COORDINATOR : Prof. Dr. Ing., Florea, Adrian

GRADUATE:

Aftenie, Ana-Maria

Specialization ACS

Contents

[Chapter 1. Introduction 4](#_Toc169690423)

[1.1. Project scope 4](#_Toc169690424)

[1.2. Project objectives 4](#_Toc169690425)

[1.5. Paper structure 5](#_Toc169690426)

[Chapter 2. Technology stack description 6](#_Toc169690427)

[2.1. IntelliJ IDEA 6](#_Toc169690428)

[2.2. Sourcetree 7](#_Toc169690429)

[2.3. Java language 8](#_Toc169690430)

[2.3.1. Comparison with other programming languages 10](#_Toc169690431)

[2.4. jMetal library 11](#_Toc169690432)

[2.4.1. How to install and use it? 13](#_Toc169690433)

[2.4.2. Algorithm types 16](#_Toc169690434)

[2.4.3. Quality Indicators 17](#_Toc169690435)

[2.4.4. Problems 19](#_Toc169690436)

[Chapter 3. Theoretical part of the implementation 21](#_Toc169690437)

[3.1. Pareto concept 21](#_Toc169690438)

[3.1.1. Multi-objective genetic algorithms 22](#_Toc169690439)

[3.1.1.1. Non-dominated Sorting Genetic Algorithm 22](#_Toc169690440)

[3.1.1.2. Non-dominated Sorting Genetic Algorithm II 23](#_Toc169690441)

[3.2. Particle Swarm Optimization algorithms 25](#_Toc169690442)

[3.2.1. Particle Swarm Optimization algorithm description 26](#_Toc169690443)

[3.2.2. Strength Pareto Multi-objective Particle Swarm Optimization algorithm 28](#_Toc169690444)

[3.3. Algorithm’s comparisons 30](#_Toc169690445)

[Chapter 4. Integration of theoretical notions into practical part of the application 32](#_Toc169690446)

[4.1. Project architecture 32](#_Toc169690447)

[4.2. NSGAII algorithm 32](#_Toc169690448)

[4.3. SMPSO algorithm 33](#_Toc169690449)

[4.4. Superposition of the algorithms 33](#_Toc169690450)

[4.4.1. Superposition structure 35](#_Toc169690451)

[4.4.1.1. AbstractGeneticAlgorithm class 35](#_Toc169690452)

[4.4.1.2. GeneticAlgorithmWrapper class 37](#_Toc169690453)

[4.4.1.3. AbstractSuperPositionGA class 40](#_Toc169690461)

[4.4.1.4. AbstractParticleSwarmOptimization class 43](#_Toc169690462)

[4.4.1.5. ParticleSwarmWrapper class 45](#_Toc169690463)

[4.4.1.6. AbstractSuperPositionPSO 48](#_Toc169690464)

[4.4.1.7. SuperPositionNSGAII class 52](#_Toc169690465)

[4.4.1.8. SuperPositionSMPSO class 54](#_Toc169690466)

[4.4.1.9. SuperPositionCombinator1 class 56](#_Toc169690467)

[4.4.1.10. AbstractSuperPositionGAPSOCombinator1 57](#_Toc169690468)

[4.5. Simulations without GAP simulator 60](#_Toc169690469)

[4.5.1. NSGAII simulation 61](#_Toc169690470)

[4.5.2. SMPSO simulation 62](#_Toc169690471)

[4.5.3. SuperPosition of NSGAII and SMPSO simulation 63](#_Toc169690472)

[4.6. Simulations with GAP simulator 65](#_Toc169690473)

[Chapter 6. Conclusions and future work 66](#_Toc169690474)

[Chapter 7. References 67](#_Toc169690475)

# Chapter 1. Introduction

Multi-objective optimization is essential in fields like engineering, finance, and logistics, where decision-makers often have to balance conflicting goals. Traditional optimization methods fail to deal with such situations effectively since they cannot handle multiple goals simultaneously. In this regard, metaheuristic algorithms like NSGA-II and SMPSO have been proven to be valuable approaches for tackling multi-objective optimization problems.

# Project scope

This project aims to explore how combining the NSGA-II and SMPSO algorithms can enhance multi-objective optimization. By merging the genetic algorithm techniques of NSGA-II with the swarm intelligence of SMPSO, a hybrid algorithm that efficiently searches for solutions while maintaining a diverse set of options is created.

# Project objectives

The principle project objectives that should be implemented are the following:

* Analyze the mechanisms of NSGA-II and SMPSO algorithms.
* Design an integration framework to combine NSGA-II and SMPSO components effectively – GAP simulator.
* Validate the algorithms performance on benchmarks.
* Evaluate the effectiveness of the algorithms with coverage, hypervolume (HV), Trailing Suction Hopper Dredger (TSHD) metrics.
* Investigate the impact of various parameter settings on the performance of the algorithms.
* Implement the superposition of NSGAII and SMPSO algorithms and analyze the results in comparison with individual implementation of the algorithms
  1. Contribution

This dissertation project advances the field of multi-objective optimization by

* Proposing a framework to combine the NSGA-II and SMPSO algorithms.
* Evaluating how well this integrated algorithm performs on standard test problems.
* Offering insights into how genetic algorithms (GA) and particle swarm optimization (PSO) techniques work together in multi-objective optimization.
  1. Scope and limitations
* The scope of this project is limited to the integration of NSGA-II and SMPSO algorithms and the superposition of both algorithms for multi-objective optimization.
* The evaluation of performance is focused on benchmark problems and may not cover all possible scenarios.
* The study assumes fixed parameter settings and does not explore dynamic parameter adaptation strategies.

# Paper structure

* 1. Used technologies

The technologies that were used in the implementation of this project were the following:

* Sourcetree – version control
* IntelliJ IDEA – software tool
* Java language – backend implementation
* JDK
* jMetal library – library used for working with evolutionary algorithms
* Excel – tool used for generating the graphics
* Draw.io – tool used for creating diagrams

# Chapter 2. Technology stack description

This chapter provides detailed information regarding the important tools used in this project and how they helped in implementing the scope of the project.

## 2.1. IntelliJ IDEA

How does it work? IntelliJ IDEA does not only like a code instrument, but it is an entire platform that is meant to optimize all development processes. It provides a smooth and complete environment for numerous programming languages including Java, Kotlin, Python, and others with its intuitive interface and full suite of kit.

A good thing about it is its great debugging function that allows programmers detect and eliminate bugs in an easy and quick way. The integrated debugger provides the real-time insights on the code behaviour which helps the developers detect faults and amend them with the accuracy. Actually, IntelliJ IDEA's powerful analysis tools give a breakthrough analysis into code, performance and code quality as well as potential vulnerabilities hence developers can be proactive in code enhancement. It also enables the seamless collaborations by integrating with popular collaboration tools such as Git, GitHub, and Bitbucket hence making the version control and team collaboration possible.

In fact, IntelliJ IDEA is no longer just an IDE; It is a powerful ally that spotlights creative potential and turns it into outstanding software solutions.

Installing IntelliJ IDEA is a straightforward process:

1. Download IntelliJ IDEA: Visit the official JetBrains website and download the needed (Windows, macOS, or Linux).
2. Run the Installer.
3. Select Installation Options: During the installation process, you may be prompted to select installation options such as the installation location and creating desktop shortcuts. Customize these options according to your preferences.
4. Complete the Installation: After configuring the installation options, proceed with the installation process. Once completed, you'll have IntelliJ IDEA installed on your system.

Running a project in IntelliJ IDEA:

1. Open IntelliJ IDEA: Launch IntelliJ IDEA by double-clicking on the desktop shortcut or selecting it from the installed applications list.
2. Open or Create a Project: If you already have a project, you can open it by selecting "File" > "Open" and navigating to the project directory. If you're starting a new project, select "File" > "New" > "Project" and follow the prompts to create a new project.
3. Configure Project Settings (if necessary): Depending on the type of project you're working on, you may need to configure project settings such as SDKs, libraries, and frameworks. IntelliJ IDEA provides wizards and tools to assist with this configuration.
4. Run the Project: Once your project is open, navigate to the main class or file containing the entry point of your application. Right-click on the file and select "Run <Your\_Project\_Name>".
5. View Output: IntelliJ IDEA will compile and run your project. You can view the output, including any errors or log messages, in the "Run" or "Console" tab at the bottom of the IntelliJ IDEA window.

## 2.2. Sourcetree

SourceTree is a good open source Git client with a good user interface and it offers users a smooth experience while managing Git repositories. SourceTree, which is the invention of Atlassian, is designed to deal with the complexities of version control by use of a visual interface through which users effortlessly bind to their Git operations such as commit, push, pull and merge. Its easy to use, together with other properties such as visual diffing and branching, make both beginner and experienced developers able to sail through their projects with ease. And it can be used with either git or Mercurial repositories. Whether you are a solo worker or part of a group, SourceTree will simplify version control and make it easy to get the best out of it for all individuals.

Every time a user makes certain changes in the code, he/she has the opportunity to do the commit operations, then push to the branch on which he/she is working on a certain task in the project. When the user wants to merge all the code on his branch with the source code, he makes a pull request.

Creating pull requests is often used in team software project development. Team members reviewing the code in the pull request have the opportunity to see the differences between the previous code and the newly written code that is to be changed and added to the source code. The pull request process, followed by the review, ends with the merge of the code to the source code located on the main branch, usually called master or main.

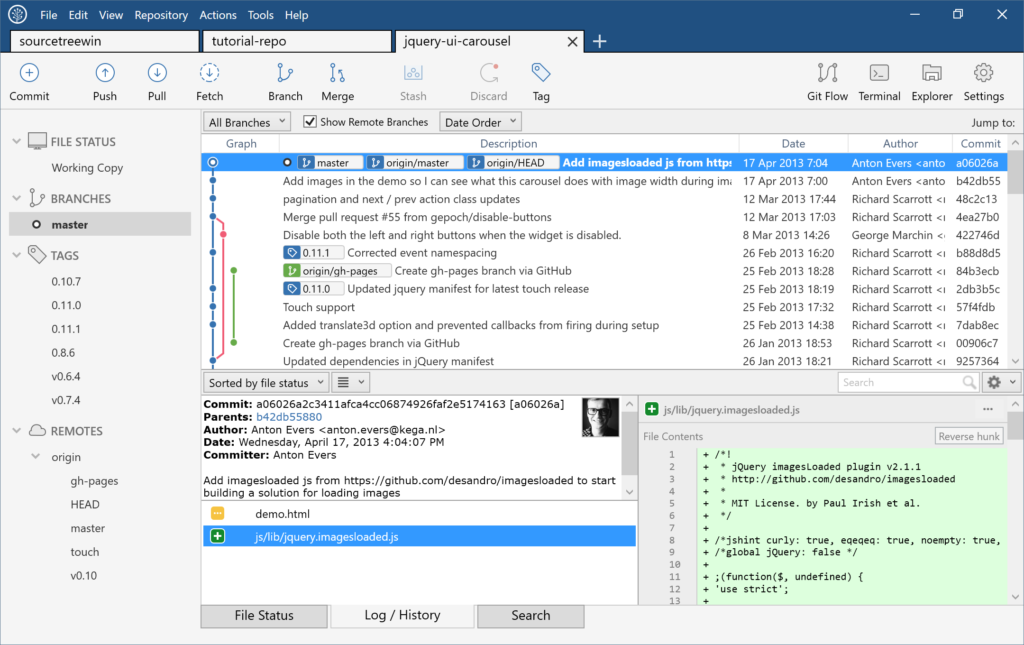


Fig. 1 Sourcetree interface []

## 2.3. Java language

Java is the language which is valued for its ease and convenience of using, and also for its stability, reliability and robustness. Java, invented by Sun Microsystems in the mid of the 1990s, has nowadays grown into one of the most used languages which are able to develop different type of apps, such as enterprise-level software, mobile apps and web apps. Java independence from the hardware types is provided by the Java Virtual Machine (JVM), and as a result the same code can run on any device as long as it has a JVM. This makes Java to be the ideal choice for the cross-platform development. The strength of Java as a language lies in its engineering through object-oriented programming principles which makes it very suitable for developing modular and scalable code base to be used in various where have been set up records.

The language syntax is similar to that of C, C++, C#, using:

* semicolon to mark the end of a line of code
* braces to group multiple lines of code that form a method, class, or namespace
* square brackets to define a vector or matrix

When we create a Java application, for example a console type, the Main.java class is created automatically and has the following appearance:

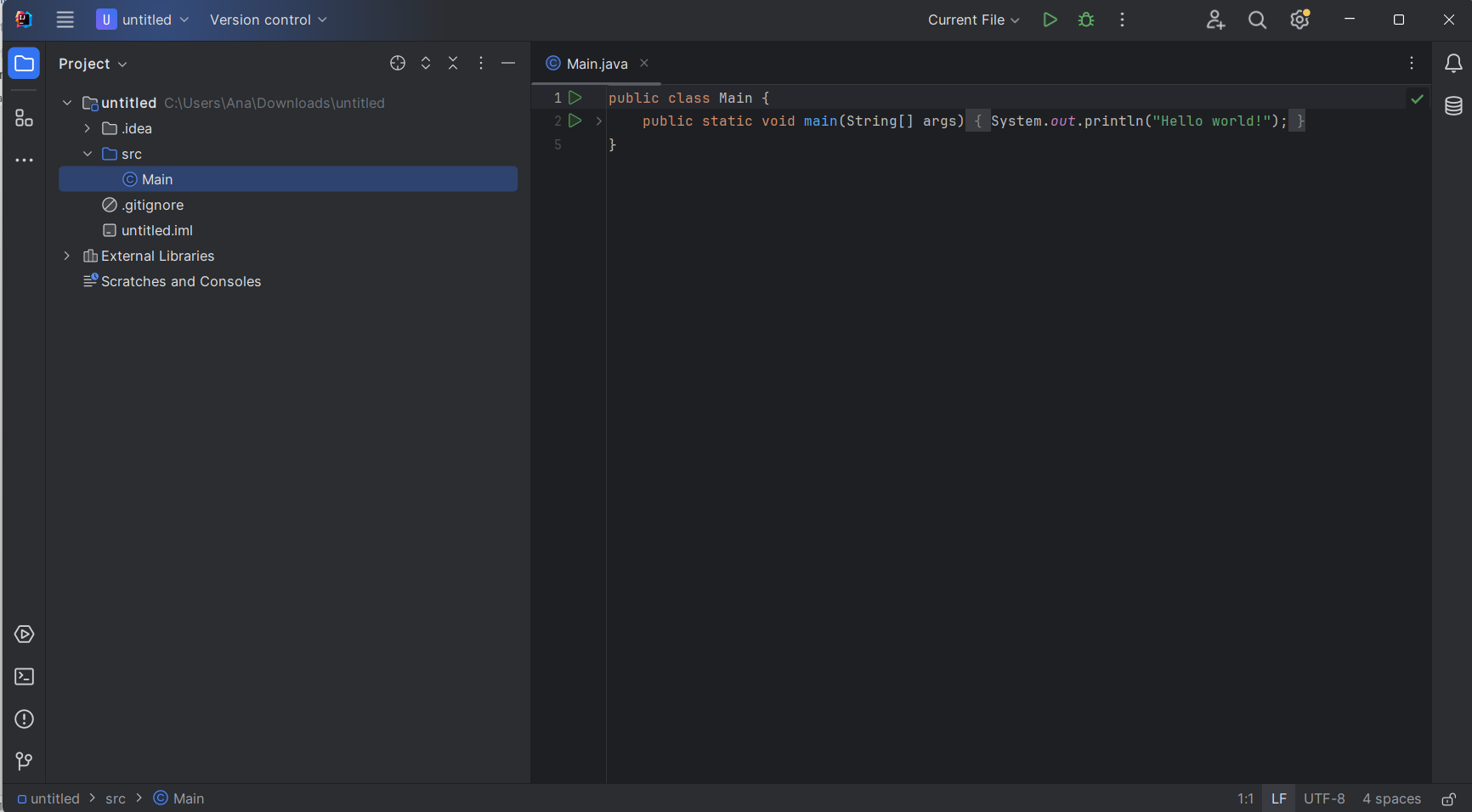


Fig. 2 Java Main class

* *public class HelloWorld {:* The next line states the helloworld class. The Java style is determined by the very first class definition, and the file name should also match the class name. The keyword public thus implies that the class is available for use from other classes.
* *public static void main(String[] args) {:* The first sentence tells that the method being executed is a main method which as primary entry of the Java program. It is a unique approach that the JVM emergency procedures for launching the program execution. "public" indicates that it is possible for a method to be accessible outside the class during runtime. "static" is the term used to refer to the fact that the particular method is attributed to the class as a whole, whereas "instance" means that these are methods of the class's instances. there is no return value stated in the signature of this method. The function name is main, and the String[] args is the parameter to main method, it is an array string of object that can hold command-line arguments.
* *System.out.println("Hello, World!");*: This line prints the following string "*Hello, World!*" to the console. System is a predefined class in the Java library. out is a static member of the System class, which represents the standard output stream. println is a method of the PrintStream class (to which out refers), used to print a line of text. "*Hello, World!*" is the string that will be printed to the console. The semicolon ; terminates the statement, indicating the end of the line of code.

### 2.3.1. Comparison with other programming languages

The list with the most globally used programming languages in 2023 are the following:

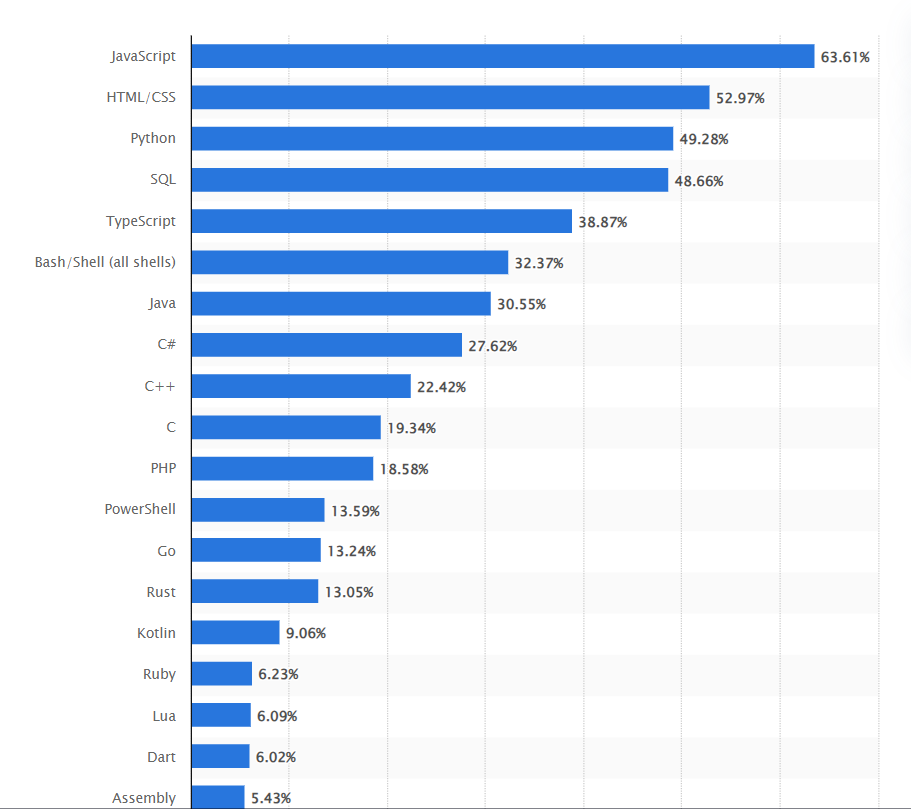


Fig. 3 The most globally used programming languages in 2023 []

Java is often compared with programming languages like Javascript, C++ and C#. Some comparison between these programming languages are described below:

* Java vs. C++:
  + Memory Management: Java has automatic memory management (garbage collection), while C++ requires manual memory allocation and deallocation.
  + Performance: C++ offers better performance than Java, as it is closer to the hardware and allows for more control over memory and CPU usage.
  + Language Complexity: C++ is considered more complex than Java, with features like pointers, operator overloading, and multiple inheritance, whereas Java emphasizes simplicity and safety.
* Java vs. C#:
  + Platform: Java is designed to be platform-independent through its "write once, run anywhere" approach with the Java Virtual Machine (JVM). On the other hand, C# primarily targets the Windows platform, although it's possible to use it with other platforms through Mono or .NET Core.
  + Syntax: Both languages have similar syntax, owing to their shared heritage from C and C++. However, C# tends to have more modern language features, such as LINQ (Language Integrated Query) and properties, which can lead to more concise code compared to Java.
  + Memory Management: Both languages use automatic memory management (garbage collection), which simplifies memory management for developers.
  + Concurrency: Both languages provide support for multithreading and concurrency. Java offers features like synchronized blocks and the java.util.concurrent package, while C# has built-in support for asynchronous programming with the async/await keywords.

## 2.4. jMetal library

jMetal is a framework based on Java language which is created for multi-objective optimization (MOO). It includes metaheuristic algorithms and provides a wide range of tools and features to solve optimization problems with conflicting objectives. Some details about jMetal, including why it's used, its performance, and aspects related to its development can be described as the following:

* Multi-objective optimization support: The major claim for jMetal toolkit is its competence to solve problems with more than one conflicting objectives, which is very common in a real world.
* Extensive algorithm library: It brings together the different species of metaheuristic algorithms like NSGA-II, NSGA-III, MOEA/D, SPEA2 and so on
* Flexibility and extensibility: the jMetal architecture possesses high flexibility and extendibility that makes it easy to interject the new problems, algorithms, operators, and other components in the system.

* Open-source and community-driven: Like being open source, the jMetal is the platform for the two-way communication and contributions from the optimization research community for the energetic research progress.

* Comprehensive documentation and examples: It has a well designed documentation, tutorial, and samples that could be very useful during learning and applied usage of the framework.

* Scalability: jMetal is a form of a software which could transfer those optimization problems of varying degree of complexities as well as boundaries. Hence, its ability to handle both small and large scale applications is its many key features.

* Efficiency: The performance of some algorithms may differ due to the fact that the tasks are different in their nature, and jMetal essentially tries to provide timesaving and effective implementations of generic algorithms for multiple objective problems.
* Parallelization support: This is because some jMetal algorithms actually give some input support that is needed for users to profit from parallel computing resource so as to provide faster convergence and improved performances.

* Continuous improvement: As the jMetal community is focused on the performance and efficiency enhancements, each release includes the featured optimization and modifications to be implemented as follows.

* Modularity and abstraction: jMetal introduces a modular architecture, and it provides a clear structure that allows developers to understand the metaheuristic search components, allowing easy extension and maintenance.

* Compatibility: It works with the multiple Java development platforms of IntelliJ IDEA, Eclipse, and others, so, the developers from different levels have an equal chance to learn it.

The current project structure is the following:

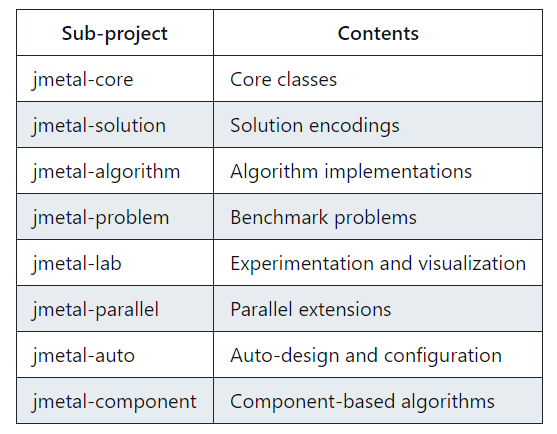


Fig. 4 Project development structure []

### 2.4.1. How to install and use it?

jMetal can be free downloaded from the official website [jMetal-github](https://github.com/jMetal/jMetal). For adding it to the project:

* Create a folder named "main" within the "src" folder of your project.
* Inside the "main" folder, create a subfolder named "java".
* Move the downloaded JAR files into the newly created "java" folder.

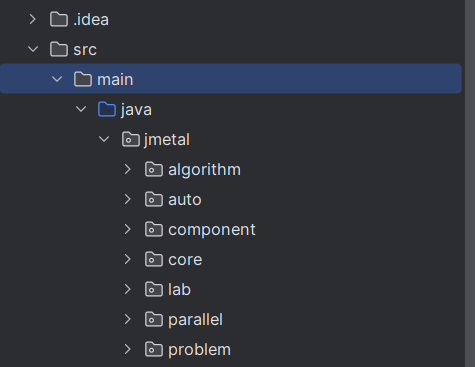


Fig. 5 jMetal library structure

Using the jMetal library in the project is easy as it only requires to import the needed files when a new class is created and extends another class from the library to use its facilities.

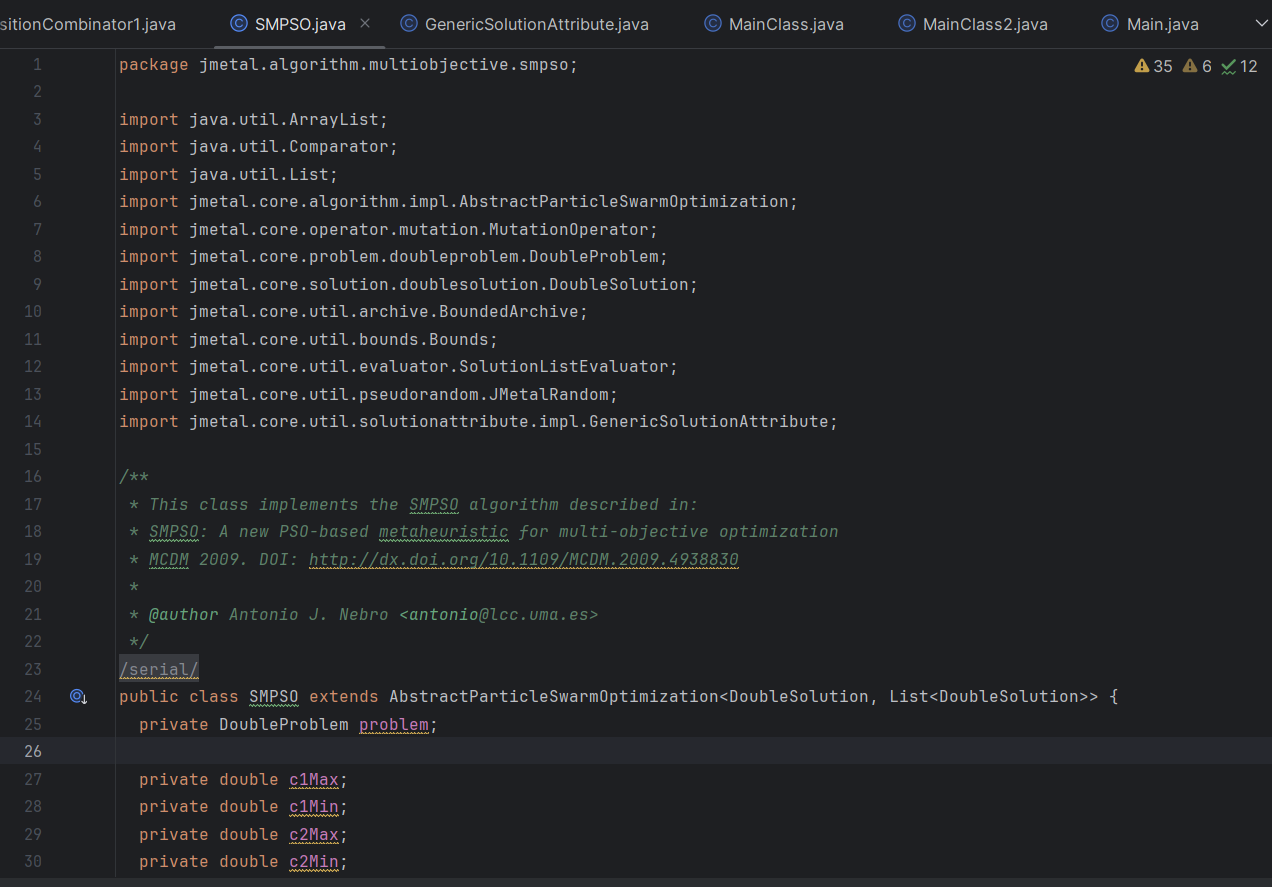


Fig. 6 SMPSO algorithm class

To run the developed algorithms, a MainClass has to be implemented. Here, we can run the implementation and generate the needed results.

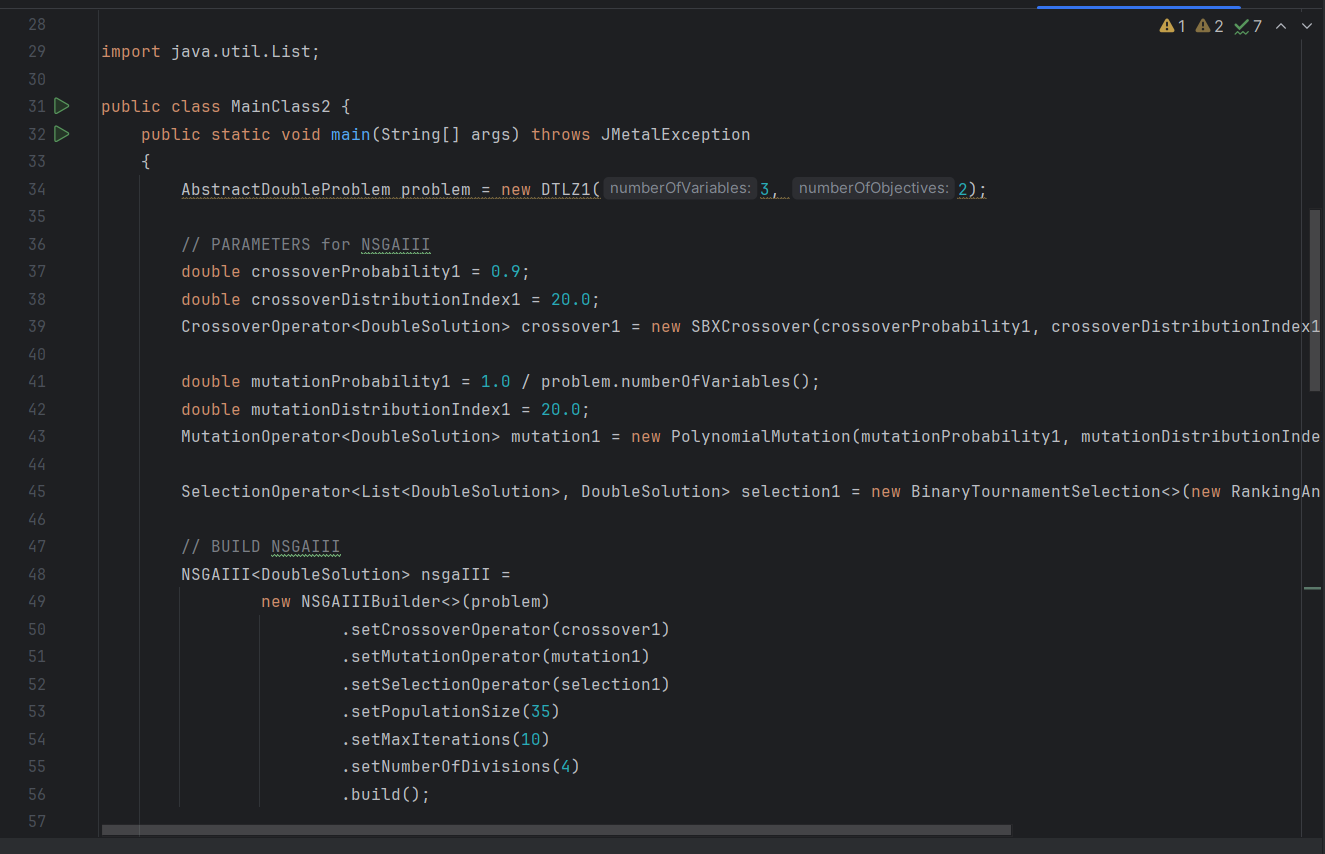


Fig. 7 MainClass.java

For the purpose of this dissertation project, the main files that are needed for the implementation are the following:

* NSGAII algorithm, which is placed in the multiobjective directory of jMetal library
* SMPSO algorithm, which is also placed in the multiobjective directory of jMetal library
* AbstractEvolutionaryAlgorithm class
* Algorithm class
* TODO: to be added after the implementation

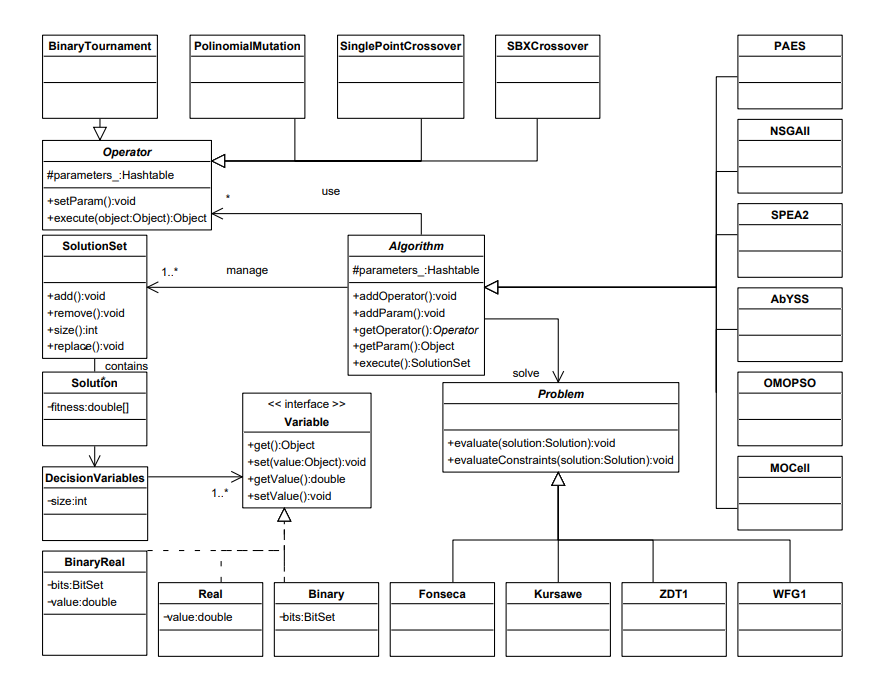


Fig. 8 UML class diagram of jMetal

### 2.4.2. Algorithm types

jMetal library contain information and development regarding many algorithms which are divided into the following categories:

* Multi-objective algorithms
* Single-objective algorithms

These two categories of algorithms are also divided into other algorithms types, for example:

* Evolutionary algorithm
* Genetic algorithm
* Swarm-based algorithm
* Parallel algorithm
* Differential evolution algorithm

The following two images display the algorithms and categories find in the jMetal library:

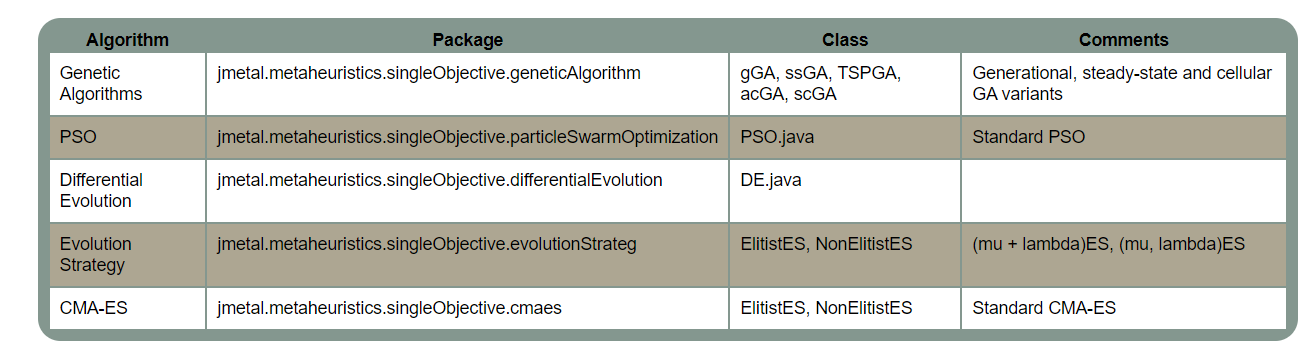


Fig. 9 Single-objective algorithms []

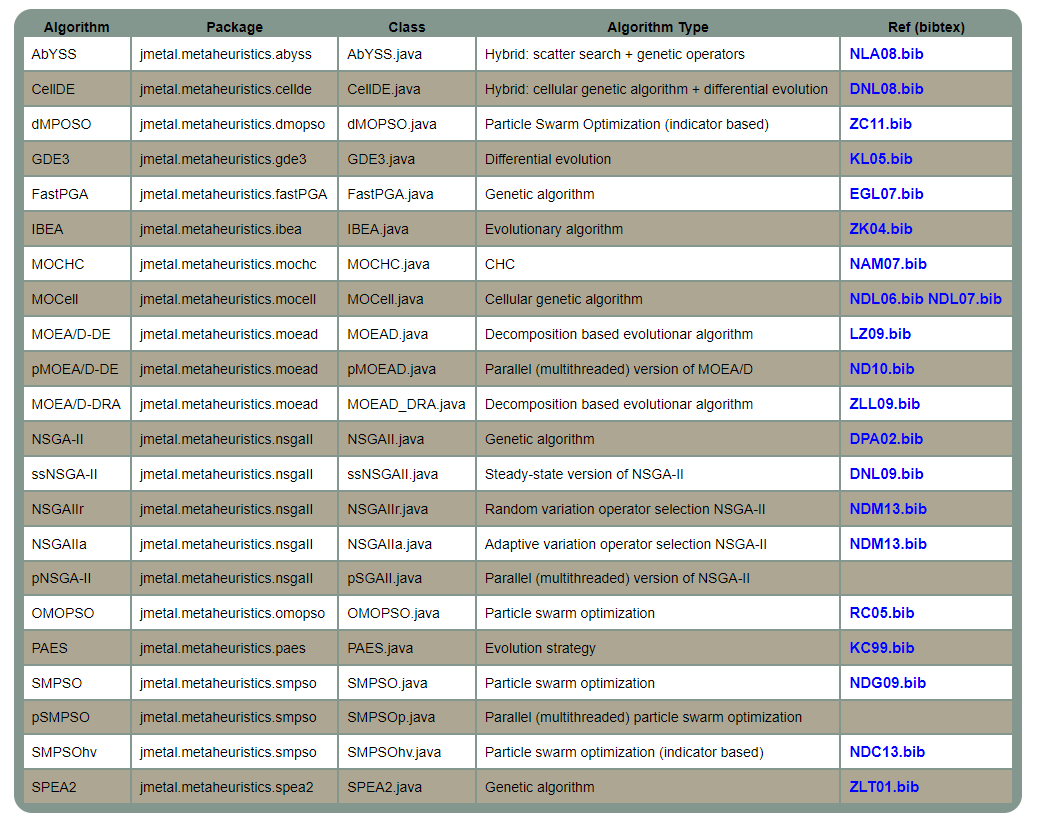


Fig. 10 Multi-objective algorithms []

In the context of this paper work, NSGAII and SMPSO algorithms were used to develop the scope of this project.

### 2.4.3. Quality Indicators

jMetal provides a suite of quality indicators that are essential for comparing the performance of algorithms use solve multi-objective optimization. These indicators help evaluating various aspects of algorithm generated solution sets, such as convergence, diversity, and overall solution quality relative to Pareto front.

The following metrics (quality indicators) are used in this version of jMetal:

* Generational distance: illustrates how close the algorithm's solution is to the actual optimal solutions.
* Inverted generational distance: Similar to GD, but it also considers the diversity of the solutions by measuring the distance from the reference Pareto front to the solutions. Lower IGD values indicate better convergence and diversity.
* Hypervolume: measures the space covered by the solutions found by the algorithm. A higher hypervolume value means the algorithm's solutions are closer to the ideal set of optimal solutions.
  + PISA Hypervolume: Provides an accuracy-efficiency trade-off, and it is useful when we have only two or three objectives. However, its performance decreases with a higher-dimensional objective space.
  + WFGH Hypervolume: Simultaneously tackles problems of many objectives and is preferred for its effectiveness and the capability to handle objective spaces which are big without problematic loss in performance. Normalized hypervolume: A variant of the hypervolume indicator that is normalized based on a reference point, making it easier to compare across different problem instances.
* Epsilon: measures how much the set of optimal solutions needs to be adjusted to match the reference optimal set. Lower values indicate better performance.
* Generalized spread: evaluates how evenly and widely the solutions are spread across the entire range of objectives on the Pareto front.
* Error ratio: The error rate measures how many solutions the algorithm identified as optimal (Pareto front) are actually not part of the true optimal set. An error rate of 0 means all identified solutions are truly optimal, which is the best outcome. An error rate of 1 means none of the identified solutions are truly optimal, indicating poor performance.
* Set coverage: compares two sets of solutions by calculating the percentage of solutions in one set that are either dominated by or equal to solutions in the other set. This helps determine how one set of solutions performs relative to the other.

### 2.4.4. Problems

Problems in jMetal represent the optimization challenges that algorithms are designed to solve. They are defined by their decision variables, objective functions, and constraints, providing a structured way to evaluate the performance of different optimization strategies.

Types of Problems

Single-objective Problems: These focus on optimizing one specific goal. The aim is to either maximize or minimize this single objective.

Multi-objective Problems: These involve optimizing two or more conflicting goals at the same time. The aim is to find a set of solutions that provide the best balance between the objectives, known as the Pareto front.

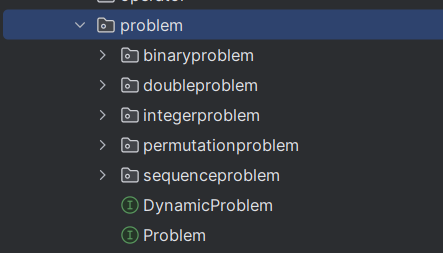


Fig. 11 jMetal problem structure

Examples of problems:

jMetal offers a variety of benchmark problems that the optimization community uses to test and compare different algorithms. Examples include:

* ZDT (Zitzler, Deb, and Thiele) Series: Represents a set of synthetic problems with varying characteristics, such as the form of Pareto front and and how solutions are distributed.
* DTLZ (Deb, Thiele, Laumanns, and Zitzler) Series: Another set of synthetic problems designed for testing algorithms on many-objective optimization tasks.

All these problems implement the Problem interface which has the following structure:

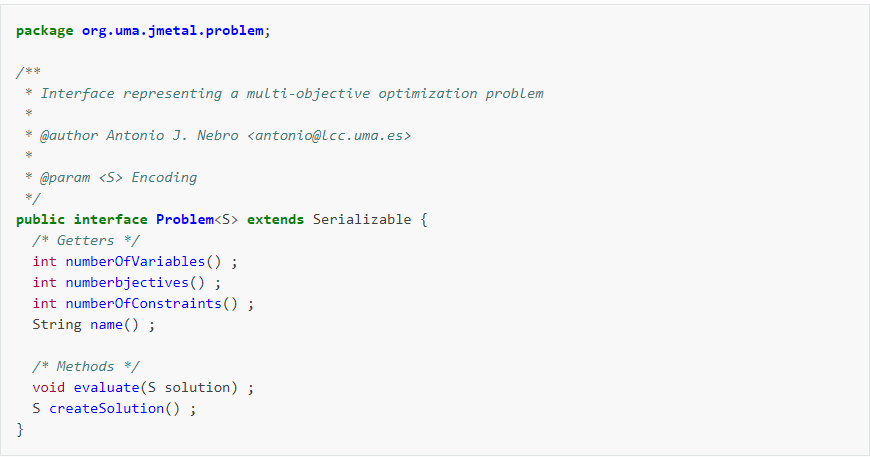


Fig. 12 Problem interface in jMetal []

# Chapter 3. Theoretical part of the implementation

## 3.1. Pareto concept

The Pareto concept, named after economist Vilfredo Pareto, is fundamental to multi-objective optimization and widely used in economics, engineering, and decision-making. It applies when improving one aspect of a solution cannot be done without worsening another aspect. The efficiency of this method is found in the following concepts:

* Pareto optimality: In a multi-objective optimization problem, a solution is Pareto optimal if no other solution can improve any objective without worsening at least one other objective. These solutions are considered efficient or non-dominated.
* Pareto dominance:

A solution X can dominate another solution Y if X is at least as good as Y in all objectives and better in at least one. In a minimization problem, for any two solutions X and Y:

* + If f1(X) ≤ f1(Y) and f2(X) < f2(Y), then X dominates Y.
  + If f1(X) < f1(Y) and f2(X) ≤ f2(Y), then X dominates Y.
  + If neither X dominates Y nor Y dominates X, then they are non-dominated.
* Pareto front: (or Pareto boundary) represents the set of all optimal solutions of Pareto. It shows the trade-offs between different conflicting objectives. Choosing any point on the Pareto front means that improving one objective will inevitably worsen at least one other objective.

Defining the Pareto front through the following graphic:

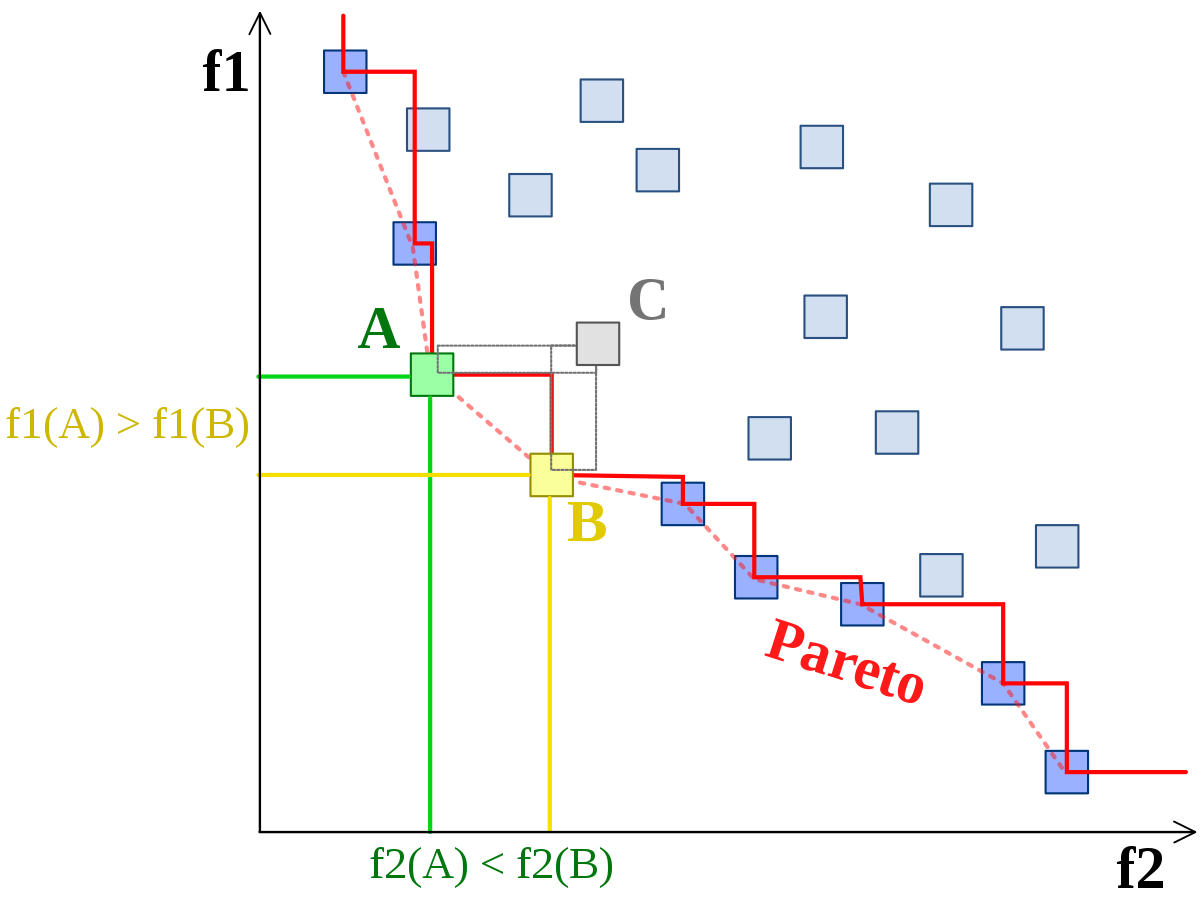


Fig. 13 Pareto front

### 3.1.1. Multi-objective genetic algorithms

The genetic algorithm begins with an initial population of individuals, often generated randomly, and runs until a stopping condition is met (like finding a satisfactory solution or hitting a time limit). After evaluating the initial population, the algorithm enters its main loop. If the stopping condition hasn't been met, the selection phase starts, where certain individuals are chosen to be parents. These parents produce offspring through crossover (recombination) and/or mutation. The offspring make up the new population, which is then evaluated. Depending on the algorithm, the next parent population may consist of these offspring, or a combination of offspring and previous parents (e.g., through elitism).

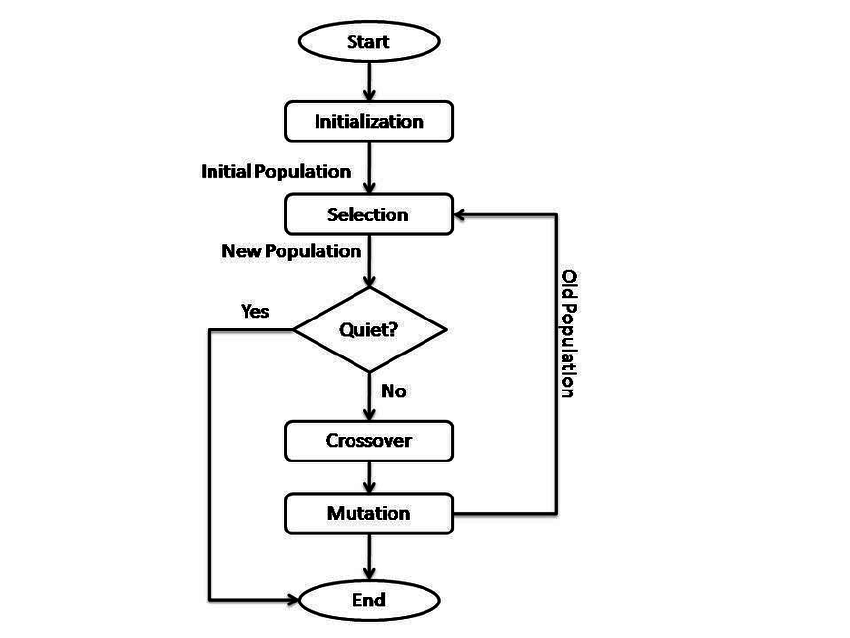


Fig. 14 Genetic algorithm []

### 3.1.1.1. Non-dominated Sorting Genetic Algorithm

The Non-dominated Sorting Genetic Algorithm (NSGA) is a multi-objective optimization algorithm based on natural selection and evolution. It is designed to tackle problems with multiple conflicting goals and is widely used in engineering, economics, and other areas where decisions must balance various criteria.

At its core, NSGA keeps a population represented by candidate solutions, called individuals or chromosomes, each representing a potential solution to the optimization problem. Unlike traditional single-objective genetic algorithms that evaluate solutions based on one objective, NSGA evaluates them based on multiple objectives at the same time. This helps NSGA find a set of solutions that are not outperformed by any other solution across all objectives, known as the Pareto-optimal front.

The key innovation of NSGA is its unique method of selection and reproduction. Traditional genetic algorithms use fitness proportionate selection, where individuals are chosen for reproduction based on their relative fitness. NSGA, on the other hand, uses non-dominated sorting to rank individuals according to how they compare to others in the population. This process sorts the population into several groups or "fronts." The first front consists of non-dominated solutions, which are not outperformed by any other solutions in the population.

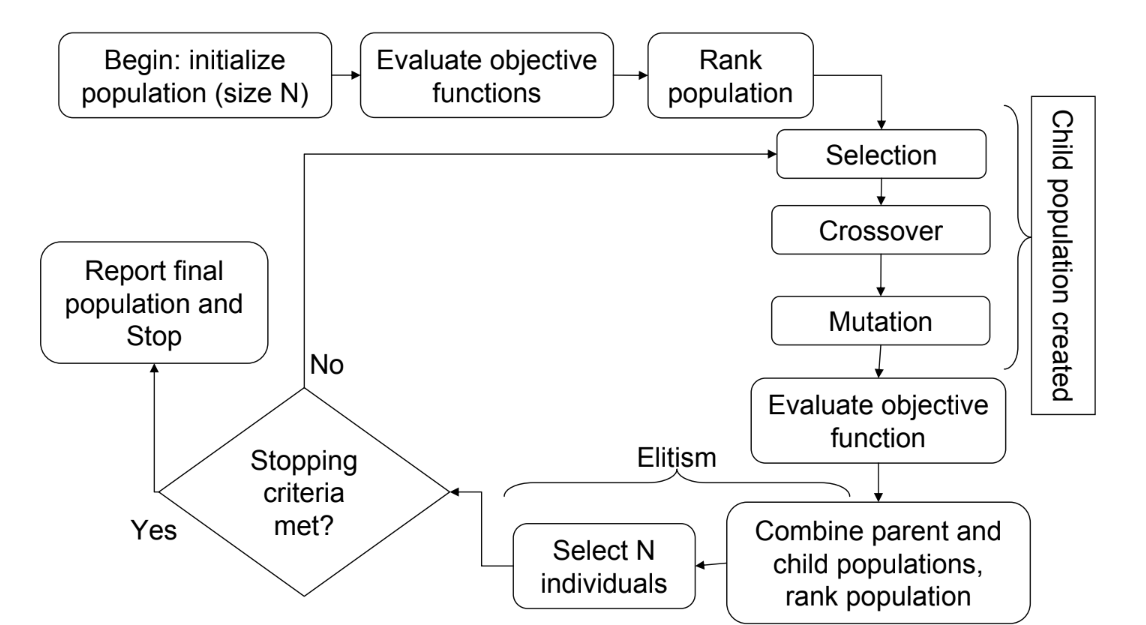
After sorting, NSGA uses a mix of elitist selection and diversity preservation to create the next generation of individuals. Elitist selection ensures that the best solutions from the current population are carried over to the next generation. Meanwhile, diversity preservation methods, like crowding distance, encourage exploring the solution space and prevent the algorithm from getting stuck in suboptimal solutions.

In the reproduction phase of NSGA, offspring are generated by applying genetic operators like crossover and mutation to selected parent individuals. These operators help explore and utilize the solution space, enabling NSGA to produce a variety of high-quality solutions over successive generations.

### 3.1.1.2. Non-dominated Sorting Genetic Algorithm II

NSGA-II (Nondominated Sorting Genetic Algorithm II) is a type of genetic algorithm within the evolutionary algorithms family, designed specifically for solving multi-objective optimization problems. It is an improved version of the original NSGA (Nondominated Sorting Genetic Algorithm), with enhancements to boost its performance and convergence.

NSGA-II works on principles inspired by natural selection and genetics. It keeps a population of potential solutions, often called individuals or chromosomes, and evolves this population over multiple generations. The goal is to find solutions that form the Pareto-optimal front, where no solution can be improved in one objective without compromising another.

Fig. 15 NSGA II algorithm process []

The main components of the algorithm which make the difference between NSGA and its own are the following:

Step 1: Calculation of Dominance Counter

For every individual in the population, a dominance counter is calculated. This counter shows how many other individuals in the population outperform the current individual.

Step 2: Identification of Pareto Fronts

After calculating the dominance counters in Step 1, all individuals in the first Pareto front (Front I) will have a dominance counter of 0, as no other individual in the population outperforms them. For each individual in Front I, the algorithm finds the individuals they dominate and decreases their dominance counters by 1. This process identifies who is dominated by those in Front I and adjusts their counters accordingly. Once this is done, the individuals with a dominance counter of 0 will form the second Pareto front (Front II).

Iterative Process for All Fronts

This process of identifying Pareto fronts is repeated for each subsequent front. For each front, the algorithm finds individuals with a dominance counter of 0, and these individuals form the next Pareto front. The dominance counters of individuals dominated by those in the current front are decreased. This continues until every individual in the population has been assigned to a Pareto front.

Termination of the Process

Termination process continues until there are no individuals left in the population to be assigned to the Pareto front. At this point, the entire population has been divided into fronts based on their Pareto dominance relationships. This marks the end of the process, and the algorithm proceeds to the next stage, typically involving the selection and reproduction of individuals to generate the next generation of solutions.

NSGA-II improves on the original NSGA by addressing issues like high computational complexity and the tendency to produce too many solutions in crowded areas of the Pareto front. It offers better convergence and maintains diversity, making it a popular choice for solving multi-objective optimization problems in various fields.

This algorithm has a computational complexity of O(MN^2), where M stands for the number of objectives and N signifies the number of individuals in the population. Although this modification reduces the computational complexity, it increases the memory usage from O(N) in NSGA to O(N^2) in NSGA-II.

## 3.2. Particle Swarm Optimization algorithms

The Particle Swarm Optimization (PSO) algorithms represent nature-inspired optimization techniques that are inspired by the behavior of organisms, particularly birds or fish schooling.

In this algorithm, a set of potential solutions, known as particles, moves through the search space by adjusting their positions based on their own experiences and the experiences of their neighbors. Each particle's movement is influenced by two main factors:

* its personal best position (the best solution it has discovered so far) and
* the global best position (the best solution found by any particle in the group).

By continuously updating their positions, particles explore the search space and gradually move towards optimal solutions. PSO algorithms are popular because they are simple, efficient, and capable of handling complex, high-dimensional problems. They have been successfully used in areas like engineering, finance, and machine learning, where they are excellent at finding nearly optimal solutions for difficult optimization challenges.

### 3.2.1. Particle Swarm Optimization algorithm description

Particle Swarm Optimization (PSO) is an optimization algorithm inspired by the social behaviors of animals, like birds flocking or fish schooling. It uses a group approach to find the best solutions.Developed by Kennedy and Eberhart in 1995, PSO has gained popularity due to its simplicity, efficiency, and effectiveness.

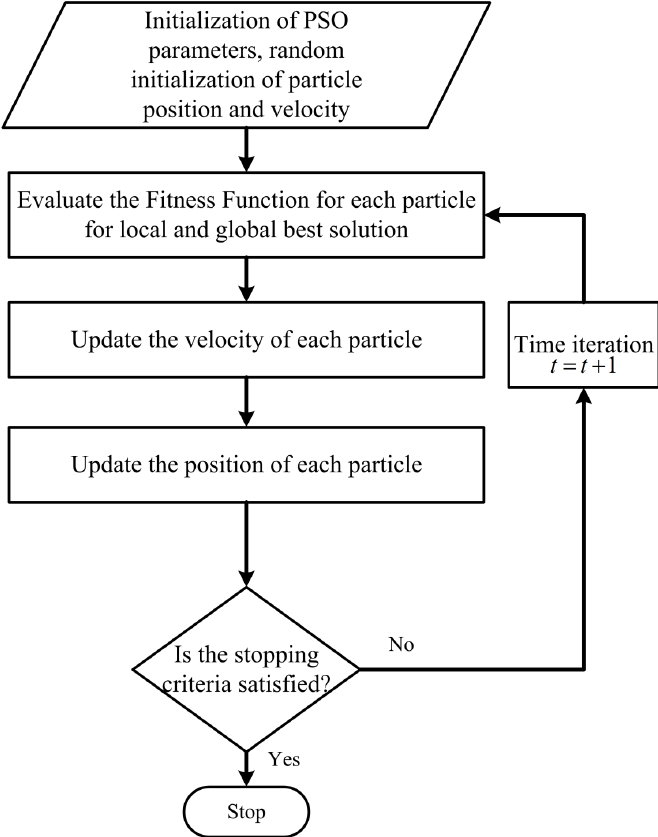


Fig. 16 PSO algorithm process []

Initialization

The algorithm begins by creating a group of particles, each one representing a possible solution to the problem. These particles are spread out randomly across the search space, and each particle is given a random direction and speed.

Fitness Evaluation

After initialization, each particle's fitness is assessed using an objective function that measures the quality of the solution it represents. This objective function is specific to the problem and reflects the optimization goals.

Initialization of Particle's Memory

Each particle keeps track of two key positions: its personal best position (Pbest) and the overall best position (Gbest). At the start, both of these positions are set to where the particle is currently located.

Particle Movement

In each cycle of the algorithm, particles update their positions and speeds based on their current position, their personal best position, and the global best position. The movement of each particle is influenced by two main factors:

* Cognitive Component: This component directs the particle towards its personal best position, encouraging exploitation of promising regions of the search space.
* Social Component: This component guides the particle towards the global best position found by any particle in the swarm, facilitating exploration of the entire search space.

The new velocity of each particle is calculated using a combination of its current velocity, cognitive component, and social component. The particle then updates its position based on the new velocity, potentially moving towards more promising regions of the search space.

Updating Particle's Memory

After moving, each particle checks if its new position is better than its previous personal best (Pbest). If it is, the particle updates its Pbest to this new position. Additionally, each particle's Pbest is compared to the global best position (Gbest) found by any particle in the group. If a particle's Pbest is better than the current Gbest, the Gbest is updated to this new, better position.

Termination Criterion

The PSO algorithm keeps running through cycles until a stopping condition is met. Common reasons to stop include reaching a set number of iterations, finding a good enough solution, or hitting a predefined computational limit.

### 3.2.2. Strength Pareto Multi-objective Particle Swarm Optimization algorithm

The SMPSO algorithm is an improved version of the classic Particle Swarm Optimization (PSO) algorithm, specifically designed for solving multi-objective optimization problems. Introduced by Durillo and Nebro in 2009, SMPSO enhances the traditional PSO by including Pareto dominance and methods to maintain diversity among solutions.

In SMPSO, particles move through the search space by adjusting their positions based on their personal best and the global best positions, while also taking into account Pareto dominance among solutions. By keeping a diverse set of non-dominated solutions and promoting exploration of the entire Pareto front, SMPSO efficiently finds a variety of trade-off solutions that represent the best balance between conflicting objectives.

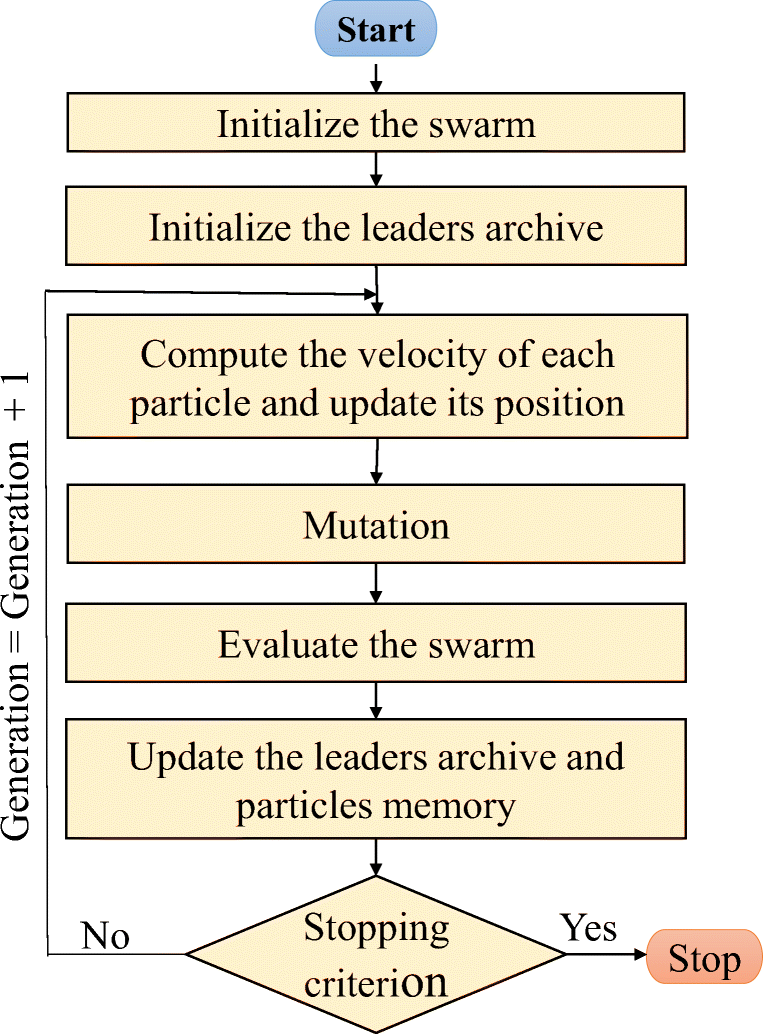


Fig. 17 SMPSO algorithm process <https://www.researchgate.net/figure/Followchart-of-the-SMPSO-algorithm_fig3_349661785>

The comparison between PSO and SMPSO in the algorithm process are the following:

Initialization:

Like PSO, SMPSO starts by randomly initializing a population of particles within the search space. Each particle represents a possible solution to the multi-objective optimization problem.

Fitness Evaluation:

Unlike PSO, which evaluates fitness based on a single objective, SMPSO assesses each particle's fitness using multiple objective functions at the same time. This enables SMPSO to tackle multi-objective optimization problems, optimizing solutions with respect to conflicting goals.

Pareto Dominance:

In SMPSO, particles keep a set of personal best solutions (Pbest) and a global best solution (Gbest). Unlike PSO, which compares solutions using a single fitness value, SMPSO uses Pareto dominance to evaluate solutions. SMPSO ensures that the population only includes non-dominated solutions, forming the Pareto front, by maintaining these dominance relationships.

Velocity and Position Update:

In SMPSO, the equations for updating velocity and position are similar to those in PSO but are modified to include Pareto dominance and diversity preservation mechanisms. Particles adjust their velocities based on their personal best and global best solutions, as well as their interactions with other particles in the swarm. However, in SMPSO, the velocity update also takes into account Pareto dominance relationships among solutions to steer the search towards non-dominated areas of the objective space.

Diversity Preservation:

A major difference between SMPSO and classic PSO is the inclusion of diversity preservation mechanisms in SMPSO. These mechanisms encourage the exploration of the entire Pareto front by promoting solution diversity. Methods like crowding distance or grid-based diversity measures are used to maintain a varied set of non-dominated solutions in the population.

Termination Criterion:

The termination criteria for SMPSO are usually similar to those for PSO. The algorithm stops running when a predefined condition is met, such as reaching a maximum number of iterations, finding a satisfactory set of non-dominated solutions, or exceeding a computational limit.

Compared to classic PSO, SMPSO offers several advantages for solving multi-objective optimization problems.

Firstly, SMPSO handles problems with multiple conflicting objectives more effectively by keeping a diverse set of non-dominated solutions along the Pareto front. Additionally, SMPSO promotes thorough exploration of the entire Pareto front through diversity preservation mechanisms, whereas PSO may have difficulty adequately exploring the objective space in multi-objective scenarios.

As a conclusion, SMPSO is better suited for multi-objective optimization tasks, offering improved convergence towards Pareto-optimal solutions and better exploration of the solution space compared to classic PSO.

## 3.3. Algorithm’s comparisons

Evolutionary algorithms for multi-objective optimization generate sets of solutions that often cannot be easily ordered to determine the best one, so comparing the performance of exploration algorithms becomes a complex task. To enhance objectivity in the comparison process, it is necessary that experiments involving heuristic algorithms adhere to standardized experimental procedures. These procedures typically involve the following steps:

* defining the objectives or aims of the experiment.
* selecting appropriate performance measures, referred to as metrics, to evaluate the algorithms' performance.
* designing and executing the experiment according to established protocols.
* analyzing the collected data and drawing meaningful conclusions based on the results.
* publishing the outcomes and conclusions of the experiment to contribute to the collective understanding of algorithm performance.

While delving into the theoretical aspects and providing specific examples of experimental procedures exceeds the scope of this work, it is essential to outline the metrics used for evaluating the algorithms. These metrics serve as quantitative indicators to assess various aspects of algorithm performance and effectiveness. These metrics are also described above in the chapter [2.4.3. Quality Indicators](#_2.4.3._Quality_Indicators).

For the purpose of this project, the following metrics were developed:

* Coverage of two sets:

This metric compares the performances of two tested algorithms by determining the proportion of individuals from the Pareto front determined by one algorithm that are dominated by individuals from the Pareto front determined by the other algorithm.

A = the set of individuals from the Pareto front of the first algorithm, and

B = the set of individuals from the Pareto front of the second algorithm.

Let ai denote the individual i from set A, and bj denote the individual j from set B. We define the coverage function, C(A, B) as follows:

*C*(*A*,*B*)=∣*B*∣∑*bj*∈*B*​*ai*​∈*A* where that *ai*​ dominates *bj*​​

From the formula, it is evident that the metric takes values in the interval [0, 1]. For example, if

C(A, B) = 0.5, it shows that a half of individuals from set B are dominated by individuals from set A. However, for accuracy, it is necessary to calculate C(B, A) as well, because C(A, B) = 1 - C(B, A).

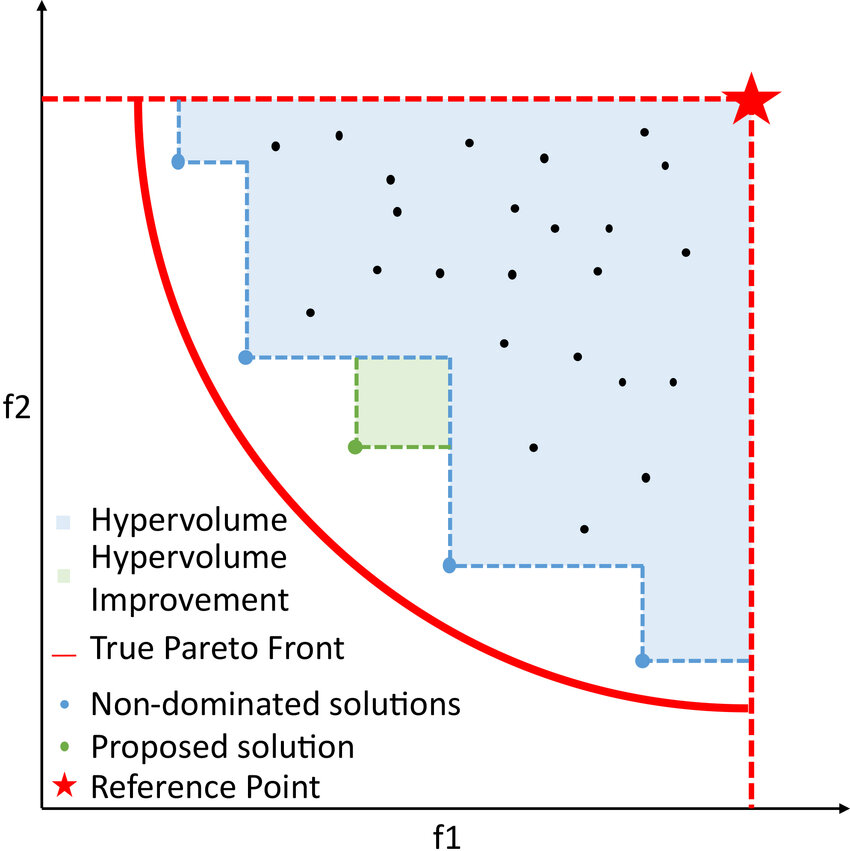
* Hypervolume: measures the volume of the objective space enclosed between the Pareto front and a reference point, with the reference point representing the worst possible solution for each objective.

Fig. 18 Hypervolume of a convex minimization problem []

# Chapter 4. Integration of theoretical notions into practical part of the application

## Project architecture

## NSGAII algorithm

The NSGAII algorithm was developed through jMetal library, which was integrated in the project. The algorithm described at chapter 3.1.1.2. has the following diagram representation:

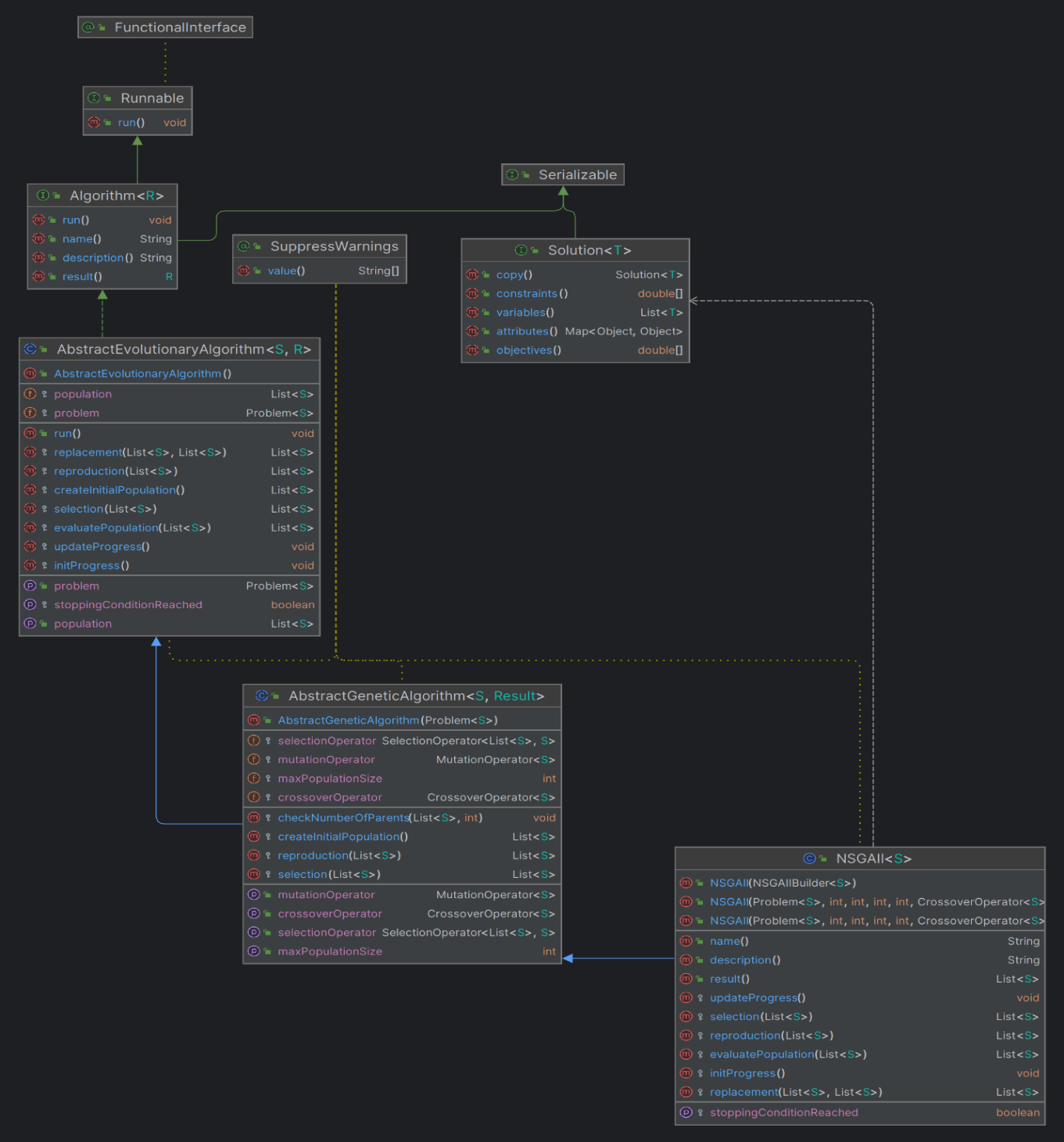


Fig. 19 NSGAII UML diagram

## SMPSO algorithm

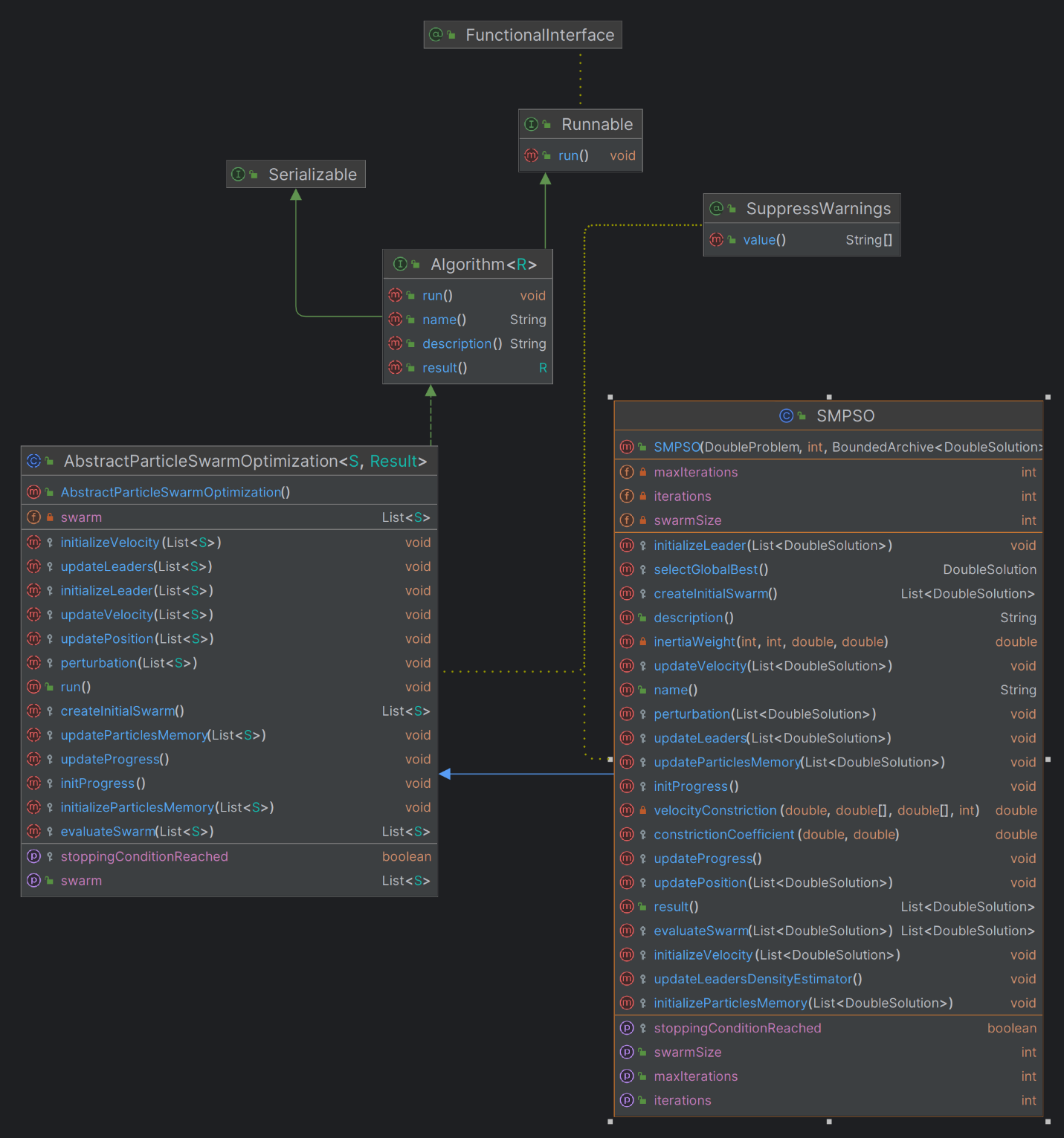
The SMPSO algorithm was developed through jMetal library, which was integrated in the project. The algorithm described at chapter 3.2.2 has the following diagram representation:

Fig. 20 SMPSO UML diagram

## Superposition of the algorithms

Multi-objective evolutionary optimization (MOEO) is a powerful method for solving problems that have multiple conflicting goals. In MOEO, a group of potential solutions evolves over time, with each solution balancing different objectives. The aim is to find solutions that are as close as possible to the Pareto-optimal front, meaning no solution is better in all aspects than any other. The main challenge is balancing how quickly the solutions improve (convergence speed) with maintaining a variety of solutions (diversity). To tackle this, a new approach called superposition of populations has been introduced to enhance MOEO.

A few advantages of the superposition are the following ones:

* Improved diversity:
* Multi-objective algorithms strive to keep a diverse set of solutions to fully cover the Pareto front. NSGA-II achieves this by using non-dominated sorting and crowding distance mechanisms, which help spread out the solutions.
* SMPSO uses a crowding distance mechanism and a velocity clamping technique to maintain diversity, ensuring that particles explore various regions of the search space.
* Improves Convergence:
* While maintaining diversity, multi-objective algorithms aim to converge toward the Pareto front. NSGA-II achieves this balance through elitism, crowding distance sorting, and tournament selection, promoting convergence toward the true Pareto front.
* SMPSO incorporates a speed constraint mechanism that prevents particles from moving too fast, allowing for better convergence towards the Pareto front.
* Scalability to high-dimensional problems:
* Multi-objective algorithms are designed to handle high-dimensional search spaces effectively. Both NSGA-II and SMPSO can efficiently handle problems with numerous decision variables by thoroughly exploring the search space.
* Both NSGA-II and SMPSO are population-based algorithms that maintain a diverse set of solutions, which can help in exploring high-dimensional search spaces more thoroughly.
* Potential for parallel implementation:
* Multi-objective algorithms can be parallelized to exploit modern computing architectures effectively. NSGA-II and SMPSO can benefit from parallelization techniques such as parallel evaluation of solutions, parallel selection, and parallel update of the population.
* Parallel implementations of NSGA-II and SMPSO have been explored in various research studies, leveraging multi-core CPUs, GPU computing, and distributed computing environments for improved performance.

These characteristics make multi-objective algorithms like NSGA-II and SMPSO ideal for tackling intricate optimization challenges with numerous conflicting goals. By balancing diversity and convergence while being scalable and amenable to parallel implementation, these algorithms offer efficient and effective solutions across various domains.

To explain better the superposition concept, in the followings subchapters, the implementation of NSGAII and SMPSO superposition is described.

## Superposition structure

For implementing the superposition concept, the following classes were added in the project:

* AbstractSuperPositionGA
* AbstractSuperPositionPSO
* AbstractParticleSwarm
* AbstractGeneticAlgorithm
* SuperPositionNSGAII
* SuperPositionSMPSO
* SuperPositionCombinator1
* AbstractSuperPositionGAPSOCombinator1
* GeneticAlgorithmWrapper
* ParticleSwarmWrapper

### AbstractGeneticAlgorithm class

In jMetal 6.0, the AbstractGeneticAlgorithm class serves as the base class for genetic algorithm implementations. It provides common functionality and structure for implementing various genetic algorithms:

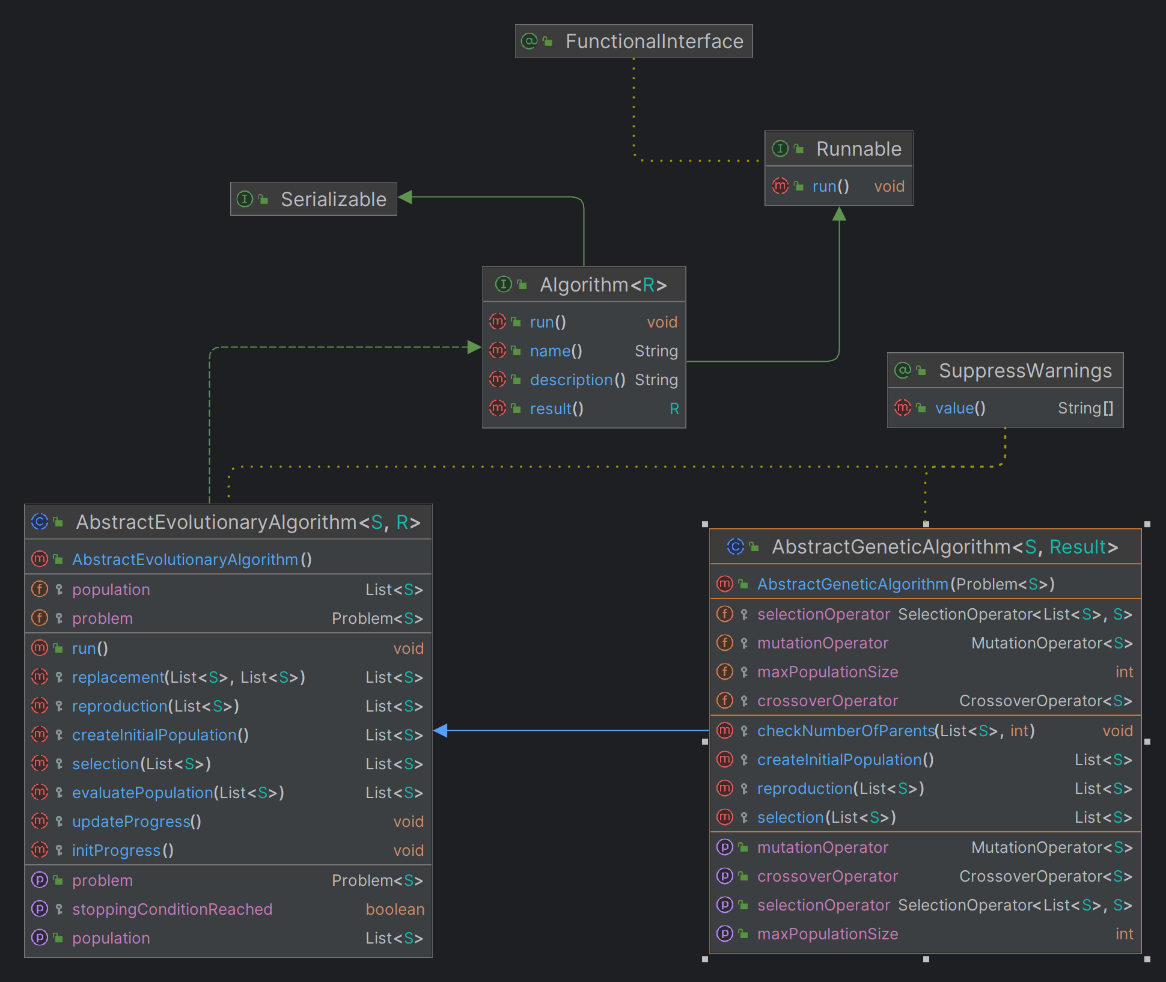


Fig. 21 AbstractGeneticAlgorithm UML diagram

1. **Class Structure**: The **AbstractGeneticAlgorithm** class is typically an abstract class, meaning it cannot be instantiated directly. It contains common attributes and methods shared by genetic algorithms.
2. **Attributes**:
   * **problem**: A reference to the problem being solved. It encapsulates the optimization problem and its characteristics.
   * **population**: The current population of candidate solutions.
   * **maxIterations**: The maximum number of generations the algorithm will execute.
   * **currentIteration**: The current iteration or generation number.
   * **evaluations**: The number of times the algorithm evaluates the objective function.
   * **selectionOperator**: The selection operator used to select parents for reproduction.
   * **crossoverOperator**: The crossover operator used to create offspring solutions from selected parents.
   * **mutationOperator**: The mutation operator used to introduce variability in the population.
   * Other parameters related to algorithm configuration, such as population size, probability parameters, etc.
3. **Methods**:
   * **AbstractGeneticAlgorithm(Problem problem)**: Constructor to initialize the genetic algorithm with a specific optimization problem.
   * **initializePopulation()**: Initializes the population of candidate solutions. This method typically generates random solutions or uses some heuristic initialization strategy.
   * **evaluatePopulation()**: Assesses the objective functions for each solution within the population.
   * **updateProgress()**: Updates the algorithm's progress, such as incrementing the iteration count and evaluation count.
   * **shouldStop()**: Checks if the termination criterion is met (e.g., maximum iterations reached).
   * **selectParents()**: Selects parent solutions from the current population for reproduction using the selection operator.
   * **crossover()**: Applies the crossover operator to selected parents to produce offspring solutions.
   * **mutate()**: Applies the mutation operator to introduce variability in the population.
   * **replacement()**: Performs survivor selection to determine the next generation of solutions.
   * **run()**: Executes the genetic algorithm by iteratively applying selection, crossover, mutation, and replacement until the termination criterion is met.
   * Other helper methods for logging, statistical analysis, etc.

### 4.4.1.2. GeneticAlgorithmWrapper class

This class is a wrapper for a genetic algorithm class to access easier the other private or protected members of the genetic algorithms, such as population, stop condition and so on. Its role is to help further in the process of the superposition implementation. The class has the following code:

|  |
| --- |
| public class GeneticAlgorithmWrapper**<**S**>**  **{**  protected AbstractGeneticAlgorithm**<**S**,** List**<**S**>>** algorithm**;**  public GeneticAlgorithmWrapper**(**AbstractGeneticAlgorithm**<**S**,** List**<**S**>>** algorithm**)**  **{**  **this.**algorithm **=** algorithm**;**  **}**  public List**<**S**>** createInitialPopulation**()**  **{**  **return** algorithm**.**createInitialPopulation**();**  **}**  public List**<**S**>** evaluatePopulation**(**List**<**S**>** population**)**  **{**  **return** algorithm**.**evaluatePopulation**(**population**);**  **}**  public boolean isStoppingConditionReached**()**  **{**  **return** algorithm**.**isStoppingConditionReached**();**  **}**  public List**<**S**>** selection**(**List**<**S**>** population**)**  **{**  **return** algorithm**.**selection**(**population**);**  **}**  public List**<**S**>** reproduction**(**List**<**S**>** population**)**  **{**  **return** algorithm**.**reproduction**(**population**);**  **}**  public List**<**S**>** replacement**(**List**<**S**>** population**,** List**<**S**>** offspringPopulation**)**  **{**  **return** algorithm**.**replacement**(**population**,** offspringPopulation**);**  **}**  public void initProgress**()**  **{**  algorithm**.**initProgress**();**  **}**  public void updateProgress**()**  **{**  algorithm**.**updateProgress**();**  **}**  public Problem**<**S**>** getProblem**()**  **{**  **return** algorithm**.**getProblem**();**  **}**  public List**<**S**>** result**()**  **{**  **return** algorithm**.**result**();**  **}**  public int getMaxPopulationSize**(){** **return** algorithm**.**maxPopulationSize**;** **}**  public int getMaxIterations**()** **throws** Exception  **{**  Class myClass **=** algorithm**.**getClass**();**  Field myField **=** getField**(**myClass**,** "maxIterations"**);**  myField**.**setAccessible**(true);**  **return** **(**int**)** myField**.**get**(**algorithm**);**  **}**  public void setIterations**(**int n**)** **throws** Exception  **{**  Class myClass **=** algorithm**.**getClass**();**  Field myField **=** getField**(**myClass**,** "iterations"**);**  myField**.**setAccessible**(true);**  myField**.**set**(**algorithm**,** n**);**  **}**  protected Field getField**(**Class clazz**,** String fieldName**)** **throws** NoSuchFieldException  **{**  **try**  **{**  **return** clazz**.**getDeclaredField**(**fieldName**);**  **}**  **catch** **(**NoSuchFieldException e**)**  **{**  Class superClass **=** clazz**.**getSuperclass**();**  **if** **(**superClass **==** **null)**  **{**  **throw** e**;**  **}**  **else**  **{**  **return** getField**(**superClass**,** fieldName**);**  **}**  **}**  **}**  **}** |

The diagram class is represented in the following image:

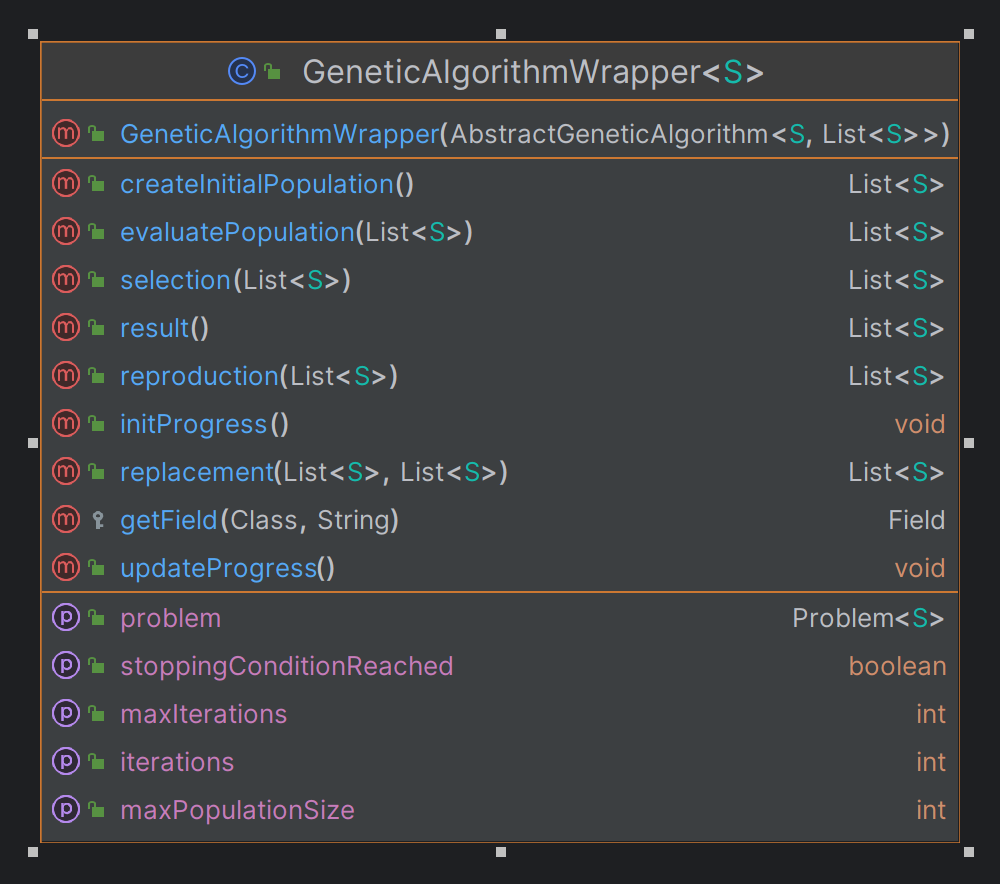


Fig. 22 GeneticAlgorithmWrapper UML diagram



### AbstractSuperPositionGA class

The class structure is very simple: it has to be easily inherited by any other genetic algorithm developed in the jMetal library. With the run() method, the algorithm can create the initial population and then evaluate it. While the stop condition is not met, the selection, reproduction and evaluate processes are executed, so a new generation can be invoked. The class has the following code:

|  |
| --- |
| public abstract class AbstractSuperPositionGA**<**S**,** R**>** **extends** AbstractEvolutionaryAlgorithm**<**S**,** R**>** **implements** Runnable  **{**  protected GeneticAlgorithmWrapper**<**S**>** algorithmWrapper**;**  public volatile boolean isWaiting**;**  public IAction onNewGeneration**;**  public AbstractSuperPositionGA**(**GeneticAlgorithmWrapper**<**S**>** wrapper**)**  **{**  algorithmWrapper **=** wrapper**;**  isWaiting **=** **false;**  setProblem**(**algorithmWrapper**.**getProblem**());**  **}**  @Override  public void run**()**  **{**  List**<**S**>** offspringPopulation**;**  List**<**S**>** matingPopulation**;**  population **=** createInitialPopulation**();**  population **=** evaluatePopulation**(**population**);**  initProgress**();**  **while** **(!**isStoppingConditionReached**())** **{**  matingPopulation **=** selection**(**population**);**  offspringPopulation **=** reproduction**(**matingPopulation**);**  offspringPopulation **=** evaluatePopulation**(**offspringPopulation**);**  population **=** replacement**(**population**,** offspringPopulation**);**  onNewGeneration**.**invoke**();**  updateProgress**();**  isWaiting **=** **true;**  **while** **(**isWaiting**)**  **{**  Thread**.**onSpinWait**();**  **}**  **}**  **}**  @Override  protected List**<**S**>** createInitialPopulation**()**  **{**  **return** algorithmWrapper**.**createInitialPopulation**();**  **}**  @Override  protected List**<**S**>** evaluatePopulation**(**List**<**S**>** population**)**  **{**  **return** algorithmWrapper**.**evaluatePopulation**(**population**);**  **}**  @Override  public boolean isStoppingConditionReached**()**  **{**  **return** algorithmWrapper**.**isStoppingConditionReached**();**  **}**  @Override  protected List**<**S**>** selection**(**List**<**S**>** population**)**  **{**  **return** algorithmWrapper**.**selection**(**population**);**  **}**  @Override  protected List**<**S**>** reproduction**(**List**<**S**>** population**)**  **{**  **return** algorithmWrapper**.**reproduction**(**population**);**  **}**  @Override  protected List**<**S**>** replacement**(**List**<**S**>** population**,** List**<**S**>** offspringPopulation**)**  **{**  **return** algorithmWrapper**.**replacement**(**population**,** offspringPopulation**);**  **}**  @Override  protected void initProgress**()**  **{**  algorithmWrapper**.**initProgress**();**  **}**  @Override  protected void updateProgress**()**  **{**  algorithmWrapper**.**updateProgress**();**  **}**  public int getMaxPopulationSize**()** **{**  **return** algorithmWrapper**.**getMaxPopulationSize**()** **;**  **}**  public void forceStoppingCondition**()**  **try** **{**  int maxIterations **=** algorithmWrapper**.**getMaxIterations**();**  algorithmWrapper**.**setIterations**(**maxIterations**);**  **}**  **catch** **(**Exception ex**)**  **{**  // TODO  **}**  **}**  **}** |

The UML diagram of the class with the other class connections is represented in the following diagram:

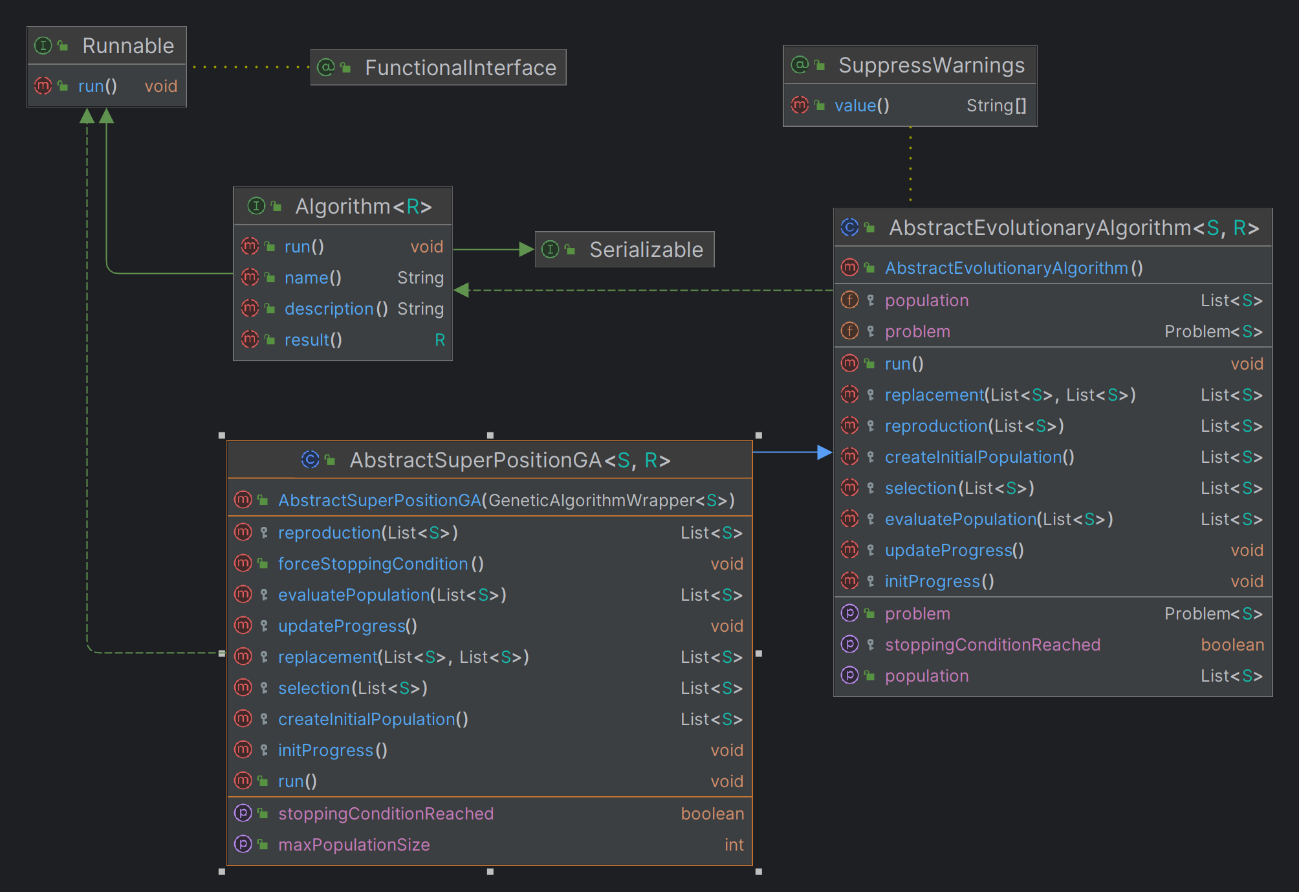


Fig. 23 AbstractSuperPositionGA UML diagram

### 4.4.1.4. AbstractParticleSwarmOptimization class

In jMetal 6.0, the AbstractParticleSwarmOptimization class serves as the base class for particle swarm optimization (PSO) algorithms. It has the following attributes and classes in its development:

1. **Attributes**:
   * **problem**: A reference to the problem being solved. It encapsulates the optimization problem and its characteristics.
   * **swarm**: The current group of particles, with each particle representing a potential solution in the search space.
   * **maxIterations**: The maximum number of generations the algorithm will execute
   * **currentIteration**: The current iteration or generation number.
   * **evaluations**: The number of times the algorithm evaluates the objective function.
   * **leader**: The global best solution found by the swarm.
   * **leaders**: A collection of historical global best solutions found during the optimization process.
   * Parameters related to algorithm configuration, such as swarm size, inertia weight, cognitive and social components, etc.
2. **Methods**:
   * **AbstractParticleSwarmOptimization(Problem problem)**: Constructor to initialize the PSO algorithm with a specific optimization problem.
   * **initializeSwarm()**: Initializes the swarm of particles. This method typically generates random particles within the search space.
   * **evaluateSwarm()**: Evaluates the objective functions for each particle in the swarm.
   * **updateLeaders()**: Updates the collection of historical global best solutions found during the optimization process.
   * **updateParticles()**: Updates each particle's velocity and position in the swarm using the PSO equations.
   * **updateProgress()**: Updates the algorithm's progress, such as incrementing the iteration count and evaluation count.
   * **stoppingConditionReached:** Checks if the termination criterion is met (e.g., maximum iterations reached).
   * **run()**: Executes the PSO algorithm by iteratively updating particles' positions and velocities until the termination criterion is met.
   * Other helper methods for logging, statistical analysis, etc.

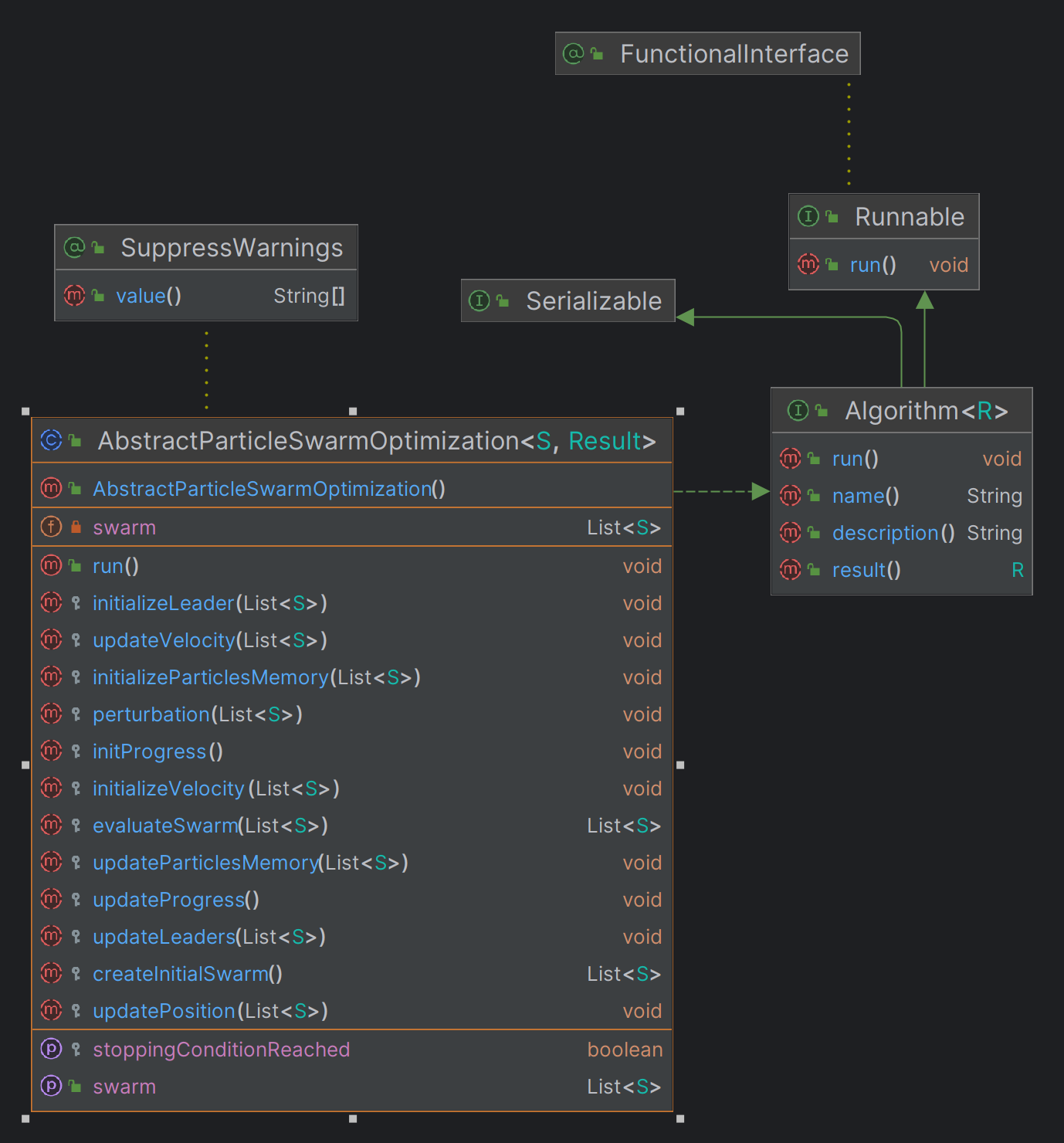


Fig. 24 AbstractParticleSwarmOptimization UML class

### 4.4.1.5. ParticleSwarmWrapper class

This class is a wrapper for a PSO algorithm class to access easier the other private or protected members of the genetic algorithms, such as population, stop condition, etc. Its role is to help further in the process of the superposition implementation. It contains the main methods helpful for every particle swarm optimization algorithm that is using it. From creating the initial swarm to the final result. The class has the following code:

|  |
| --- |
| public class ParticleSwarmWrapper**<**S**>**  **{**  protected AbstractParticleSwarmOptimization**<**S**,** List**<**S**>>** algorithm**;**  public ParticleSwarmWrapper**(**AbstractParticleSwarmOptimization**<**S**,** List**<**S**>>** algorithm**)**  **{**  **this.**algorithm **=** algorithm**;**  **}**  public List**<**S**>** createInitialSwarm**()**  **{**  **return** algorithm**.**createInitialSwarm**();**  **}**  public List**<**S**>** evaluateSwarm**(**List**<**S**>** swarm**)**  **{**  **return** algorithm**.**evaluateSwarm**(**swarm**);**  **}**  public void initializeLeader**(**List**<**S**>** swarm**)**  **{**  algorithm**.**initializeLeader**(**swarm**);**  **}**  public void initializeParticlesMemory**(**List**<**S**>** swarm**)**  **{**  algorithm**.**initializeParticlesMemory**(**swarm**);**  **}**  public void initializeVelocity**(**List**<**S**>** swarm**)**  **{**  algorithm**.**initializeVelocity**(**swarm**);**  **}**  public void updateVelocity**(**List**<**S**>** swarm**)**  **{**  algorithm**.**updateVelocity**(**swarm**);**  **}**  public void updatePosition**(**List**<**S**>** swarm**)**  **{**  algorithm**.**updatePosition**(**swarm**);**  **}**  public void perturbation**(**List**<**S**>** swarm**)**  **{**  algorithm**.**perturbation**(**swarm**);**  **}**  public void updateLeaders**(**List**<**S**>** swarm**)**  **{**  algorithm**.**updateLeaders**(**swarm**);**  **}**  public void updateParticlesMemory**(**List**<**S**>** swarm**)**  **{**  algorithm**.**updateParticlesMemory**(**swarm**);**  **}**  public void initProgress**()**  **{**  algorithm**.**initProgress**();**  **}**  public void updateProgress**()**  **{**  algorithm**.**updateProgress**();**  **}**  public boolean isStoppingConditionReached**()**  **{**  **return** algorithm**.**isStoppingConditionReached**();**  **}**  public List**<**S**>** result**()**  **{**  **return** algorithm**.**result**();**  **}**  public List**<**S**>** getSwarm**()**  **{**  **return** algorithm**.**getSwarm**();**  **}**  public void setSwarm**(**List**<**S**>** swarm**)**  **{**  algorithm**.**setSwarm**(**swarm**);**  **}**  public int getMaxIterations**()** **throws** Exception  **{**  Class myClass **=** algorithm**.**getClass**();**  Field myField **=** getField**(**myClass**,** "maxIterations"**);**  myField**.**setAccessible**(true);**  **return** **(**int**)** myField**.**get**(**algorithm**);**  **}**  public void setIterations**(**int n**)** **throws** Exception  **{**  Class myClass **=** algorithm**.**getClass**();**  Field myField **=** getField**(**myClass**,** "iterations"**);**  myField**.**setAccessible**(true);**  myField**.**set**(**algorithm**,** n**);**  **}**  protected Field getField**(**Class clazz**,** String fieldName**)** **throws** NoSuchFieldException  **{**  **try**  **{**  **return** clazz**.**getDeclaredField**(**fieldName**);**  **}**  **catch** **(**NoSuchFieldException e**)**  **{**  Class superClass **=** clazz**.**getSuperclass**();**  **if** **(**superClass **==** **null)**  **{**  **throw** e**;**  **}**  **else**  **{**  **return** getField**(**superClass**,** fieldName**);**  **}**  **}**  **}**  **}** |

The class has the following diagram:

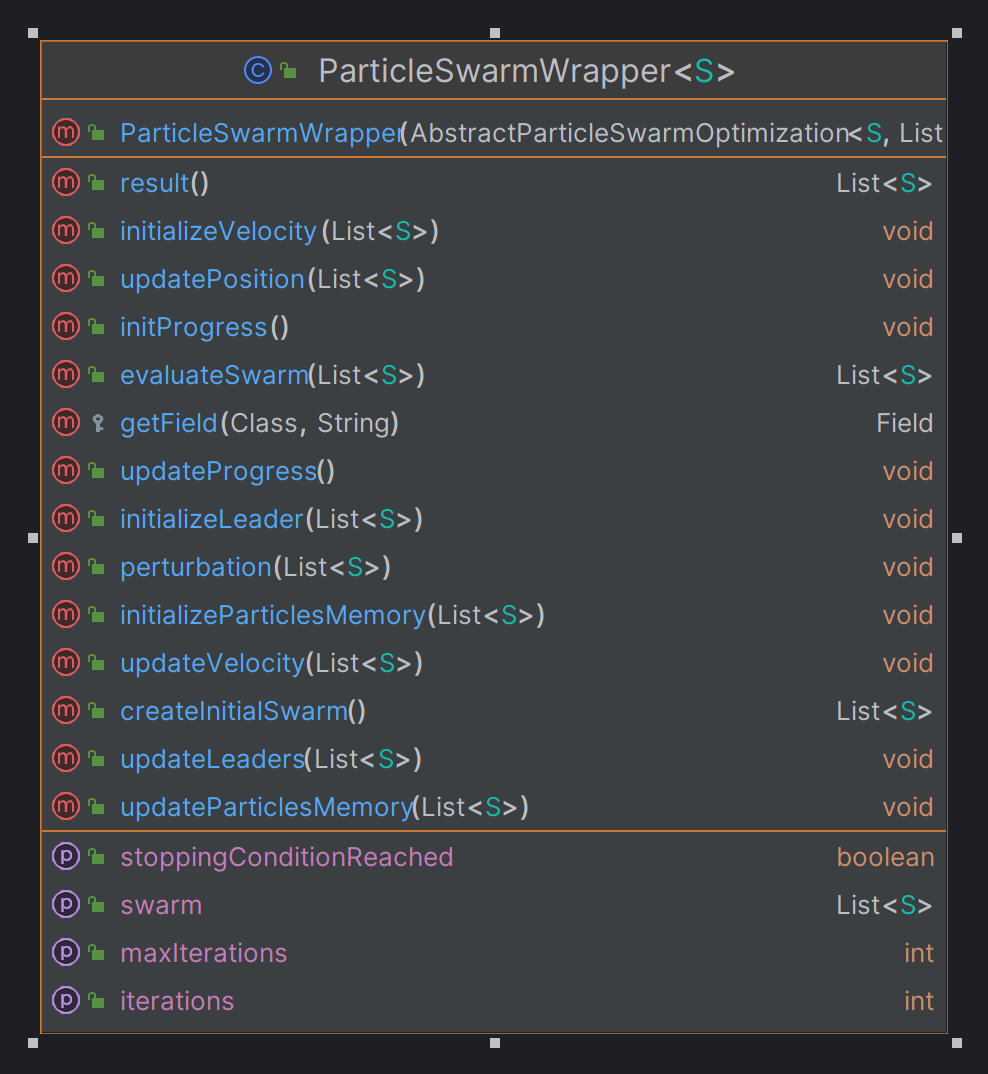


Fig. 25 ParticleSwarmWrapper UML diagram

### 4.4.1.6. AbstractSuperPositionPSO

The class structure is very simple: it has to be easily inherited by any other PSO algorithm developed in the jMetal library. This is developed in the same way as AbstractSuperPositionGA described at chapter 4.4.1.3, but for a PSO algorithm. The class has the following code:

|  |
| --- |
| public abstract class AbstractSuperPositionPSO**<**S**,** R**>** **implements** Algorithm**<**R**>**  **{**  protected ParticleSwarmWrapper**<**S**>** algorithmWrapper**;**  public volatile boolean isWaiting**;**  public IAction onNewGeneration**;**  public AbstractSuperPositionPSO**(**ParticleSwarmWrapper**<**S**>** wrapper**)**  **{**  algorithmWrapper **=** wrapper**;**  isWaiting **=** **false;**  **}**  @Override  public void run**()**  **{**  createInitialSwarm**();**  evaluateSwarm**();**  initializeVelocity**();**  initializeParticlesMemory**()** **;**  initializeLeader**()** **;**  initProgress**();**  **while** **(!**isStoppingConditionReached**())** **{**  updateVelocity**();**  updatePosition**();**  perturbation**();**  evaluateSwarm**();**  updateLeaders**();**  updateParticlesMemory**();**  onNewGeneration**.**invoke**();**  updateProgress**();**  isWaiting **=** **true;**  **while** **(**isWaiting**)**  **{**  Thread**.**onSpinWait**();**  **}**  **}**  **}**  public void createInitialSwarm**()** **{**  List**<**S**>** initialSwarm **=** algorithmWrapper**.**createInitialSwarm**();**  algorithmWrapper**.**setSwarm**(**initialSwarm**);**  **}**  public void evaluateSwarm**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  List**<**S**>** evaluatedSwarm **=** algorithmWrapper**.**evaluateSwarm**(**currentSwarm**);**  algorithmWrapper**.**setSwarm**(**evaluatedSwarm**);**  **}**  public void initializeLeader**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**initializeLeader**(**currentSwarm**);**  **}**  public void initializeParticlesMemory**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**initializeParticlesMemory**(**currentSwarm**);**  **}**  public void initializeVelocity**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**initializeVelocity**(**currentSwarm**);**  **}**  public void updateVelocity**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**updateVelocity**(**currentSwarm**);**  **}**  public void updatePosition**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**updatePosition**(**currentSwarm**);**  **}**  public void perturbation**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**perturbation**(**currentSwarm**);**  **}**  public void updateLeaders**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**updateLeaders**(**currentSwarm**);**  **}**  public void updateParticlesMemory**()** **{**  List**<**S**>** currentSwarm **=** algorithmWrapper**.**getSwarm**();**  algorithmWrapper**.**updateParticlesMemory**(**currentSwarm**);**  **}**  public void initProgress**()**  **{**  algorithmWrapper**.**initProgress**();**  **}**  public void updateProgress**()**  **{**  algorithmWrapper**.**updateProgress**();**  **}**  public boolean isStoppingConditionReached**()**  **{**  **return** algorithmWrapper**.**isStoppingConditionReached**();**  **}**  public List**<**S**>** getSwarm**()**  **{**  **return** algorithmWrapper**.**getSwarm**();**  **}**  public void setSwarm**(**List**<**S**>** newSwarm**)**  **{**  algorithmWrapper**.**setSwarm**(**newSwarm**);**  **}**  public void forceStoppingCondition**()**  **{**  **try** **{**  int maxIterations **=** algorithmWrapper**.**getMaxIterations**();**  algorithmWrapper**.**setIterations**(**maxIterations**);**  **}**  **catch** **(**Exception ex**)**  **{**  // TODO  **}**  **}**  **}** |

The following diagram represent the fields and methods of this class, together with the connections with the other classes from the project.



Fig. 26 AbstractSuperPositionPSO UML diagram

### 4.4.1.7. SuperPositionNSGAII class

The implementation of this class is abstract and helps in the final result of the superposition method of NSGAII and SMPSO algorithms.

|  |
| --- |
| public class SuperPositionNSGAII**<**S **extends** Solution**<?>>** **extends** AbstractSuperPositionGA**<**S**,** List**<**S**>>**  **{**  public SuperPositionNSGAII**(**GeneticAlgorithmWrapper**<**S**>** wrapper**)**  **{**  **super(**wrapper**);**  **}**  @Override  public String name**()**  **{**  **return** "SuperPositionNSGAII"**;**  **}**  @Override  public String description**()**  **{**  **return** "SuperPosition Non-dominated Sorting Genetic Algorithm II"**;**  **}**  @Override  public List**<**S**>** result**()**  **{**  **return** algorithmWrapper**.**result**();**  **}**  **}** |

The diagram of the class and its connections is represented through the following:

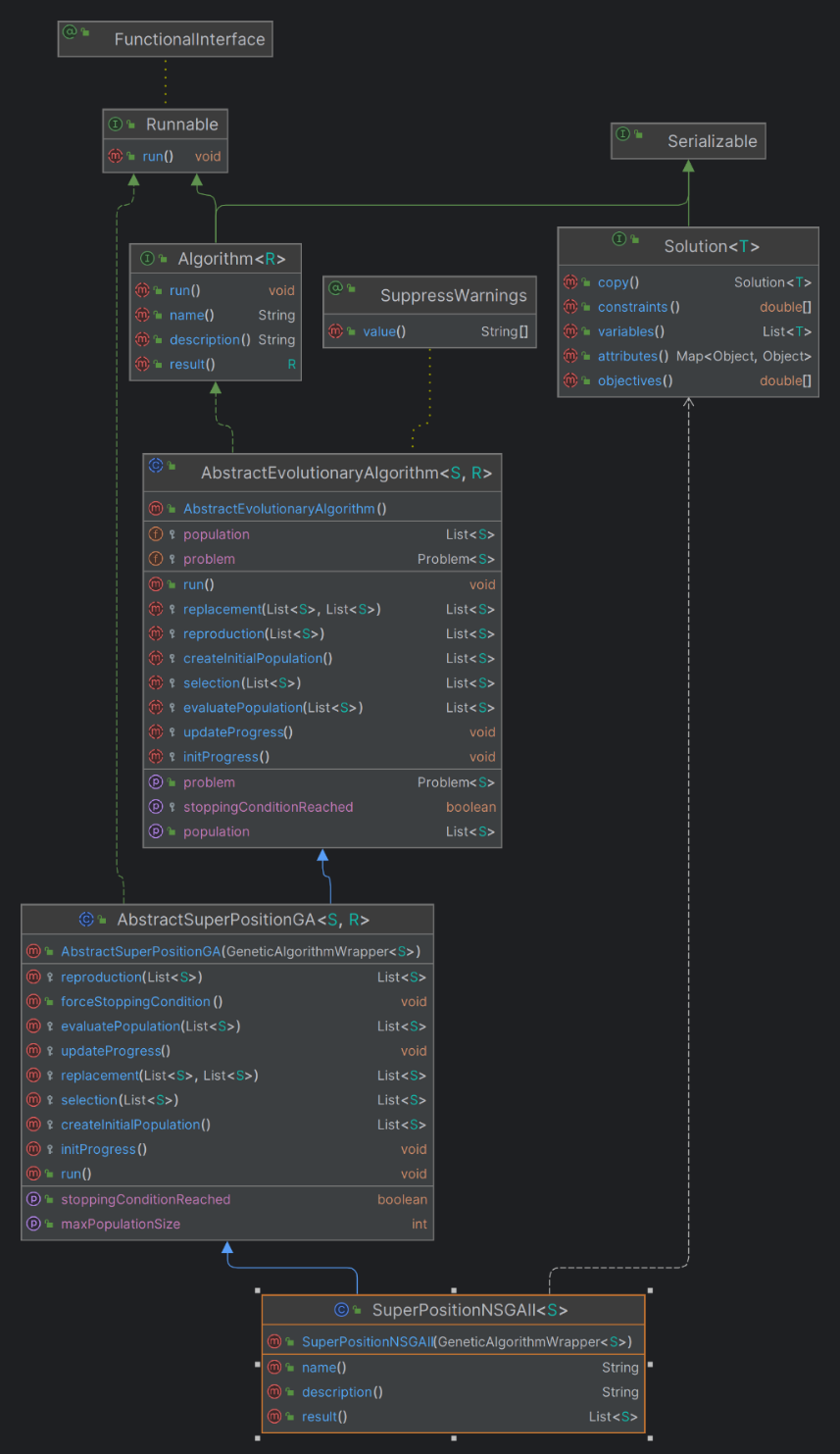


Fig. 27 SuperpositionNSGAII UML diagram

### 4.4.1.8. SuperPositionSMPSO class

The implementation of this class is abstract and helps in the final result of the superposition method of NSGAII and SMPSO algorithms.

|  |
| --- |
| public class SuperPositionSMPSO **<**S **extends** Solution**<?>>** **extends** AbstractSuperPositionPSO**<**S**,** List**<**S**>>**  **{**  public SuperPositionSMPSO**(**ParticleSwarmWrapper**<**S**>** wrapper**)** **{**  **super(**wrapper**);**  **}**  @Override  public String name**()**  **{**  **return** "SuperPositionSMPSO"**;**  **}**  @Override  public String description**()**  **{**  **return** "SuperPosition Speed-constrained Multi-objective PSO"**;**  **}**  @Override  public List**<**S**>** result**()** **{** **return** algorithmWrapper**.**result**();** **}**  **}** |

The diagram of the class and its connections is represented through the following:

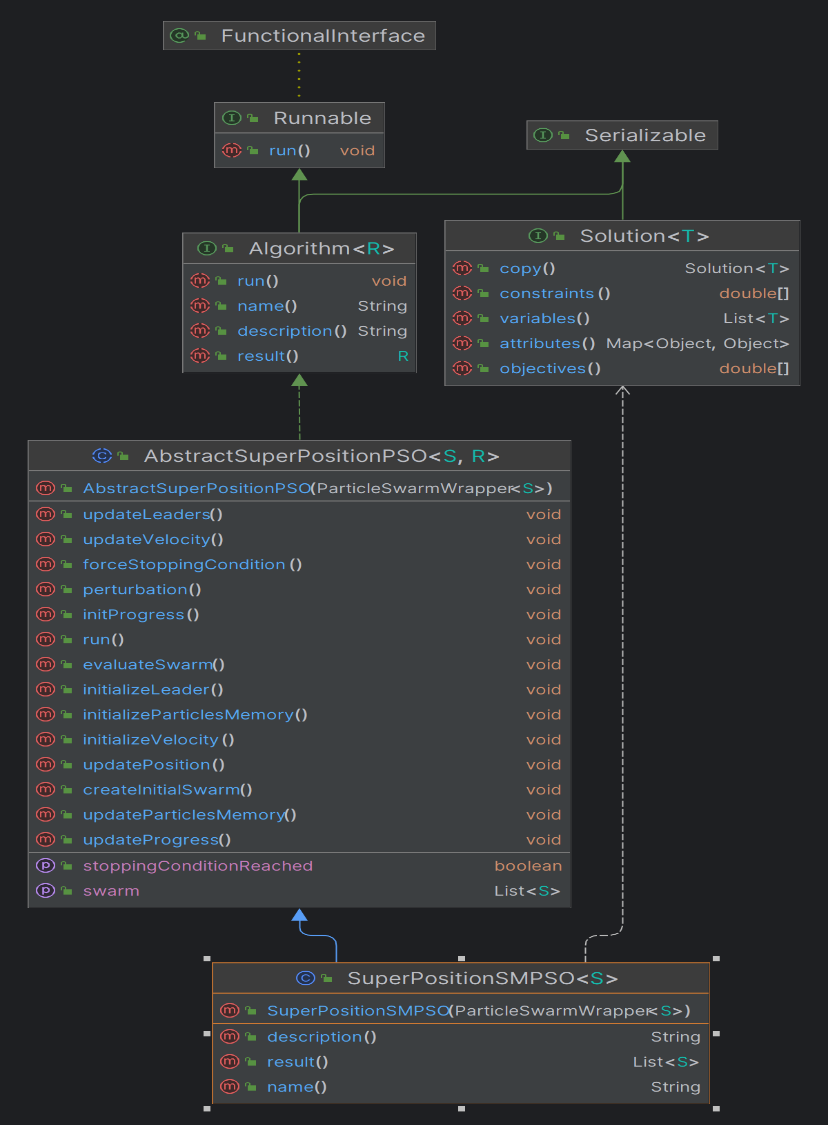


Fig. 28 Superposition SMPSO UML diagram

### 4.4.1.9. SuperPositionCombinator1 class

The **SuperPositionCombinator1** class extends **AbstractSuperPositionGAPSOCombinator1** and acts as a combinator for combining two superposition-based optimization algorithms: a genetic algorithm (**AbstractSuperPositionGA**) and a particle swarm optimization algorithm (**AbstractSuperPositionPSO**). The population from the first algorithm, in this case, NSGAII is combined with the swarm of the SMPSO algorithm, and a new population is generated in the end, after calculated the fronts. The class code is represented as the following:

|  |
| --- |
| public class SuperPositionCombinator1**<**S **extends** Solution**<?>>** **extends** AbstractSuperPositionGAPSOCombinator1**<**S**>** **{**  public SuperPositionCombinator1**(**AbstractSuperPositionGA**<**S**,** List**<**S**>>** alg1**,** AbstractSuperPositionPSO**<**S**,** List**<**S**>>** alg2**)** **{**  **super(**alg1**,** alg2**);**  **}**    @Override  protected List**<**S**>** ApplySuperPosition**()**  **{**  // 1. COMBINE POPULATIONS  List**<**S**>** combinedPopulation **=** **new** ArrayList**<>();**  // 2. RANKING AND TAKING THE BEST FRONT - we make this step to take the best 100 individuals from the combined population  combinedPopulation**.**addAll**(**algorithm1**.**getPopulation**());**  combinedPopulation**.**addAll**(**algorithm2**.**getSwarm**());**  // 3. GENERATE NEW POPULATION - here we set the 100 best as the new generation  Ranking**<**S**>** fronts **=** computeRanking**(**combinedPopulation**);**  List**<**S**>** newPopulation **=** take**(**fronts**,** algorithm1**.**getMaxPopulationSize**());**  // 4. SET NEW POPULATION FOR EACH ALGORITHM  algorithm1**.**setPopulation**(**newPopulation**);**  algorithm2**.**setSwarm**(**newPopulation**);**  **return** newPopulation**;**  **}**  **}** |

Below, the diagram of the class is represented, together with the connections:

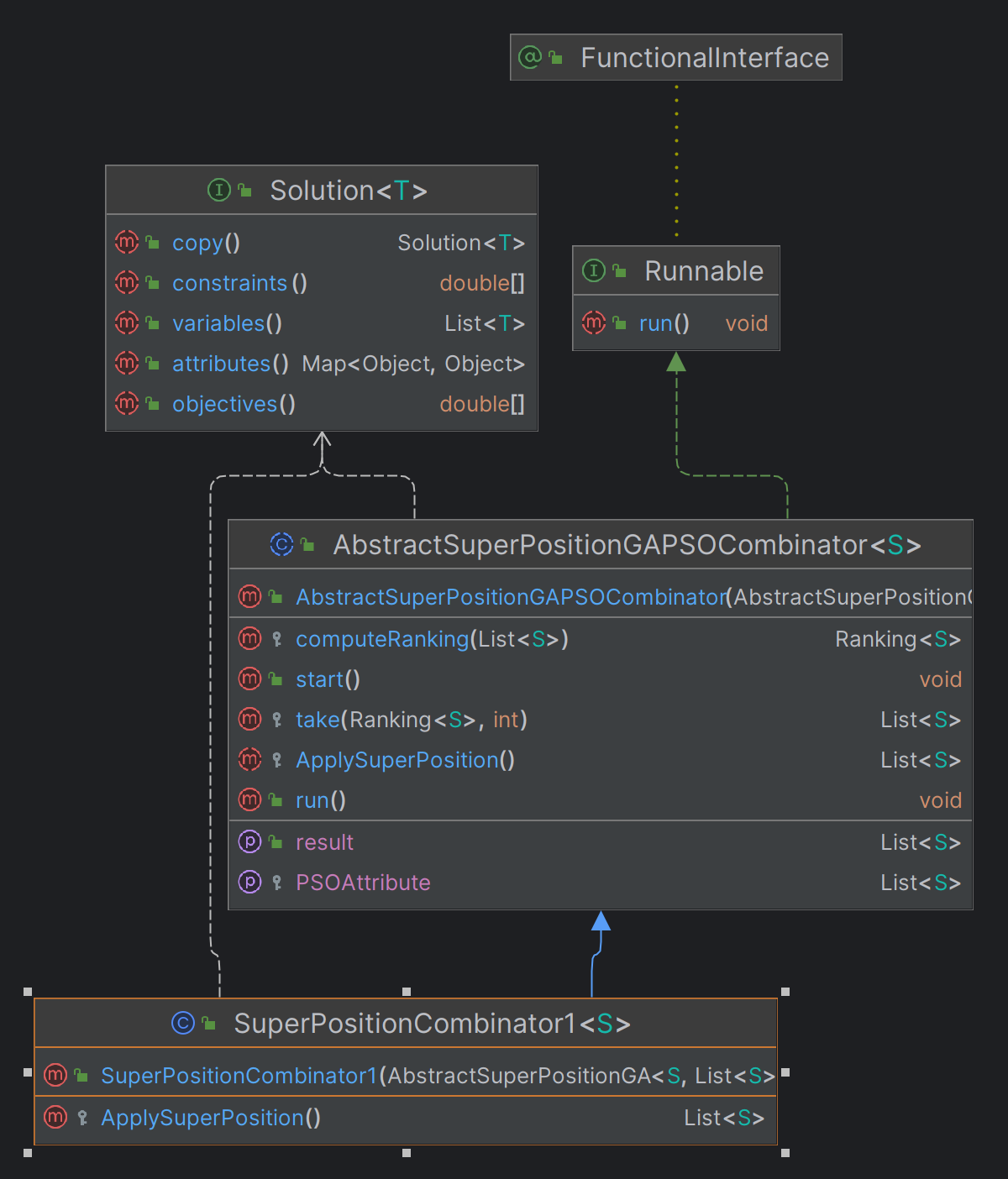


Fig. 29 SuperpositionCombinator1 UML diagram

### 4.4.1.10. AbstractSuperPositionGAPSOCombinator1

This class is the most important one, because it simulated the whole superposition process of a genetic algorithm, like NSGAII and a particle swarm optimization algorithm, like SMPSO algorithm. There are two lists representing both algorithms and also two events onNewGenerationDone and onSuperPositionDone that are invoked after the algorithm has reached its best solution and the result of the superposition is saved. The run() method executed the combination of the algorithms for providing the superposition result. It depends if one algorithm or another has reached its stopping condition or not, so the process can continue or stop at some solution during the execution of the process. The new population is obtained after calculated the pareto fronts and then it is returned.

|  |
| --- |
| public abstract class AbstractSuperPositionGAPSOCombinator**<**S **extends** Solution**<?>>** **implements** Runnable  **{**  protected AbstractSuperPositionGA**<**S**,** List**<**S**>>** algorithm1**;**  protected AbstractSuperPositionPSO**<**S**,** List**<**S**>>** algorithm2**;**  protected boolean bothAlgorithmsFinished**;**  protected List**<**S**>** resultPopulation**;**  public IAction onNewGenerationDone**;**  public IAction onSuperPositionDone**;**  public AbstractSuperPositionGAPSOCombinator**(**AbstractSuperPositionGA**<**S**,** List**<**S**>>** alg1**,**  AbstractSuperPositionPSO**<**S**,** List**<**S**>>** alg2**)**  **{**  algorithm1 **=** alg1**;**  algorithm2 **=** alg2**;**  bothAlgorithmsFinished **=** **false;**  **}**  protected abstract List**<**S**>** ApplySuperPosition**();**  public void start**()**  **{**  bothAlgorithmsFinished **=** **false;**  **}**  public List**<**S**>** getResult**()**  **{**  **return** resultPopulation**;**  **}**  public void run**()**  **{**  **while(!**bothAlgorithmsFinished**)**  **{**  **if(**algorithm1**.**isWaiting **&&** algorithm2**.**isWaiting**)**  **{**  **if(**algorithm1**.**isStoppingConditionReached**())**  **{**  **if(!**algorithm2**.**isStoppingConditionReached**())**  **{**  algorithm2**.**forceStoppingCondition**();**  **}**  bothAlgorithmsFinished **=** **true;**  algorithm1**.**isWaiting **=** **false;**  algorithm2**.**isWaiting **=** **false;**  resultPopulation **=** ApplySuperPosition**();**  onNewGenerationDone**.**invoke**();**  onSuperPositionDone**.**invoke**();**  **}**  **else** **if(**algorithm2**.**isStoppingConditionReached**())**  **{**  algorithm1**.**forceStoppingCondition**();**  bothAlgorithmsFinished **=** **true;**  algorithm1**.**isWaiting **=** **false;**  algorithm2**.**isWaiting **=** **false;**  resultPopulation **=** ApplySuperPosition**();**  onSuperPositionDone**.**invoke**();**  **}**  **else**  **{**  resultPopulation **=** ApplySuperPosition**();**  onNewGenerationDone**.**invoke**();**  algorithm1**.**isWaiting **=** **false;**  algorithm2**.**isWaiting **=** **false;**  **}**  **}**  **}**  **}**  // 2. RANKING AND TAKING THE BEST FRONT - we make this step to take the best 100 individuals from the combined population  protected Ranking**<**S**>** computeRanking**(**List**<**S**>** solutionList**)**  **{**  Ranking**<**S**>** ranking **=** **new** FastNonDominatedSortRanking**<>();**  ranking**.**compute**(**solutionList**)** **;**  **return** ranking **;**  **}**  // 3. GENERATE NEW POPULATION - here we set the 100 best as the new generation  protected List**<**S**>** take**(**Ranking**<**S**>** fronts**,** int populationCount**)**  **{**  List**<**S**>** population **=** **new** ArrayList**<>();**  int noOfSelectedIndividuals **=** 0**;**  int noOfFronts **=** fronts**.**getNumberOfSubFronts**();**  **for** **(**int frontIndex **=** 0**;** frontIndex **<=** noOfFronts **-** 1**;** frontIndex**++)**  **{**  List**<**S**>** currentFront **=** fronts**.**getSubFront**(**frontIndex**);**  int currentFrontSize **=** currentFront**.**size**();**  int currentIndividualIndex **=** 0**;**  **while** **(**noOfSelectedIndividuals **<** populationCount **&&** currentIndividualIndex **<** currentFrontSize**)**  **{**  population**.**add**(**currentFront**.**get**(**currentIndividualIndex**));**  noOfSelectedIndividuals**++;**  currentIndividualIndex**++;**  **}**  **}**  **return** population**;** |

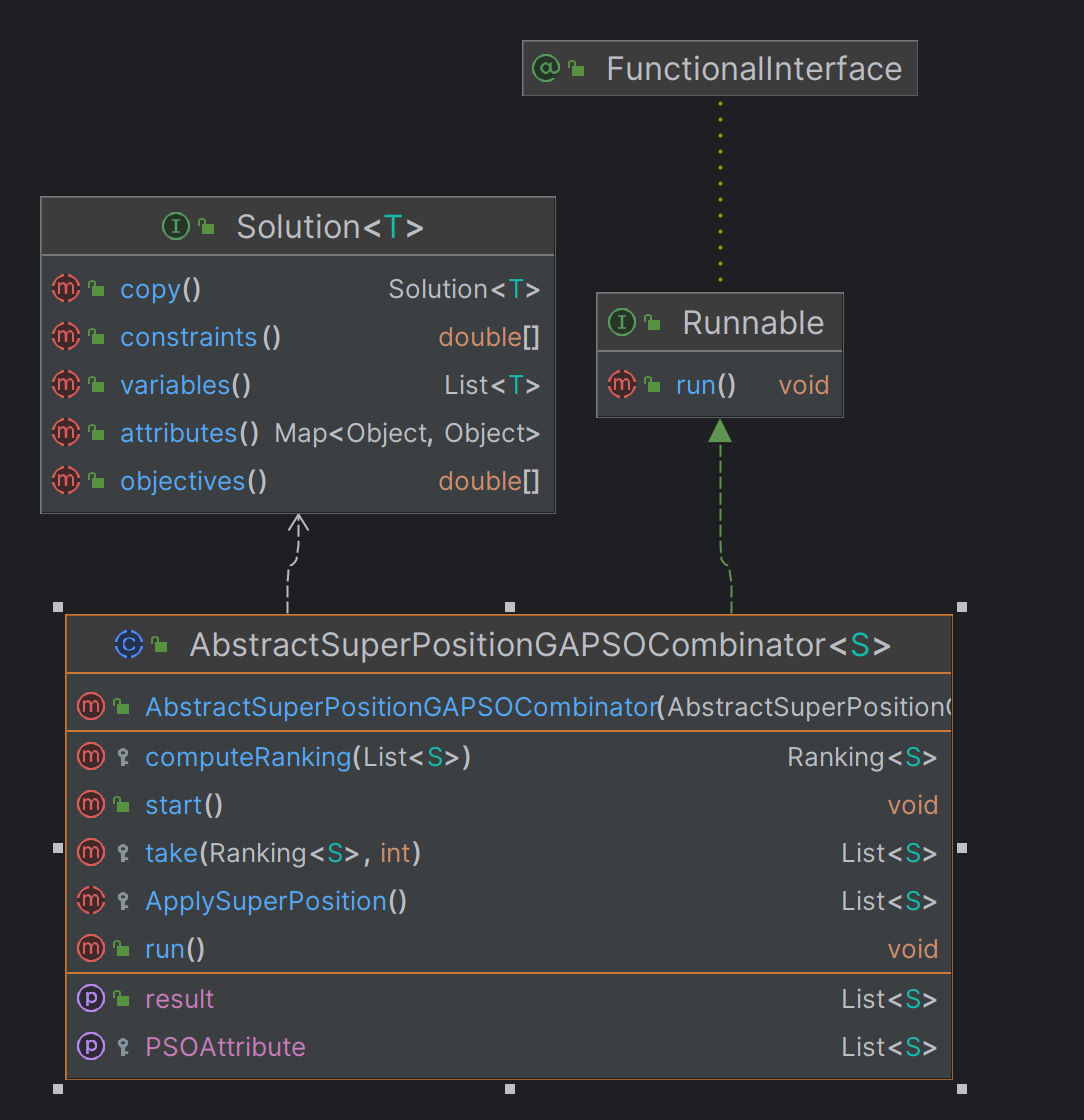


Fig. 30 AbstractSuperPositionGAPSOCombinator1 UML diagram

## 4.5. Simulations without GAP simulator

In the following subchapter, simulations of superposition of NSGAII and SMPSO algorithms are represented, using the DTLZ1 problem available in jMetal library. The results of the simulations are generated by two objectives obj1 and obj2. These two objectives have the responsibility to minimize an xOy function. The solution provided contains a set of Pareto optimal solutions. These solutions form a Pareto front in the objective space, representing the trade-offs between the objectives.

The algorithm was analyzed based on the Hypervolume and SetCoverage metrics, modifying the population and swarm size between 10 and 200 individuals. There were 5 simulations executed using DTZL1 problem.

### 4.5.1. NSGAII simulation

The results provided from the execution of NSGAII algorithm using the DTLZ1 problem are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 10 | 0.41 | 0.5 |
| 50 | 0.99 | 0.5 |
| 100 | 1.00 | 0.5 |
| 150 | 1.00 | 0.5 |
| 200 | 1.00 | 0.5 |

Analyzing the result of the hypervolume metric, we can see that as the population size increases, the hypervolume will maintain its value.

The coverage result is linear, as it has almost the same value for every population size of the algorithm, except the first simulation, where the population size is 10 individuals.

### 4.5.2. SMPSO simulation

The results provided from the execution of SMPSO algorithm using the DTLZ1 problem are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 10 | 0.5 | 0.5 |
| 50 | 0.5 | 0.5 |
| 100 | 0.5 | 0.5 |
| 150 | 0.5 | 0.5 |
| 200 | 0.5 | 0.5 |

Regarding the SMPSO algorithm, the results of coverage and hypervolume metrics were the same for all the simulations. The algorithm maintained a trend of 0.5 value for coverage and hypervolume too.

### 4.5.3. SuperPosition of NSGAII and SMPSO simulation

The results provided from the execution of superposition of NSGAII and SMPSO algorithms using the DTLZ1 problem are the following:

|  |  |  |
| --- | --- | --- |
| Population size | Hypervolume result | Set Coverage result |
| 10 | 0.41 | 0.5 |
| 50 | 0.99 | 0.5 |
| 100 | 1.00 | 0.5 |
| 150 | 1.00 | 0.5 |
| 200 | 1.00 | 0.5 |

Analyzing the result of the hypervolume metric, we can see that as the population size increases, the hypervolume will maintain its value. For the hypervolume metric, we can see a slightly difference for every simulation, comparing to the results obtained at chapters 4.5.1 and 4.5.2. The superposition of the NSGAII and SMPSO generated better results for the hypervolume metric while analyzing its individuals.

The coverage result is linear, as it has the same value for every population size of the algorithm.

As a conclusion of these simulations, we can specify the followings:

* as the population size increases, the algorithms may converge more effectively towards the true Pareto front or a close approximation of it. Increasing the population size might not significantly affect the hypervolume value once convergence is reached.
* with a larger population size, the algorithms explore a bigger portion of the search space, potentially covering more areas of the Pareto front. This increased sampling density can lead to a more accurate approximation of the Pareto front, resulting in a stable hypervolume value as the population size grows.
* while the algorithms use random population initialization, increasing the population size can lead to a broader initial exploration of the search space. This initial exploration may result in a diverse set of solutions that cover different regions of the Pareto front, contributing to a stable hypervolume value as the population size increases.

## Simulations with GAP simulator

# Chapter 6. Conclusions and future work

The conclusions of this dissertation project are the following:

* I studied the behavior of the genetic algorithms (NSGAII), as well as the inspired by nature algorithms (SMPSO).
* I created a separate project where the jMetal 6.0 library was integrated.
* The superposition of NSGAII and SMPSO algorithms was integrated in this project. It can be run using the problems implemented in jMetal, but also using a simulator.
* For the purpose of this project, the GAP simulator was used as a problem for superposition algorithm to run on different benchmarks.
* **TODO**: add other conclusions related to the implementation of GAP and results.

Future work that can be implemented in this project:

* Integrate the superposition methods in FADSE framework.
* Analyze the superposition methods with other metrics like TSHS, Spred, Error Ratio, Epsilon, etc.
* Create a more dynamic structure for GAP simulator to execute more benchmarks in the same time and process their results.

# Chapter 7. References

1. https://blog.sourcetreeapp.com/2017/01/27/sourcetree-for-windows-2-0-is-now-in-beta/
2. jMetal\_a\_Java\_framework\_for\_developing\_multi-objec.pdf
3. https://jmetal.sourceforge.net/algorithms.html
4. https://jmetal.readthedocs.io/en/latest/problems.html
5. https://www.google.com/url?sa=i&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FPareto\_front&psig=AOvVaw0VEvt2aL9oiyMcMQDcpth4&ust=1713176872063000&source=images&cd=vfe&opi=89978449&ved=0CBQQjhxqFwoTCLioz6q\_wYUDFQAAAAAdAAAAABAE
6. <https://www.researchgate.net/figure/Followchart-of-the-SMPSO-algorithm_fig3_349661785>
7. [CalH10] H. CALBOREAN, L. VINȚAN, An Automatic Design Space Exploration Framework for Multicore Architecture Optimizations, Proceedings of The 9-th IEEE RoEduNet International Conference, Sibiu, Romania, 2010
8. [CalH13] H. CALBOREAN, R. JAHR, UNGERER T., L. VINȚAN, A Comparison of MultiObjective Algorithms for the Automatic Design Space Exploration of a Superscalar System, Advances in Intelligent Control Systems and Computer Science, Advances in Intelligent Systems and Computing, Vol. 187, pg. 489-502, ISBN 978-3-642-32547-2, ISSN 2194-5357, Springer Berlin Heidelberg.
9. [Coe02] C. A. C. COELLO, D. A. V. VELDHUIZEN, G. B. LAMONT, Evolutionary Algorithms for Solving Multi-Objective Problems, 1st ed. Springer, 2002
10. [JahR12] R. JAHR, H. CALBOREAN, VINȚAN L., T. UNGERER, Finding Near-Perfect Parameters for Hardware and Code Optimizations with Automatic Multi-Objective Design Space Explorations, Concurrency and Computation: Practice and Experience, Print ISSN 1532- 0626, Online ISSN: 1532-0634, John Wiley & Sons, 2012 - disponibil la <http://onlinelibrary.wiley.com/doi/10.1002/cpe.2975/abstract;jsessionid=B1CB2ED1012CEFDF14A3E187DF39D40D.d02t03?deniedAccessCustomisedMessage=&userIsAuthenticated=false>.
11. [Neb09] A. NEBRO, J. DURILLO, J. GARCIA-NIETO, C. A. COELLO, F. LUNA, E. ALBA, Smpso: A new pso-based metaheuristic for multi-objective optimization, Proceedings of the IEEE Symposium Series on Computational Intelligence, pg. 66-73, 2009.
12. <https://www.researchgate.net/profile/Reza-Ebrahimi-Atani/publication/274714829_Multi_Objective_Design_Space_Exploaration_for_Grid_ALU_Processor/links/5624d31908ae93a5c92d2b44/Multi-Objective-Design-Space-Exploaration-for-Grid-ALU-Processor.pdf>
13. <https://www.researchgate.net/publication/220045365_Multiobjective_Optimization_Using_Evolutionary_Algorithms_Wiley_New_York>
14. <https://ro.scribd.com/document/186903190/Multi-Objective-Optimization-of-Advanced-Computer-Architectures-using-Domain-Knowledge>

TODO: write references in the correct way and add reference points in the doc.