**“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](#_Toc167619380)

[1.1. Project scope 4](#_Toc167619381)

[1.2. Project objectives 4](#_Toc167619382)

[1.5. Paper structure 5](#_Toc167619383)

[Chapter 2. Technology stack description 6](#_Toc167619384)

[2.1. IntelliJ IDEA 6](#_Toc167619385)

[2.2. Sourcetree 7](#_Toc167619386)

[2.3. Java language 8](#_Toc167619387)

[2.3.1. Comparison with other programming languages 10](#_Toc167619388)

[2.4. jMetal library 11](#_Toc167619389)

[2.4.1. How to install and use it? 13](#_Toc167619390)

[2.4.2. Algorithm types 16](#_Toc167619391)

[2.4.3. Quality Indicators 17](#_Toc167619392)

[2.4.4. Problems 19](#_Toc167619393)

[Chapter 3. Theoretical part of the implementation 21](#_Toc167619394)

[3.1. Pareto concept 21](#_Toc167619395)

[3.1.1. Multi-objective genetic algorithms 22](#_Toc167619396)

[3.1.1.1. Non-dominated Sorting Genetic Algorithm 23](#_Toc167619397)

[3.1.1.2. Non-dominated Sorting Genetic Algorithm II 24](#_Toc167619398)

[3.2. Particle Swarm Optimization algorithms 26](#_Toc167619399)

[3.2.1. Particle Swarm Optimization algorithm description 27](#_Toc167619400)

[3.2.2. Strength Pareto Multi-objective Particle Swarm Optimization algorithm 29](#_Toc167619401)

[3.3. Algorithm’s comparisons 32](#_Toc167619402)

[Chapter 4. Integration of theoretical notions into practical part of the application 35](#_Toc167619403)

[4.1. Project architecture 35](#_Toc167619404)

[4.2. NSGAII algorithm 35](#_Toc167619405)

[4.3. SMPSO algorithm 36](#_Toc167619406)

[4.4. Superposition of the algorithms 36](#_Toc167619407)

[4.4.1. Superposition structure 38](#_Toc167619408)

[4.4.1.1. AbstractGeneticAlgorithm class 38](#_Toc167619409)

[4.4.1.2. GeneticAlgorithmWrapper class 40](#_Toc167619410)

[4.4.1.3. AbstractSuperPositionGA class 43](#_Toc167619418)

[4.4.1.4. AbstractParticleSwarmOptimization class 46](#_Toc167619419)

[4.4.1.5. ParticleSwarmWrapper class 48](#_Toc167619420)

[4.4.1.6. AbstractSuperPositionPSO 51](#_Toc167619421)

[4.4.1.7. SuperPositionNSGAII class 55](#_Toc167619422)

[4.4.1.8. SuperPositionSMPSO class 57](#_Toc167619423)

[4.4.1.9. SuperPositionCombinator1 class 59](#_Toc167619424)

[4.4.1.10. AbstractSuperPositionGAPSOCombinator1 60](#_Toc167619425)

[4.5. Simulations 63](#_Toc167619426)

[4.5.1. NSGAII algorithm simulation 63](#_Toc167619427)

[4.5.2. SMPSO algorithm simulation 63](#_Toc167619428)

[4.5.3. SuperPosition of NSGAII and SMPSO simulation 63](#_Toc167619429)

[Chapter 5. Results 64](#_Toc167619430)

[Chapter 6. Conclusions 64](#_Toc167619431)

[Chapter 7. References 65](#_Toc167619432)

# Chapter 1. Introduction

Multi-objective optimization is a vital element in many areas, such as engineering, finance, and logistics, in which decision-makers must frequently consider conflicting objectives. Traditional optimization methods fail to deal with such situations effectively since they cannot handle multiple goals simultaneously. In this regard, metaheuristic algorithms such as NSGA-II and SMPSO have been proven to be valuable approaches for tackling multi-objective optimization problems.

# Project scope

The purpose of this project is to investigate the integration of NSGA-II and SMPSO algorithms to use their strengths, so to enhance the performance of multi-objective optimization. By combining the genetic algorithmic approach of NSGA-II with the swarm intelligence mechanism of SMPSO, this project develops a hybrid algorithm capable of efficiently exploring the solution space, maintaining a diverse set of solutions, and providing high-quality Pareto-optimal solutions for complex multi-objective optimization problems.

# 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 benchmark multi-objective optimization problems.
* Evaluate the effectiveness of the algorithms in terms of coverage, hypervolume (HV), Trailing Suction Hopper Dredger (TSHD).
* Investigate the impact of various parameter settings on the performance of the integrated algorithm.
* 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 contributes to the existing knowledge in multi-objective optimization by:

* Proposing an integration framework for combining NSGA-II and SMPSO algorithms.
* Evaluating the performance of the integrated algorithm on benchmark problems.
* Providing insights into the synergy between genetic algorithms and particle swarm optimization techniques 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 performance evaluation 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 installer for your operating system (Windows, macOS, or Linux).
2. Run the Installer: Once the download is complete, run the installer file and follow the on-screen instructions.
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.

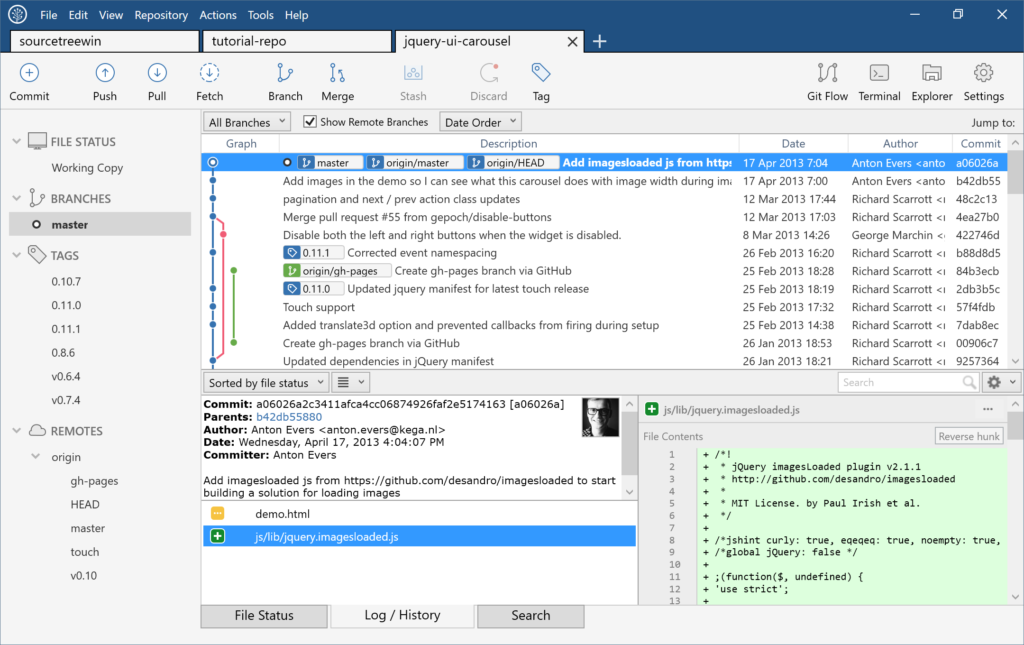
Running a project in IntelliJ IDEA:

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.

<https://blog.sourcetreeapp.com/2017/01/27/sourcetree-for-windows-2-0-is-now-in-beta/>

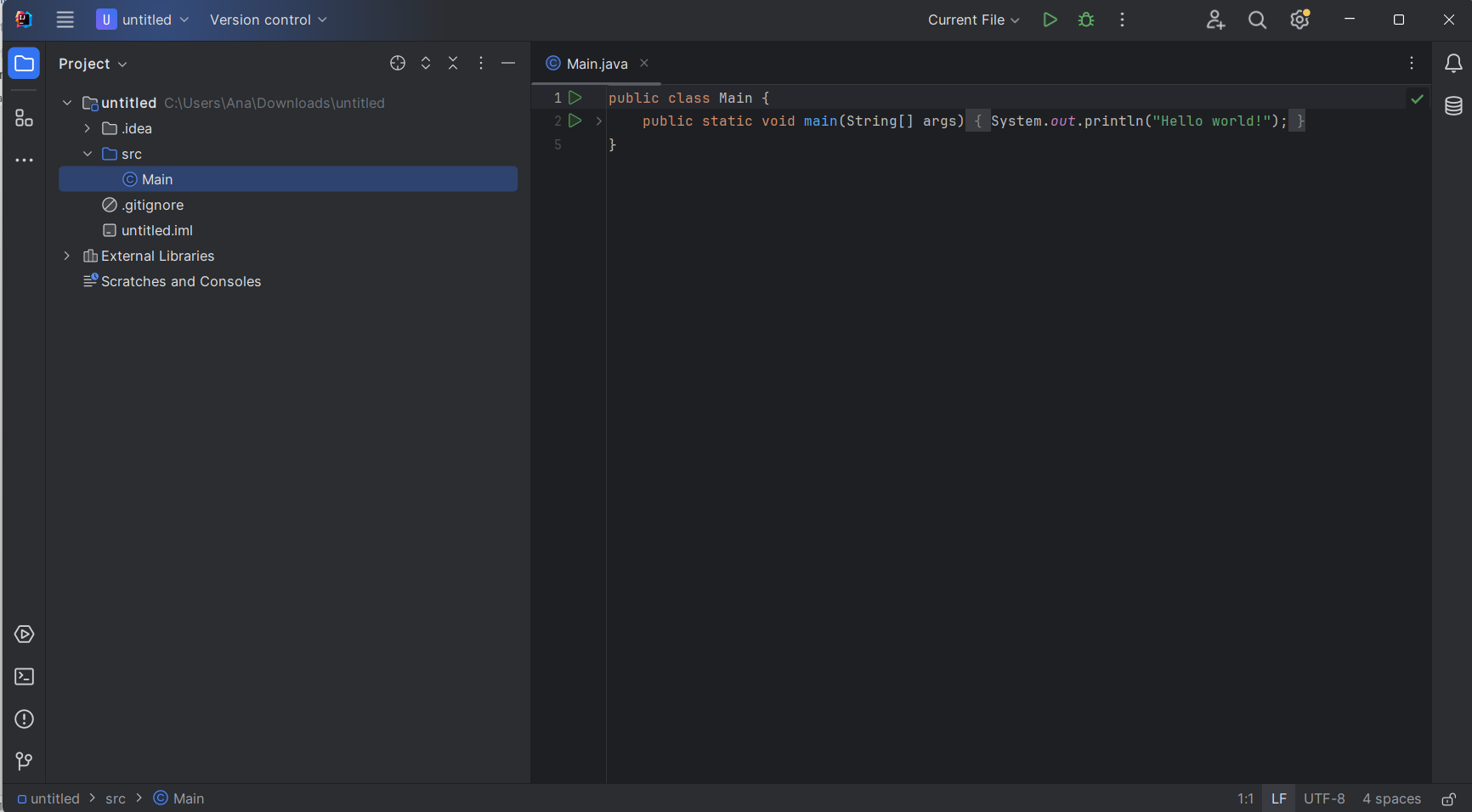
## 2.3. Java language

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 syntax of the language 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:



* *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 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:

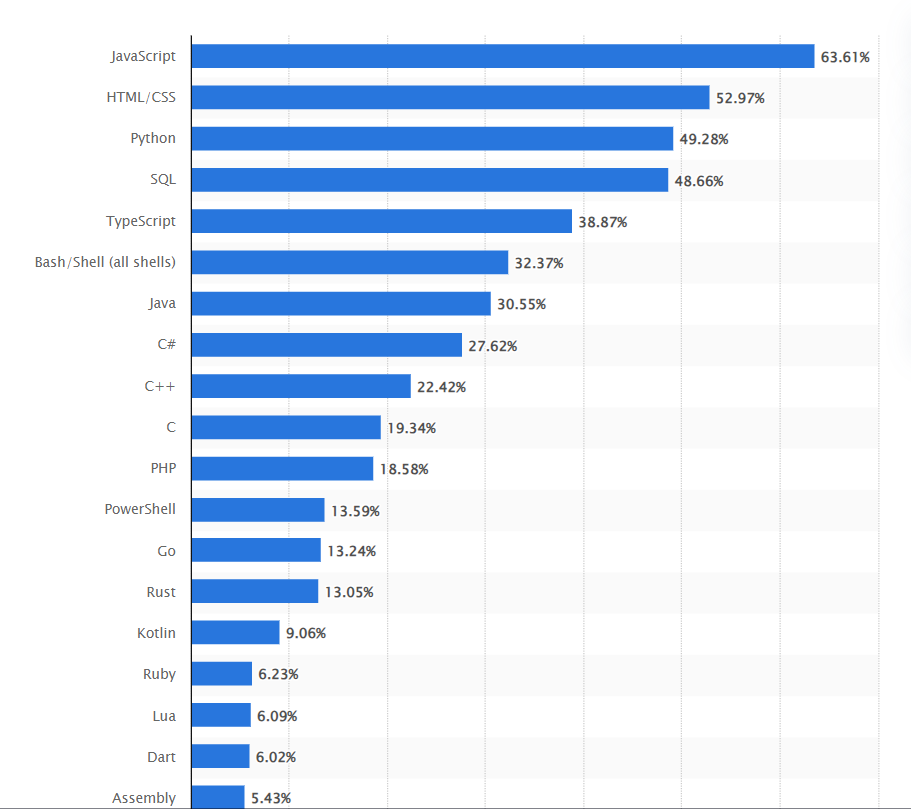


Fig. the most globally used programming languages in 2023

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.
  + Platform Independence: Java code can run on any platform with a Java Virtual Machine (JVM), while C++ code needs to be compiled separately for each target platform.
* 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

<file:///C:/Users/Ana/Downloads/jMetal_a_Java_framework_for_developing_multi-objec.pdf>

jMetal is a Java-based framework designed for multi-objective optimization (MOO) with metaheuristic algorithms. It offers a rich set of tools and functionalities for solving optimization problems involving multiple 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, which can be used for multi-objective optimization.

* 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:

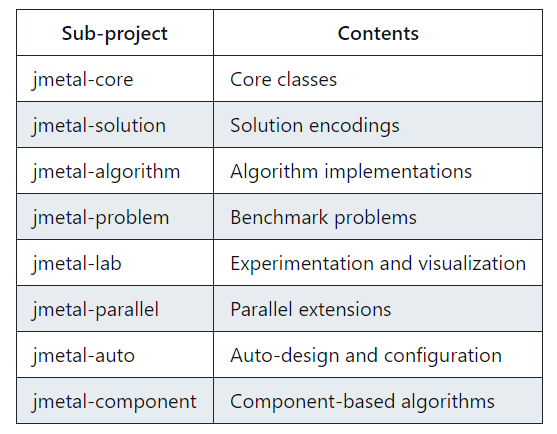


Fig. Project development structure

### 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:

* In the “src” folder of the project, create another folder named “main”
* In the “main” folder, create another folder named “java”
* Copy the downloaded JAR files into this jmetal directory.

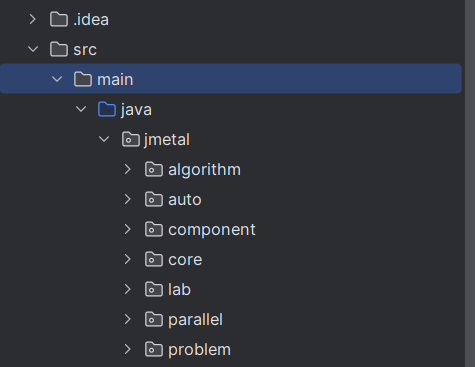


Fig. jMetal library structure

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.

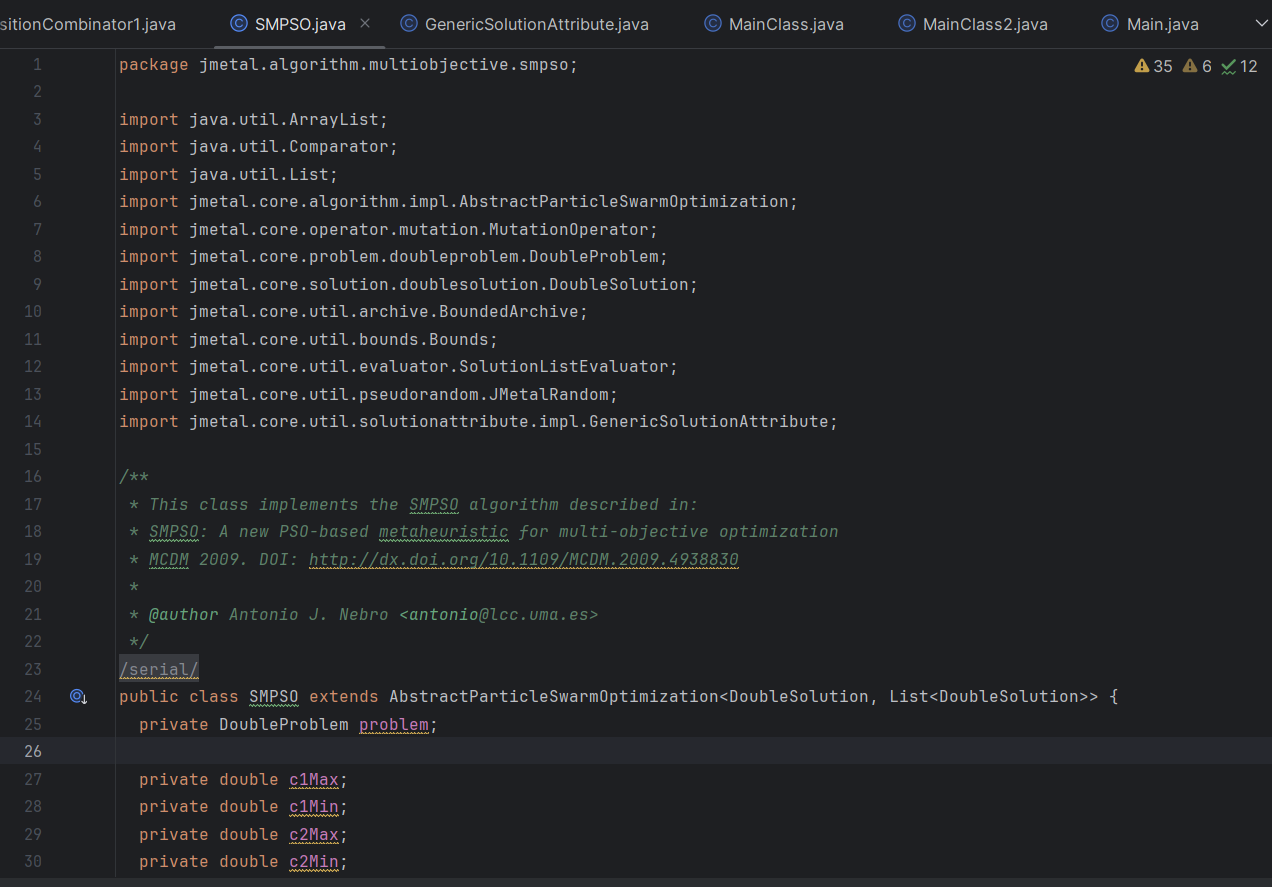


Fig. SMPSO algorithm class

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

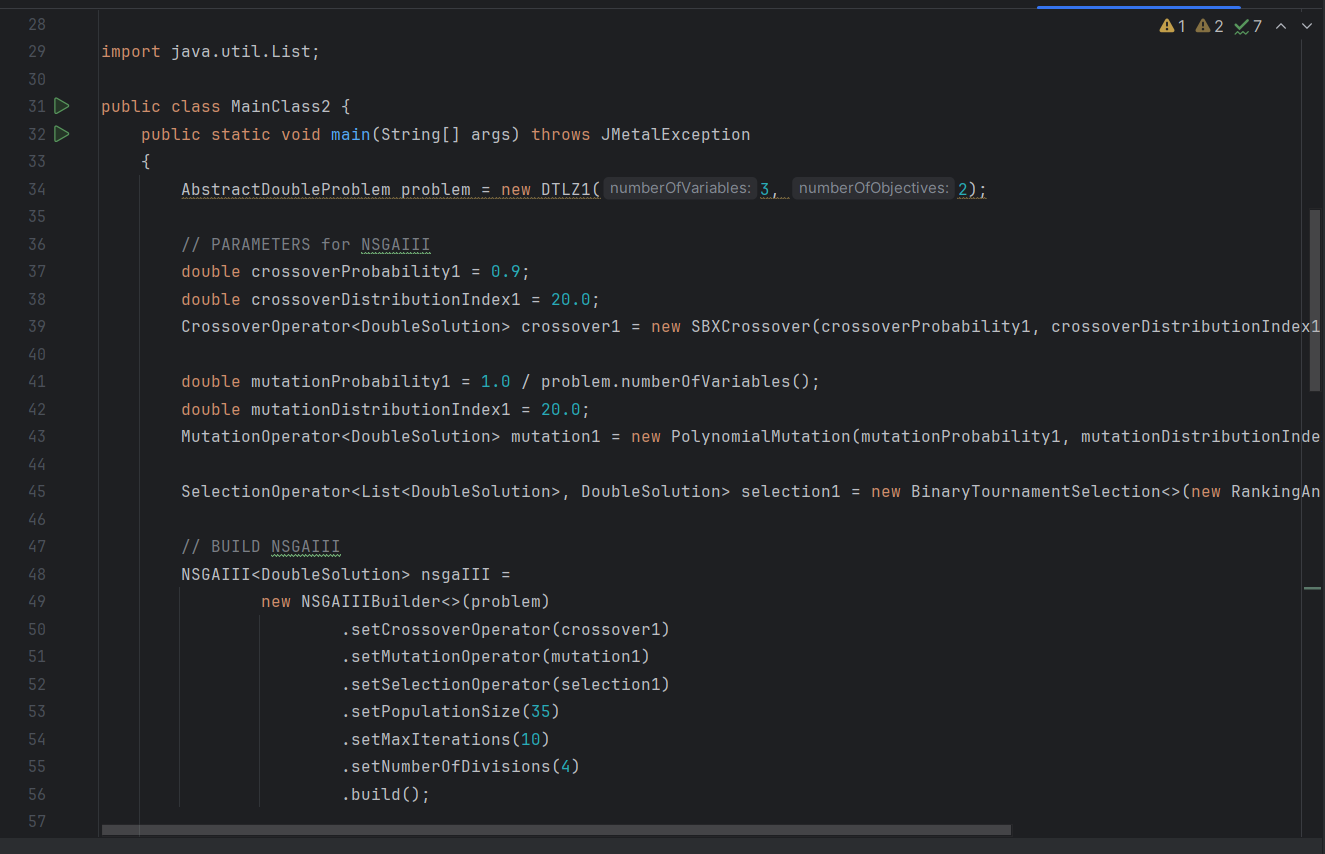


Fig. MainClass.java

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

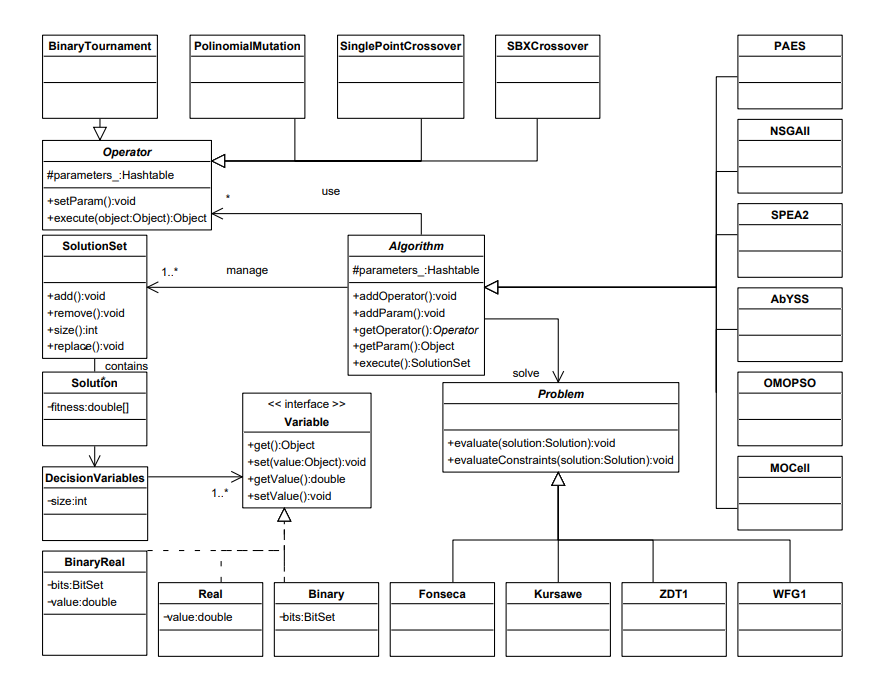


Fig. UML class diagram of jMetal

### 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

<https://jmetal.sourceforge.net/algorithms.html>

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:

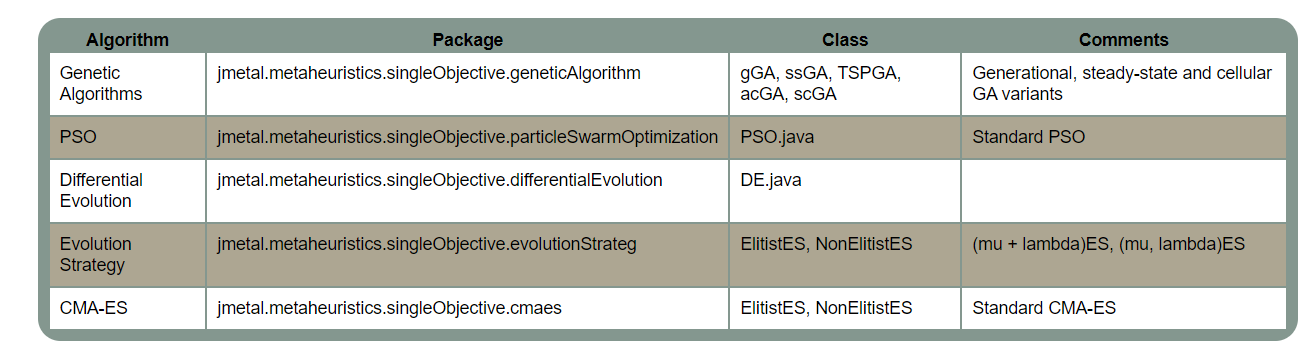


Fig. Single-objective algorithms

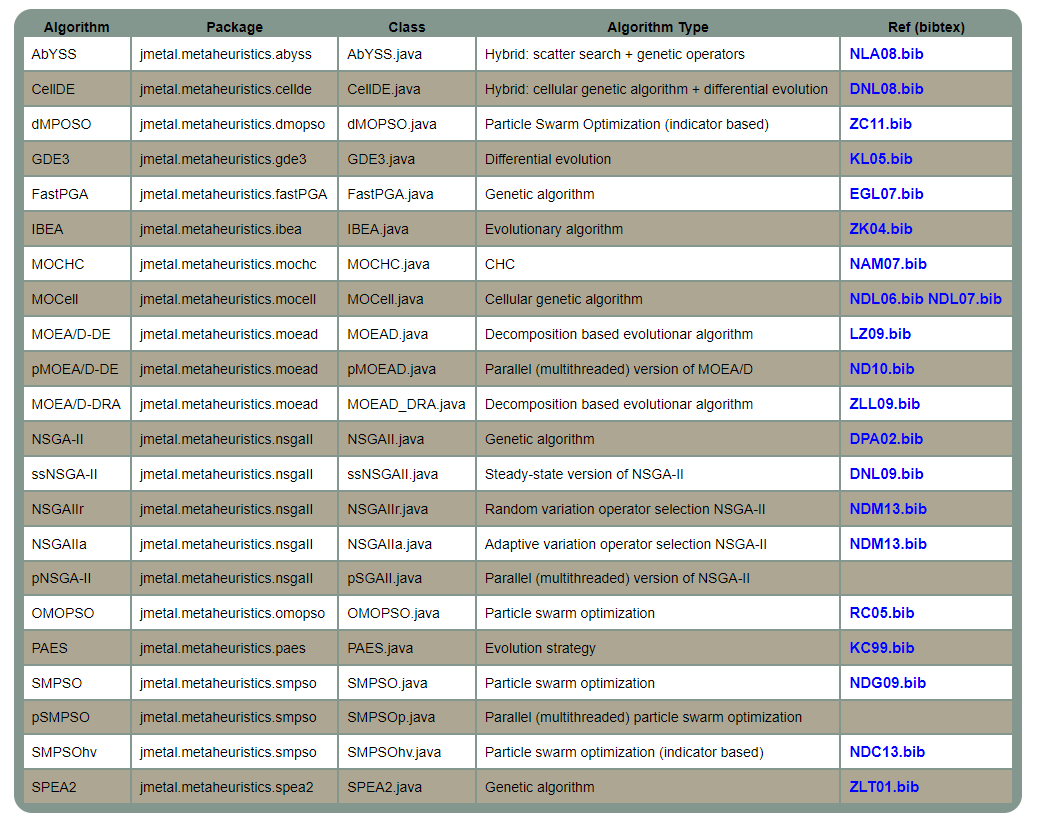


Fig. Multi-objective algorithms

In the context of this paper work, the NSGA II 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 multi-objective optimization algorithms. These indicators help in evaluating various aspects of the solution sets generated by the algorithms, such as convergence, diversity, and overall solution quality relative to the Pareto front.

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

* Generational distance: GD expresses how far the Pareto front found by the algorithm is, on average, from the Real Pareto
* 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 volume in the objective space covered by the Pareto front approximated by the algorithm. A higher hypervolume value indicates a better approximation of the Pareto front.
  + 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.
* Generalized spread: It examines the absolute and relative distribution of solutions across the Pareto front and the extent to which they cover the whole range of the objectives.
* Epsilon: Determines the minimum distance by which the Pareto-optimal solution set has to be shifted in the direction of the reference Pareto front. Lower ratings mean better performance.
* Error ratio: The error rate expresses a measure of the size of the set of individuals that

the algorithm tested classified them as Pareto eighths, and they were not part of the real Pareto front. If the ER is 0, it means that all individuals found by the algorithm tested are in the Pareto front real, and it is the most favorable case. If ER is 1, none of the individuals on the Pareto front found of the algorithm is not in the real Pareto front

* Set coverage: Compares two solution sets by calculating the proportion of solutions in one set that are dominated by or equal to solutions in the other set. It is used to assess the dominance relationship between two sets of solutions.

### 2.4.4. Problems

<https://jmetal.readthedocs.io/en/latest/problems.html>

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 involve optimizing a single objective function. The goal is to find the maximum or minimum value of this function.

Multi-objective Problems: These involve optimizing two or more conflicting objectives simultaneously. The goal is to find a set of solutions that represent the best trade-offs among the objectives, known as the Pareto front.

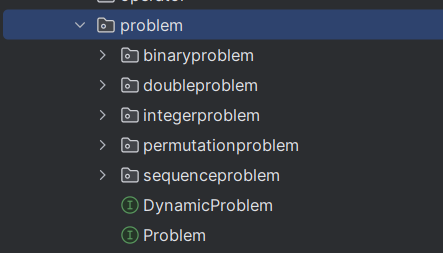


Fig. jMetal problem structure

Examples of problems:

jMetal includes a wide range of benchmark problems used in the optimization community for testing and comparing algorithms. Examples include:

* ZDT (Zitzler, Deb, and Thiele) Series: A set of synthetic problems with varying characteristics, such as the shape of the Pareto front and the distribution of solutions.
* 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:

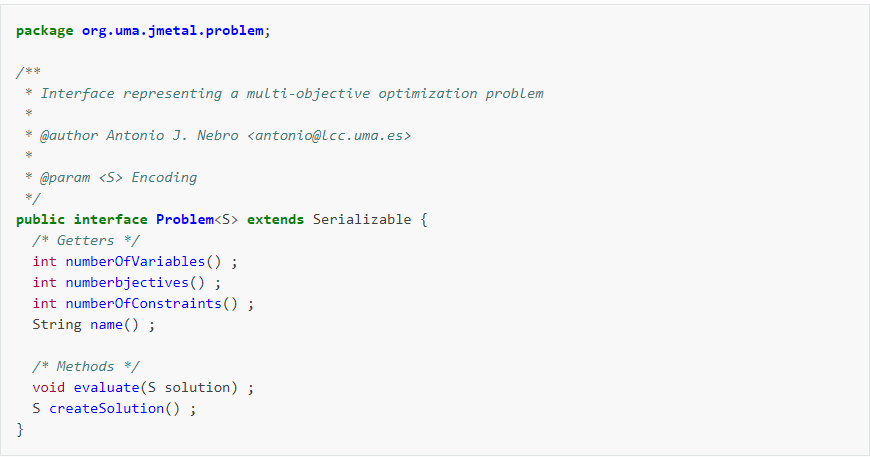


Fig. Problem interface in jMetal

# Chapter 3. Theoretical part of the implementation

## 3.1. Pareto concept

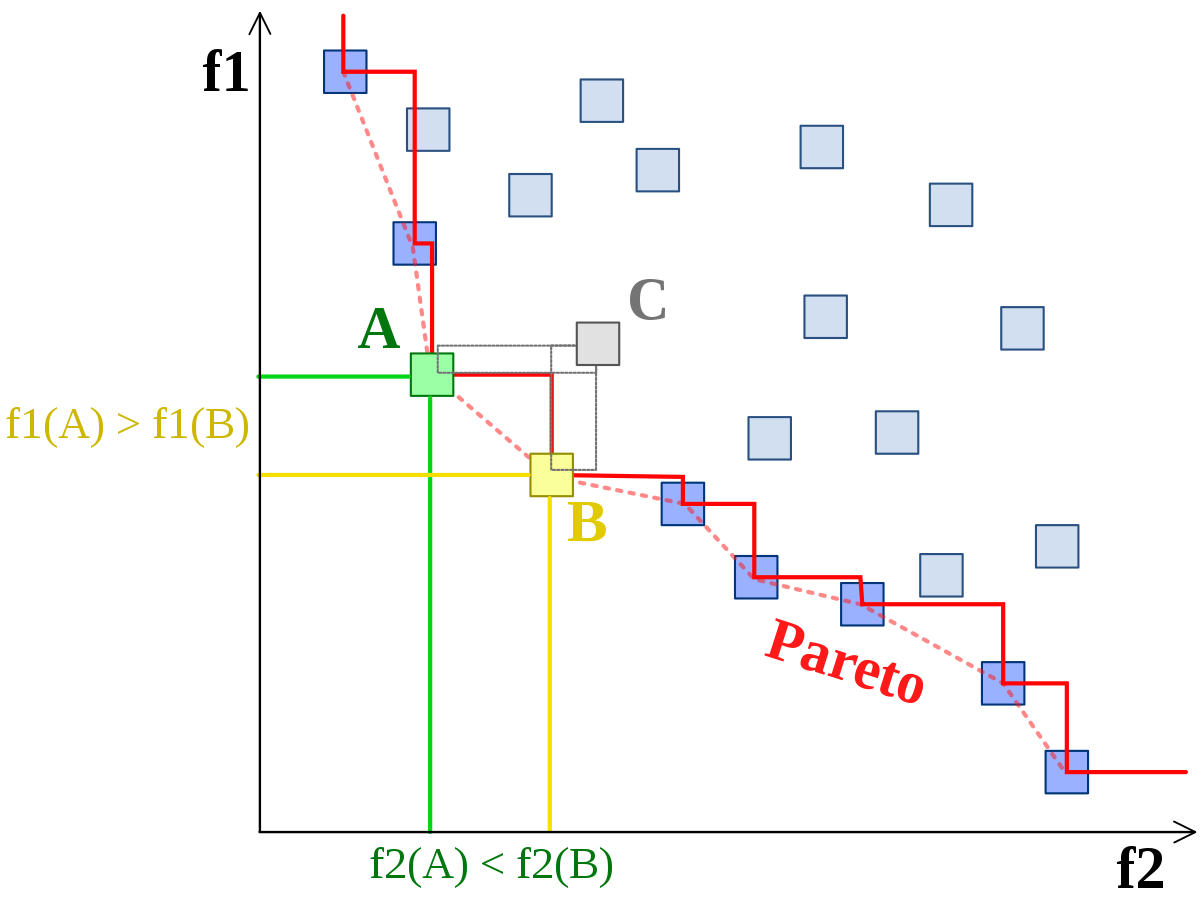
The Pareto concept, named after economist Vilfredo Pareto, is central to multi-objective optimization and is commonly used in economics, engineering, and decision-making scenarios. It is applied in the moment when an individual can’t be better than without disadvantage others. The efficiency of this method is found in the following concepts:

* Pareto optimality: A solution to a multi-objective optimization problem is said to be Pareto optimal if no other solution can improve some objectives without worsening at least one other objective. These solutions are considered efficient or non-dominated.
* Pareto dominance:

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

* + If f1(A) ≤ f1(B) and f2(A) < f2(B), then A dominates B.
  + If f1(A) < f1(B) and f2(A) ≤ f2(B), then A dominates B.
  + If neither A dominates B nor B dominates A, then they are non-dominated.
* Pareto front: In the context of a multi-objective optimization problem, the Pareto front (or Pareto boundary) is the set of all Pareto optimal solutions. It represents the trade-offs between the different conflicting objectives. If any point on the Pareto front is picked and then you try to improve one objective, at least one other objective will be inevitably worst.

Defining the Pareto front through the following graphic:

Fig. Pareto front <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>

### 3.1.1. Multi-objective genetic algorithms

The genetic algorithm typically starts with an initial population of individuals, often randomly generated, and iterates until a termination criterion is satisfied (such as achieving a satisfactory solution or reaching a time limit). Following the evaluation of the initial population, the algorithm enters a main loop. If the termination criterion is not met, the selection phase begins, during which certain individuals are chosen from the entire population to act as parents. These parents then produce offspring through crossover (recombination) and/or mutation operations. The resulting individuals constitute the children population, which is subsequently evaluated. Depending on the specific algorithm, the new parent population may be derived by utilizing the offspring directly as parents or employing other methods to combine the offspring with the previous parent population (e.g., through elitism).

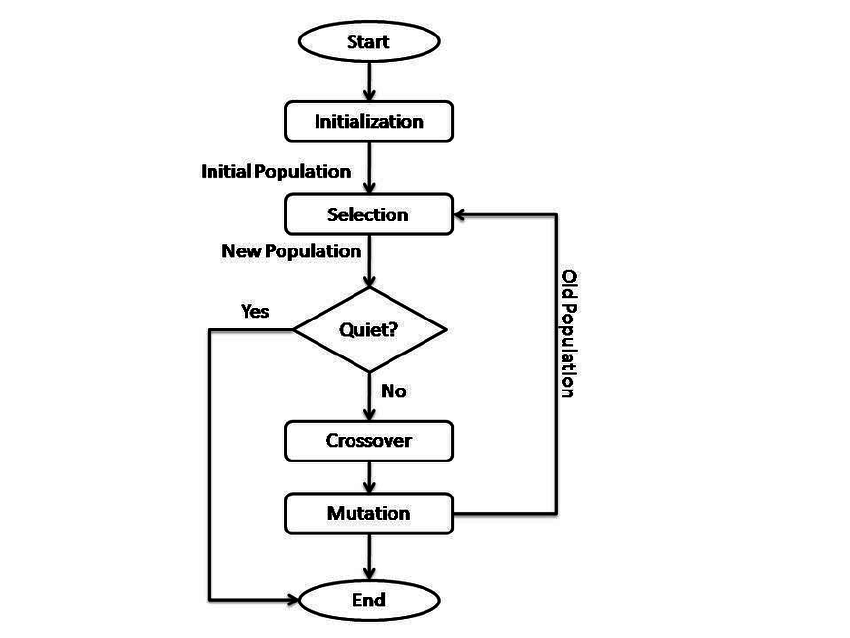


Fig. Genetic algorithm

### 3.1.1.1. Non-dominated Sorting Genetic Algorithm

The Non-dominated Sorting Genetic Algorithm (NSGA) is a multi-objective optimization algorithm inspired by the principles of natural selection and evolution. Developed to address problems with multiple conflicting objectives, NSGA is widely used in engineering, economics, and other fields where decision-making involves balancing multiple criteria.

At its core, NSGA maintains a population of candidate solutions, known as individuals or chromosomes, each representing a potential solution to the optimization problem. Unlike traditional single-objective genetic algorithms, where solutions are evaluated based on a single objective function, NSGA evaluates solutions based on multiple objectives simultaneously. This allows NSGA to identify a set of solutions that are not dominated by any other solution in terms of all objectives, known as the Pareto-optimal front.

The key innovation of NSGA lies in its approach to selection and reproduction. Traditional genetic algorithms typically use fitness proportionate selection, where individuals are selected for reproduction based on their relative fitness. In contrast, NSGA employs non-dominated sorting to rank individuals based on their dominance relationships with other individuals in the population. This sorting process divides the population into several fronts, with the first front containing non-dominated solutions that are not dominated by any other solution in the population.

After sorting, NSGA uses a combination of elitist selection and diversity preservation mechanisms to create the next generation of individuals. Elitist selection ensures that the best solutions from the current population are preserved in the next generation, while diversity preservation mechanisms, such as crowding distance, encourage exploration of the solution space and prevent premature convergence to suboptimal solutions.

The reproduction phase of NSGA involves generating offspring by applying genetic operators, such as crossover and mutation, to selected parent individuals. These genetic operators facilitate the exploration and exploitation of the solution space, allowing NSGA to generate diverse and high-quality solutions over successive generations.

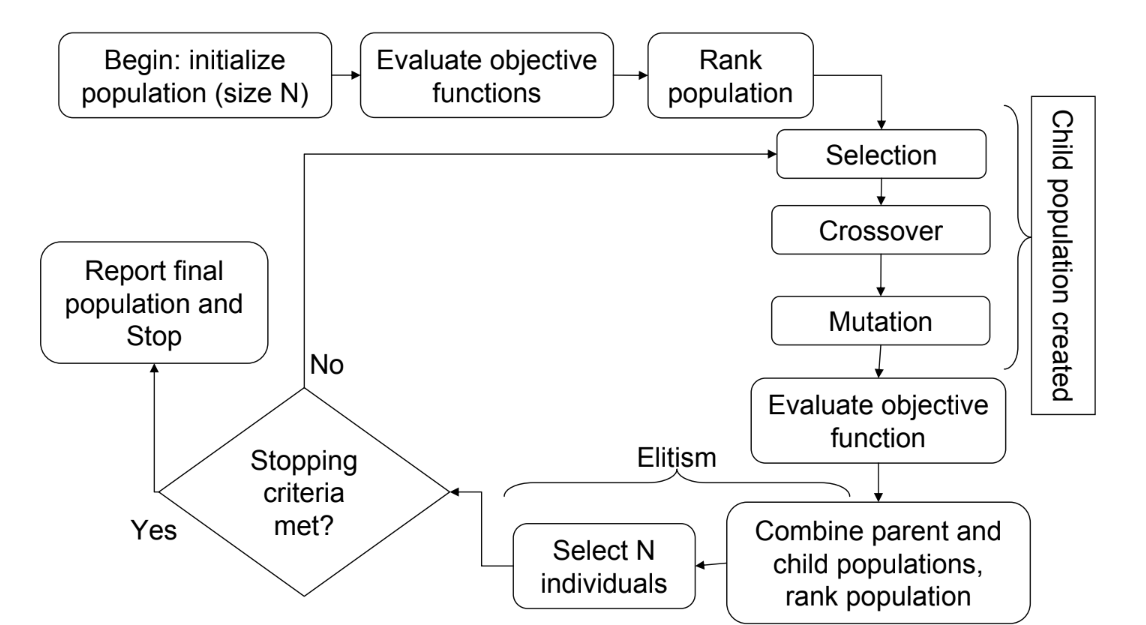
One of the key advantages of NSGA is its ability to identify a diverse set of Pareto-optimal solutions, providing decision-makers with a range of trade-off options to choose from. By representing the trade-offs between conflicting objectives, NSGA enables decision-makers to make informed decisions that balance competing priorities and objectives.

By combining the principles of natural selection with innovative selection and reproduction mechanisms, NSGA is capable of efficiently identifying a diverse set of Pareto-optimal solutions, enabling decision-makers to navigate complex decision spaces and make informed decisions.

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

NSGA-II (Nondominated Sorting Genetic Algorithm II) is also genetic algorithm and it belongs to the family of evolutionary algorithms, specifically designed for solving multi-objective optimization problems. NSGA-II is an extension of the original NSGA (Nondominated Sorting Genetic Algorithm), incorporating improvements to enhance its performance and convergence properties.

As a genetic algorithm, NSGA-II operates based on principles inspired by natural selection and genetics. It maintains a population of candidate solutions (often referred to as individuals or chromosomes) and iteratively evolves this population over multiple generations to search for solutions that represent the Pareto-optimal front, where no solution can be improved in one objective without sacrificing performance in another.

Fig. NSGA II algorithm process

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

Step 1: Calculation of Dominance Counter

For each individual in the population, the dominance counter is computed. This counter indicates the number of individuals in the population that dominate the current individual.

Step 2: Identification of Pareto Fronts

After computing the dominance counters in Step 1, all individuals belonging to the first Pareto front (Front I) will have a dominance counter of 0, as they are not dominated by any other individual in the population. For each individual in Front I, the algorithm identifies the set of individuals dominated by that individual. The dominance counter of each dominated individual is decreased by 1. This process essentially involves identifying the individuals who are dominated by those in Front I and reducing their dominance counters accordingly. After this procedure is completed, the individuals with a dominance counter equal to 0 will form the second Pareto front (Front II).

Iterative Process for All Fronts

This process of identifying Pareto fronts is repeated iteratively for each subsequent front. For each front, the algorithm identifies the individuals with a dominance counter equal to 0, and these individuals form the next Pareto front. The dominance counters of individuals dominated by those in the current front are decremented, and the process continues until all individuals in the population have been assigned to a Pareto front.

Termination of the Process

The process continues until there are no individuals left in the population to be assigned to a 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 addresses some limitations of the original NSGA, such as its computational complexity and tendency to generate too many solutions in densely populated regions of the Pareto front. It achieves better convergence and diversity maintenance, making it a popular choice for solving multi-objective optimization problems in various domains.

This algorithm has a computational complexity of O(MN2), where M is the number of objectives, and N is the number of individuals in the population. This modification decreases the computational complexity, but the memory consumed increases from O(N) in NSGA to O(N2) in NSGA-II.

## 3.2. Particle Swarm Optimization algorithms

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

In PSO, a population of potential solutions, called particles, navigates the search space by adjusting their positions based on their own experience and the experience of their neighbors. Each particle's movement is guided by two main components:

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

Through iterative updates, particles explore the search space while gradually converging towards optimal solutions. PSO algorithms are known for their simplicity, efficiency, and ability to handle high-dimensional and non-linear optimization problems. They have been successfully applied in various domains, including engineering, finance, and machine learning, where they excel in finding near-optimal solutions for complex optimization tasks.

### 3.2.1. Particle Swarm Optimization algorithm description

Particle Swarm Optimization (PSO) is a population-based metaheuristic optimization algorithm inspired by the social behavior of organisms such as birds flocking or fish schooling. Developed by Kennedy and Eberhart in 1995, PSO has gained popularity due to its simplicity, efficiency, and effectiveness in solving a wide range of optimization problems.

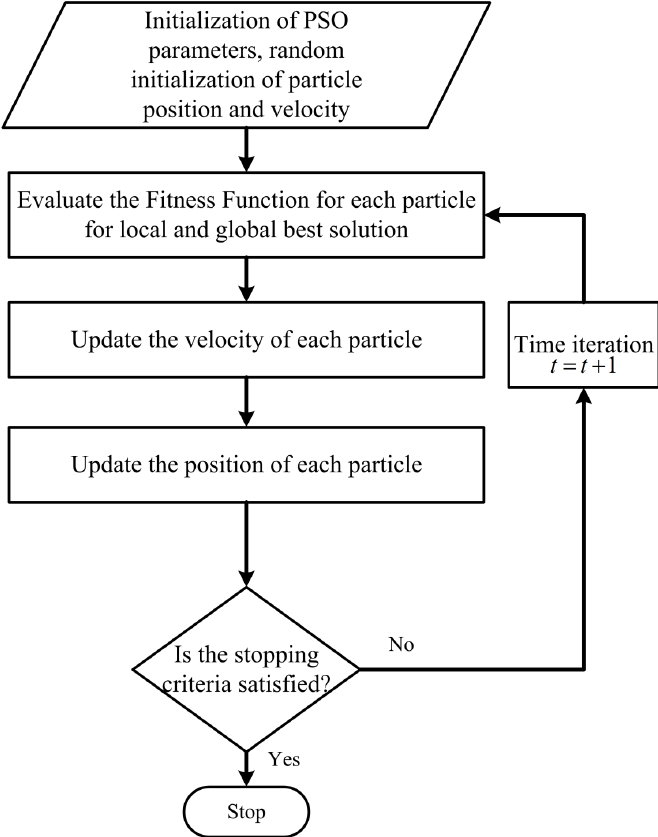


Fig. PSO algorithm process

Initialization

The PSO algorithm begins by initializing a population of particles, each representing a potential solution to the optimization problem. The particles are randomly distributed across the search space, and each particle is assigned a random velocity vector.

Fitness Evaluation

Once initialized, the fitness of each particle is evaluated based on an objective function that quantifies the quality of the solution represented by the particle. This objective function is problem-specific and reflects the optimization goals of the problem being solved.

Initialization of Particle's Memory

Each particle maintains two memory positions: its personal best position (Pbest) and the global best position (Gbest). Initially, these memory positions are set to the particle's current position.

Particle Movement

In each iteration, or generation, of the algorithm, particles adjust their positions and velocities based on their current position, personal best position, and global best position. The movement of each particle is guided by two main components:

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 updating their positions, each particle compares its current position with its personal best position (Pbest). If the new position improves the particle's personal best solution, the Pbest is updated accordingly. Additionally, each particle compares its personal best position with the global best position (Gbest) found by any particle in the swarm. If the personal best position of a particle is better than the current global best position, the Gbest is updated to reflect this improvement.

Termination Criterion

The PSO algorithm continues iterating through generations until a termination criterion is met. Common termination criteria include reaching a maximum number of iterations, finding a satisfactory solution, or reaching a predefined computational limit.

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

SMPSO algorithm is an extension of the classical Particle Swarm Optimization (PSO) algorithm designed specifically for solving multi-objective optimization problems. Introduced by Durillo and Nebro in 2009, SMPSO enhances the traditional PSO approach by incorporating Pareto dominance and diversity preservation mechanisms.

In SMPSO, particles navigate the search space by adjusting their positions based on their personal best and global best positions, while also considering the Pareto dominance relationships among solutions. By maintaining a diverse set of non-dominated solutions and encouraging exploration of the entire Pareto front, SMPSO efficiently identifies a range of trade-off solutions that represent the optimal compromise between conflicting objectives.

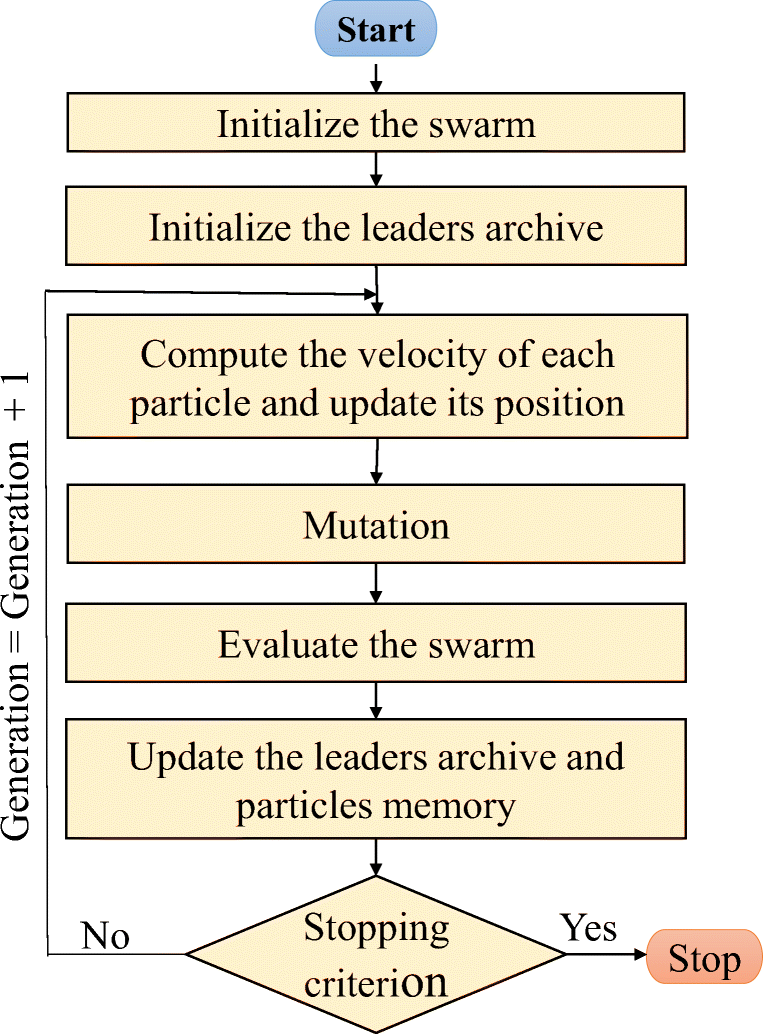


Fig. SMPSO algorithm process <https://www.researchgate.net/figure/Followchart-of-the-SMPSO-algorithm_fig3_349661785>

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

Initialization:

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

Fitness Evaluation:

Unlike PSO, where fitness is evaluated based on a single objective function, SMPSO evaluates the fitness of each particle based on multiple objective functions simultaneously. This allows SMPSO to handle multi-objective optimization problems where solutions need to be optimized with respect to conflicting objectives.

Pareto Dominance:

In SMPSO, particles maintain a set of personal best solutions (Pbest) and a global best solution (Gbest). However, unlike PSO, where solutions are compared based on a single fitness value, SMPSO employs Pareto dominance to compare solutions. A solution A is said to dominate solution B if it is equal or better than B in at least one objective and strictly better in at least one objective. SMPSO ensures that the population contains only non-dominated solutions, known as the Pareto front, by maintaining dominance relationships among solutions.

Velocity and Position Update:

The velocity and position update equations in SMPSO are similar to PSO, with modifications to incorporate Pareto dominance and diversity preservation mechanisms. Particles adjust their velocities based on their personal best and global best solutions, as well as the social interaction with other particles in the swarm. However, in SMPSO, the velocity update also considers the Pareto dominance relationships among solutions to guide the search towards non-dominated regions of the objective space.

Diversity Preservation:

One of the key differences between SMPSO and classic PSO is the inclusion of diversity preservation mechanisms in SMPSO. These mechanisms encourage exploration of the entire Pareto front by promoting diversity among solutions. Techniques such as crowding distance or grid-based diversity measures are often employed to maintain a diverse set of non-dominated solutions in the population.

Termination Criterion:

The termination criterion for SMPSO is typically similar to PSO, where the algorithm stops iterating when a predefined termination condition is met. This condition could include 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 can handle problems with multiple conflicting objectives more effectively by maintaining a diverse set of non-dominated solutions along the Pareto front. Also, SMPSO promotes exploration of the entire Pareto front by incorporating diversity preservation mechanisms, whereas PSO may struggle to adequately explore the objective space in multi-objective scenarios.

As an overall 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.

Let A be the set of individuals from the Pareto front of the first algorithm, and B be 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* such 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 means that half of the 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).

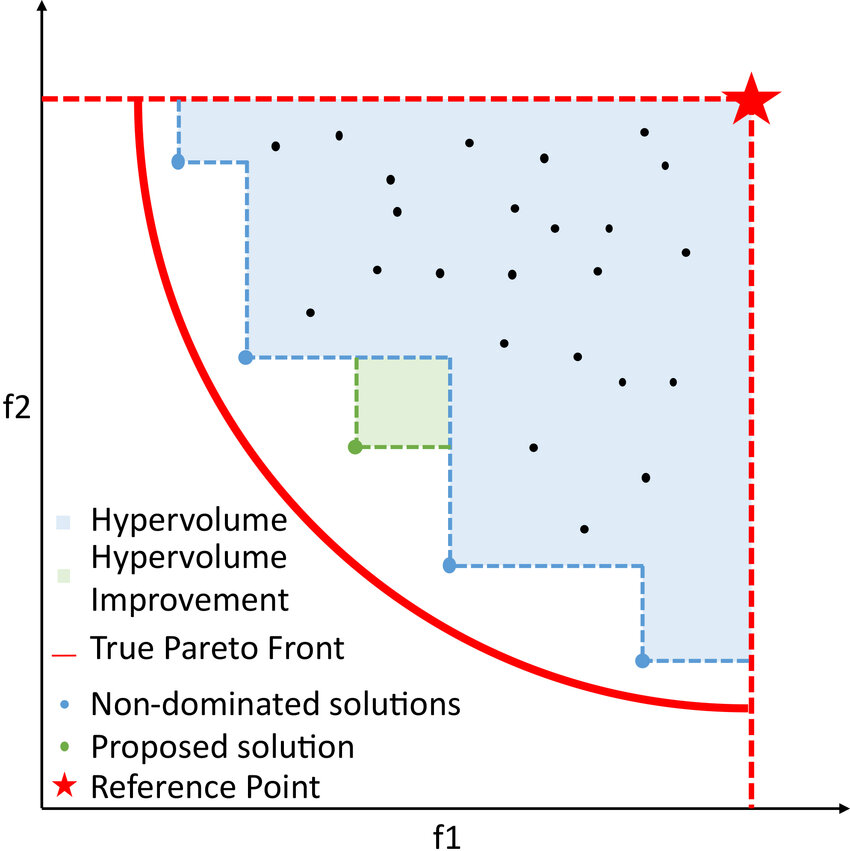
* Hypervolume: calculates the volume of the objective space enclosed by a reference point and the Pareto front, where the reference point represents the worst possible solution in each objective.

Fig. Hypervolume of a convex minimization problem

* TSHD: is a performance measure used in multi-objective optimization to quantify the difference in hypervolume between two sets of solutions.

It provides a comparative assessment of the quality of two sets of solutions in terms of their coverage of the objective space. The TSHD metric is calculated by subtracting the hypervolume of one set of solutions from the hypervolume of another set of solutions. The formula for the TSHD metric is:

TSHD(A, B) = HV(A) - HV(B), where:

* A and B are two sets of solutions.
* HV(A) is the hypervolume of set A .
* HV(B) is the hypervolume of set B.

A positive TSHD value indicates that set A covers more volume in the objective space than set B, while a negative value indicates the opposite. The TSHD metric is commonly used to compare the performance of different algorithms or to assess the improvement of a new algorithm over an existing one.

# 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:

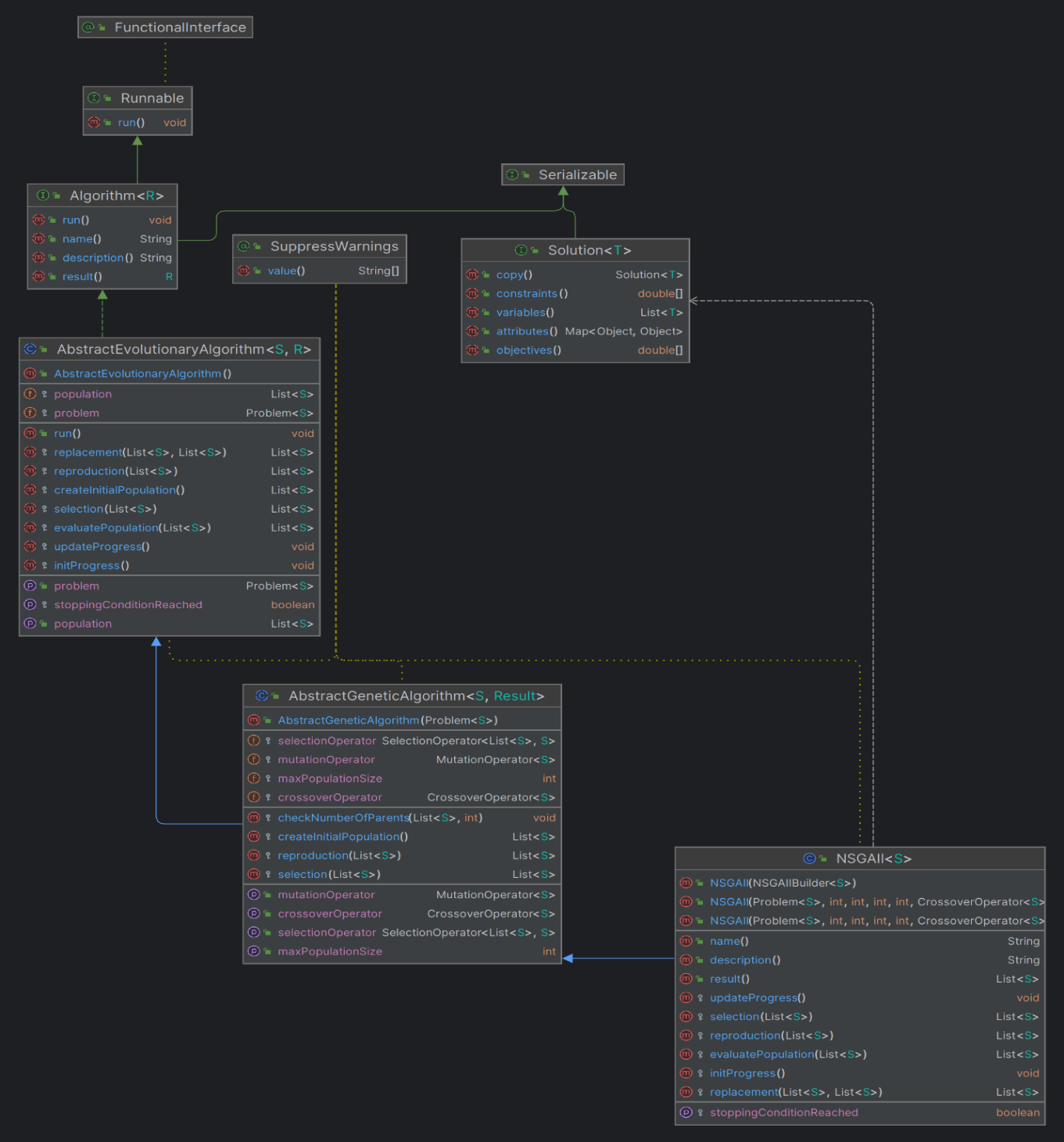


Fig. NSGAII UML diagram

## SMPSO algorithm

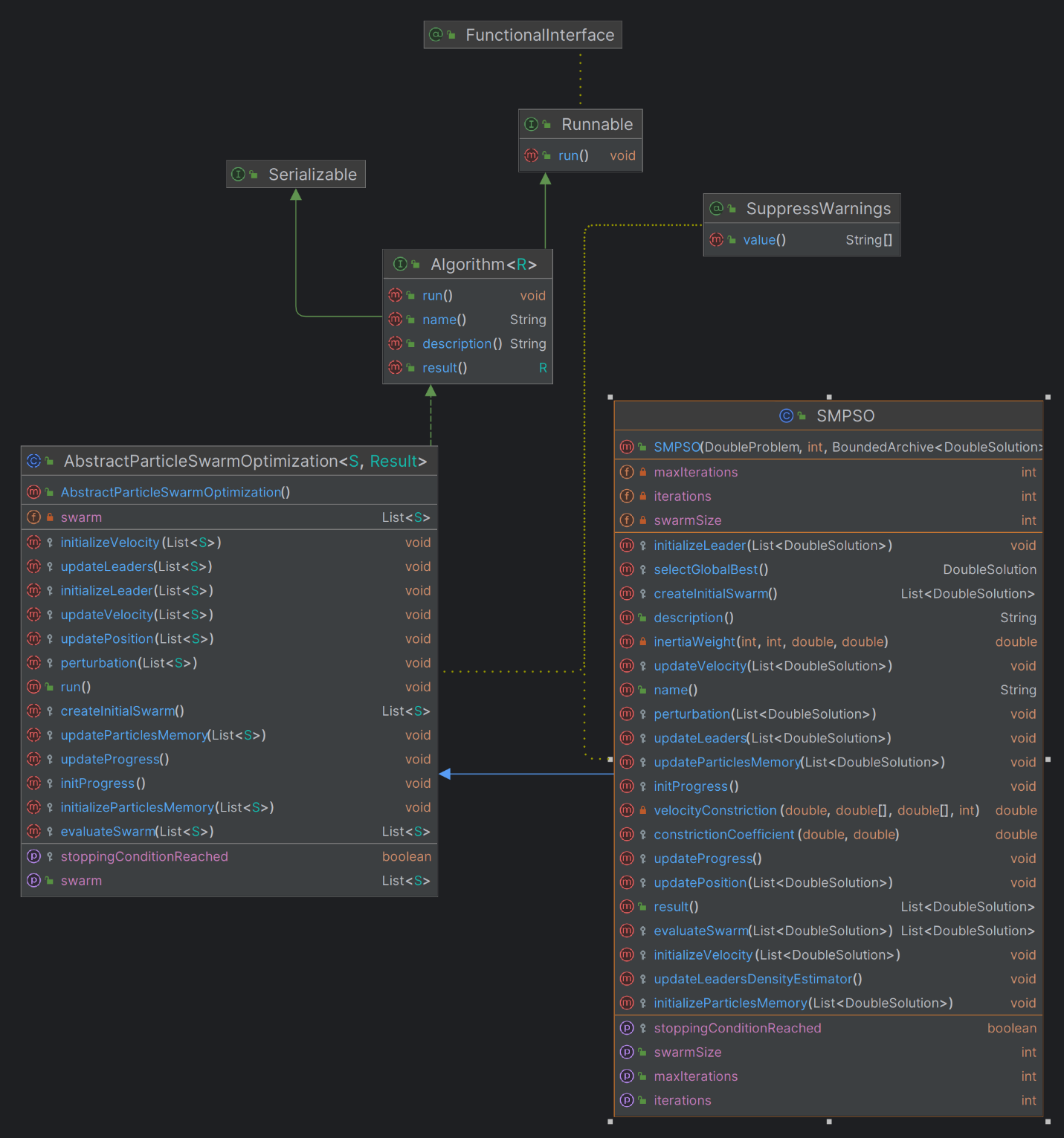
The SMPSO algorithm was developed through jMetal library, which was integrated in the project. The algorithm described at chapter 3.2.2 has the following diagram representation:

Fig. SMPSO UML diagram

## Superposition of the algorithms

Multi-objective evolutionary optimization (MOEO) is a powerful technique for solving optimization problems with multiple conflicting objectives.

In MOEO, a population of solutions is evolved over time, where each solution represents a trade-off between the different objectives.

The goal of MOEO is to find a set of solutions that are as close as possible to the Pareto-optimal front, which is the set of solutions that are not dominated by any other solution. So, the challenge that appears here is that it can be difficult to find a balance between convergence speed and diversity. To solve this new challenge, here comes the ideas of superposition of populations, a new approach to MOEO.

A few advantages of the superposition are the following ones:

* Improved diversity:
* Multi-objective algorithms aim to maintain a diverse set of solutions in the population to cover the Pareto front comprehensively. NSGA-II achieves diversity through the non-dominated sorting and crowding distance mechanism, which encourages spread among solutions.
* SMPSO utilizes a diversity maintenance mechanism based on the crowding distance and a velocity clamping technique, ensuring that the particles explore different regions of the search space.
* Enhances 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. NSGA-II and SMPSO are capable of scaling to problems with a large number of decision variables by exploring the search space efficiently.
* 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 properties make multi-objective algorithms like NSGA-II and SMPSO well-suited for solving complex optimization problems with multiple conflicting objectives. 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:

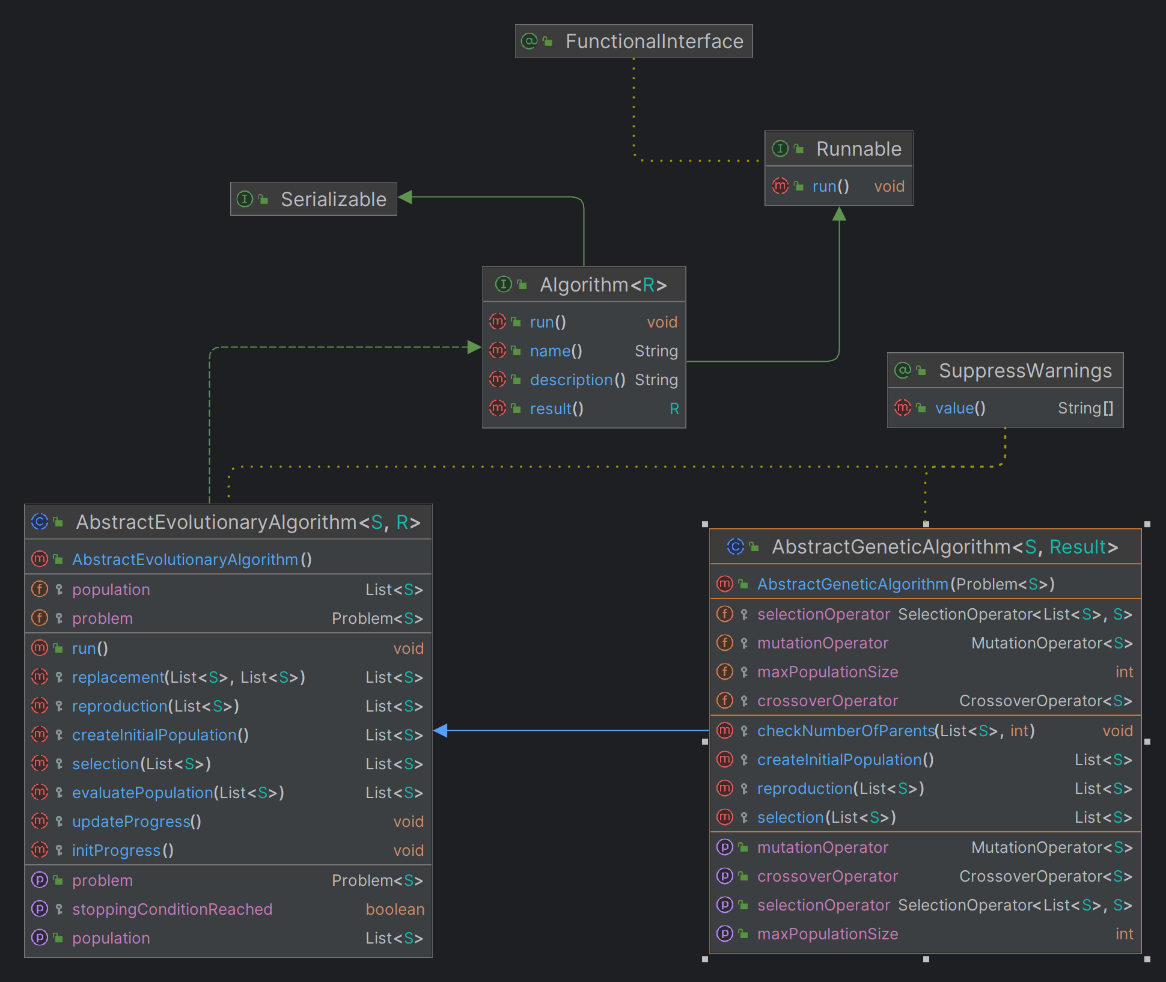


Fig. AbstractGeneticAlgorithm UML diagram

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 iterations or generations the algorithm will run.
   * **currentIteration**: The current iteration or generation number.
   * **evaluations**: The number of evaluations performed by the algorithm (evaluations of 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()**: Evaluates the objective functions for each solution in 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:

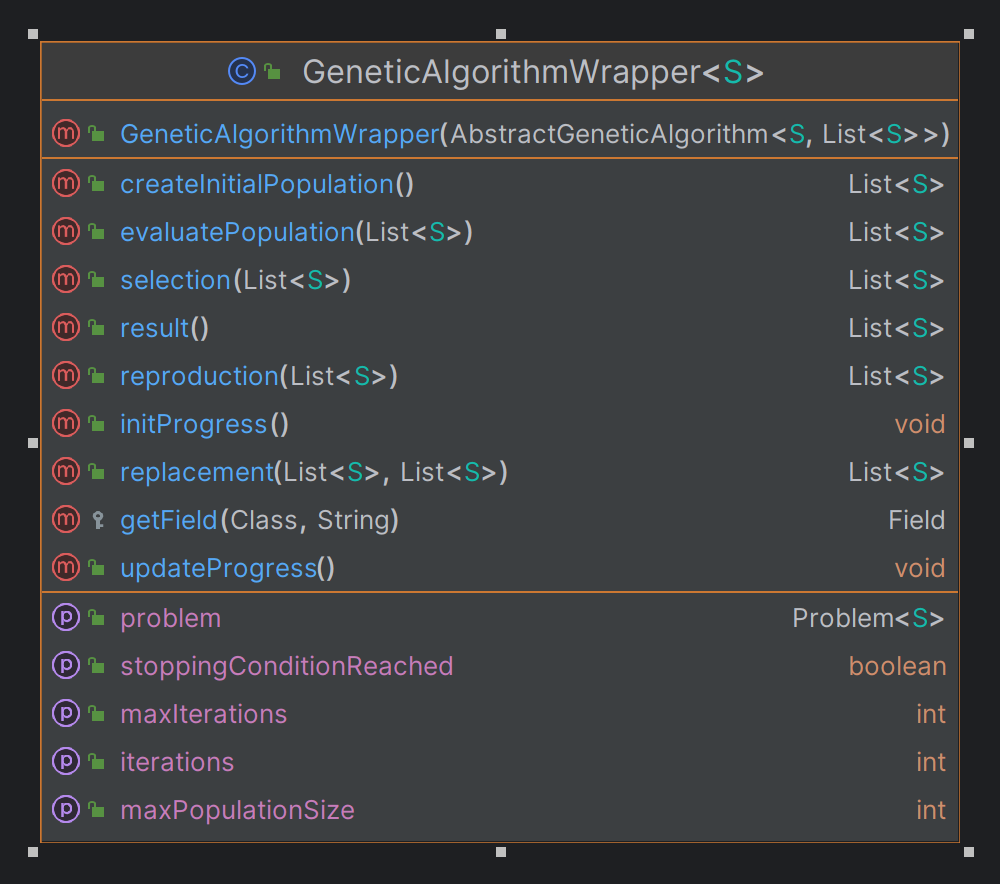


Fig. GeneticAlgorithmWrapper UML diagram



### 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:

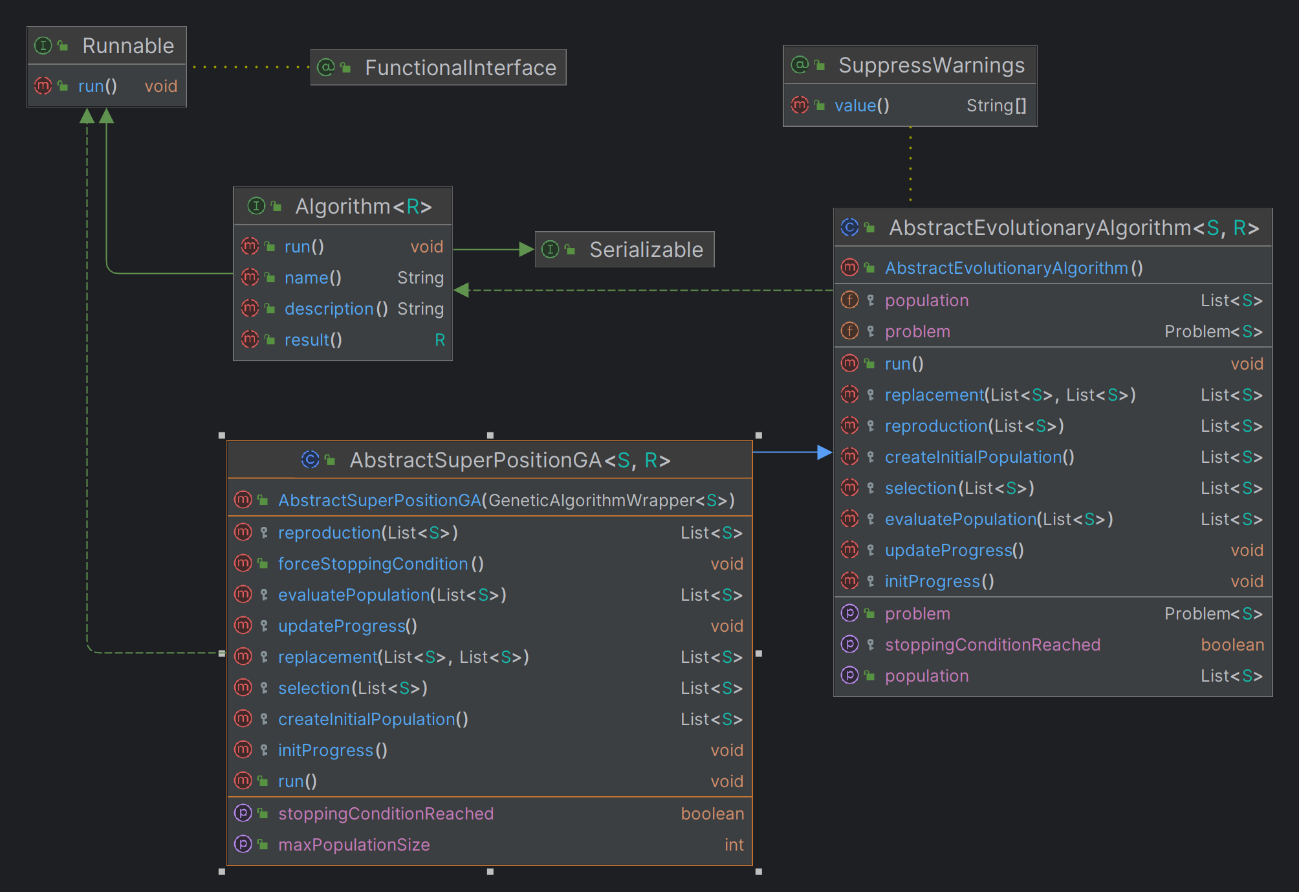


Fig. AbstractSuperPositionGA UML diagram

### 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 swarm of particles, each representing a candidate solution in the search space.
   * **maxIterations**: The maximum number of iterations or generations the algorithm will run.
   * **currentIteration**: The current iteration or generation number.
   * **evaluations**: The number of evaluations performed by the algorithm (evaluations of 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.
   * **updateLeader()**: Updates the global best solution found by the swarm based on the current particles' best solutions.
   * **updateLeaders()**: Updates the collection of historical global best solutions found during the optimization process.
   * **updateParticles()**: Updates the velocity and position of each particle in the swarm based on the PSO equations.
   * **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).
   * **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.

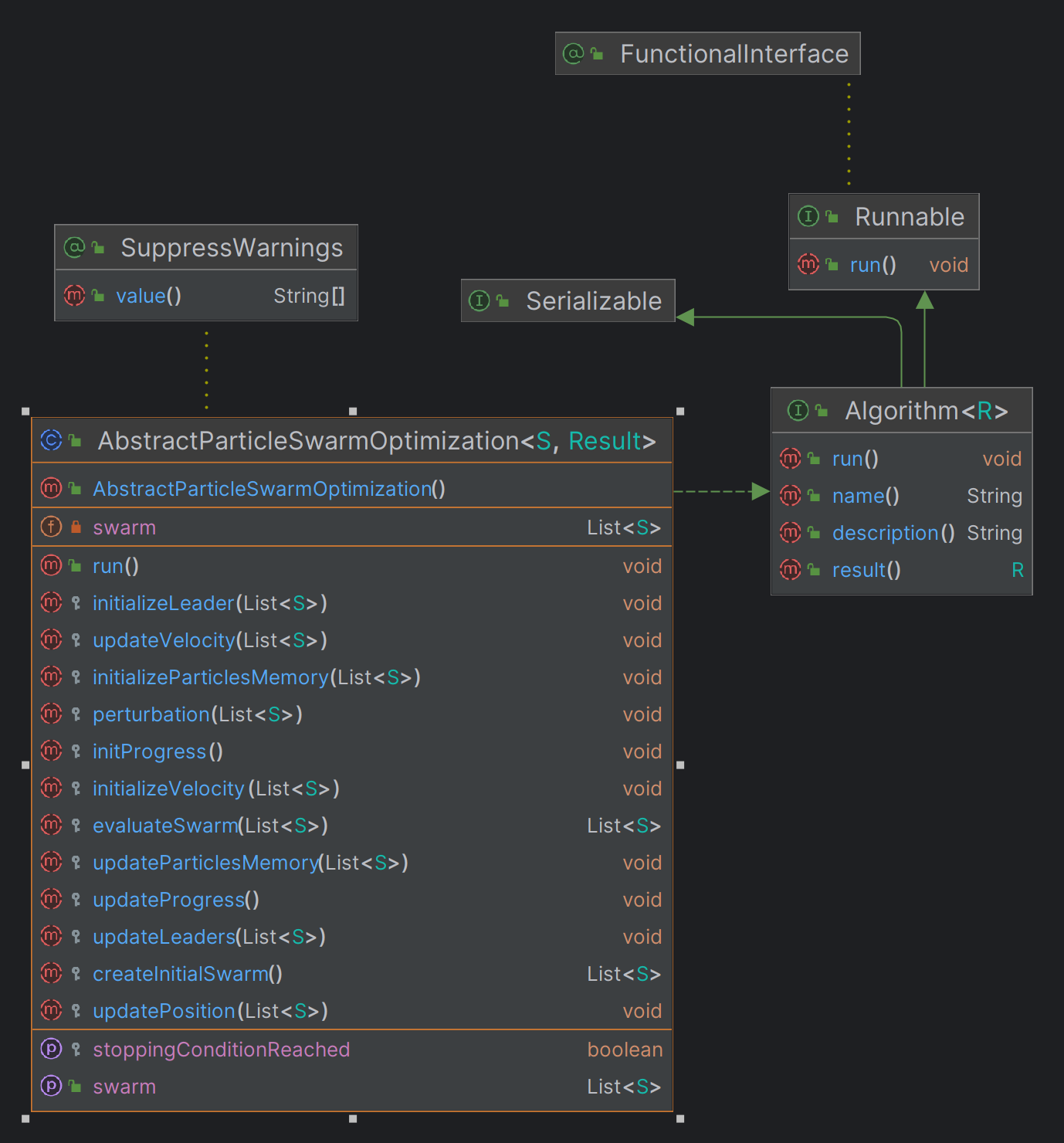


Fig. AbstractParticleSwarmOptimization UML class

### 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:

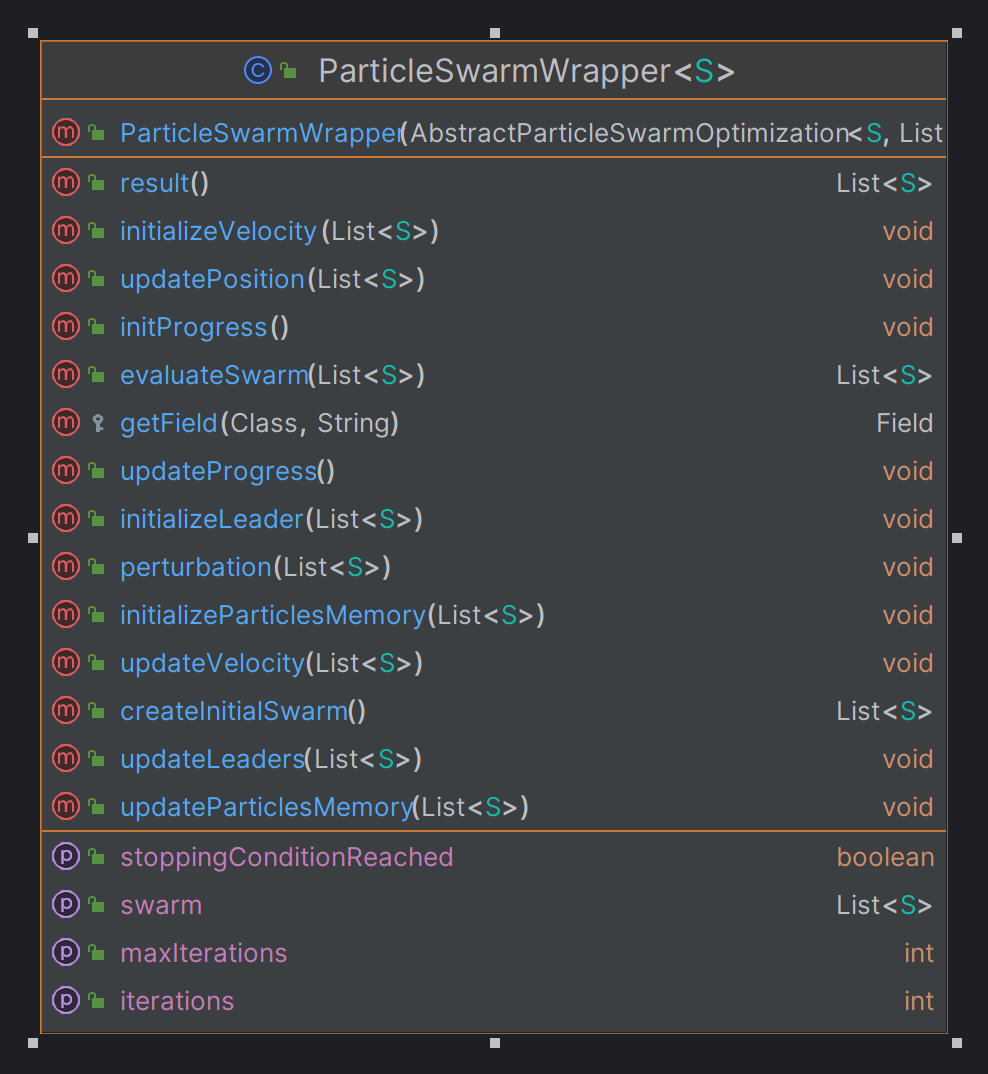


Fig. ParticleSwarmWrapper UML diagram

### 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. 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:

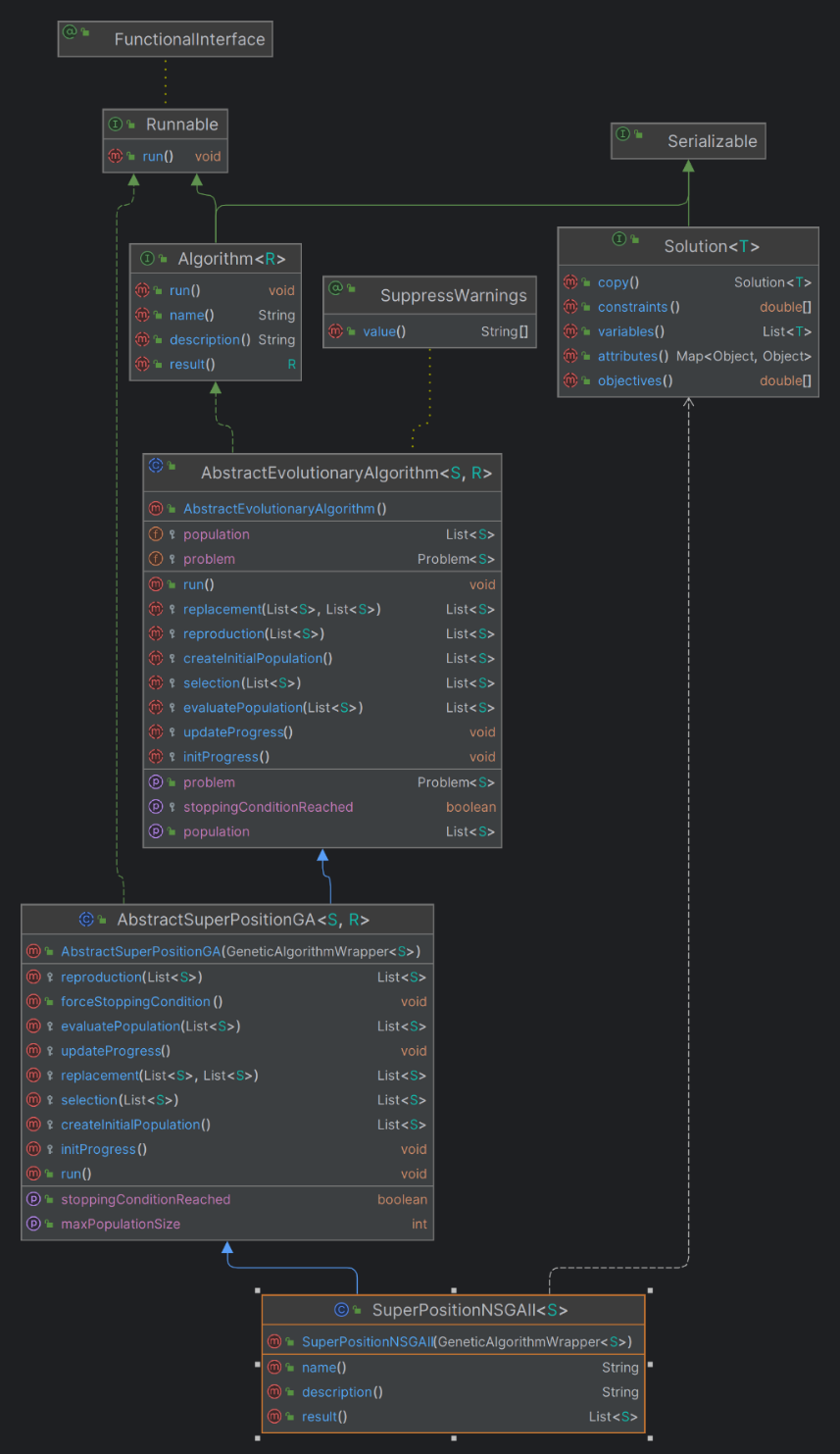


Fig.

### 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:

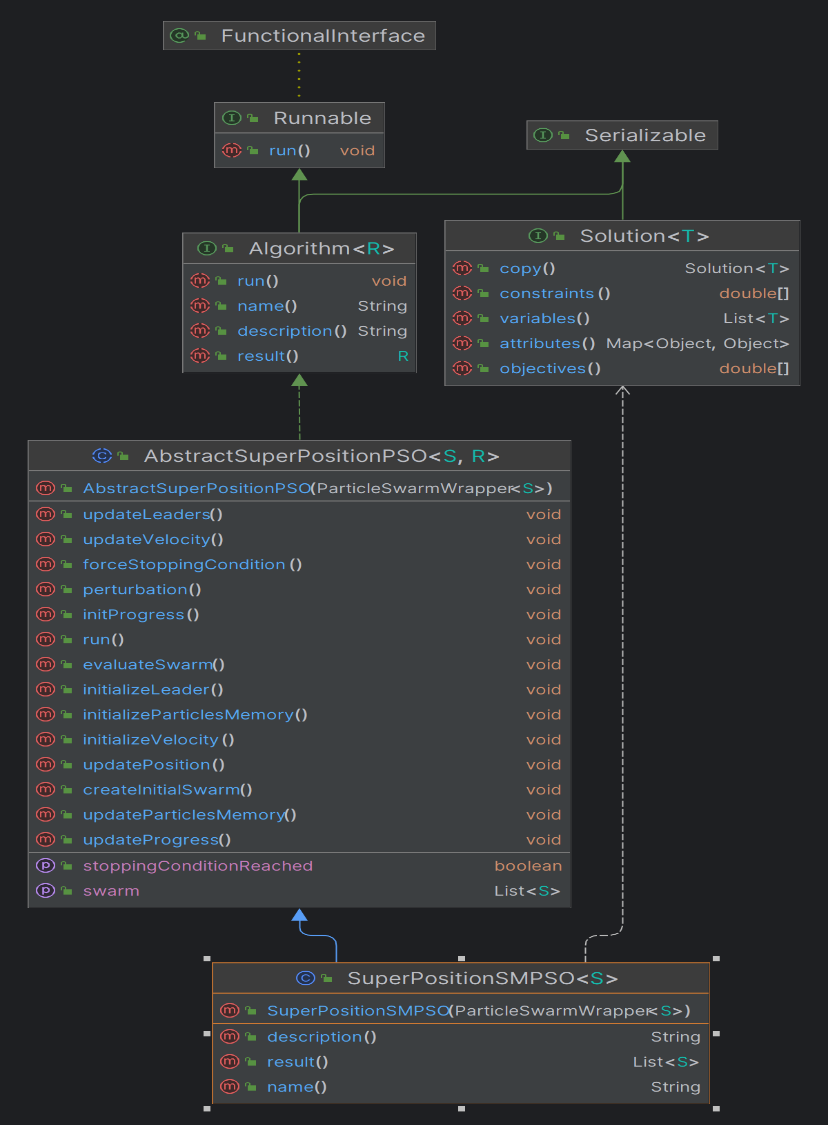


Fig.

### 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:

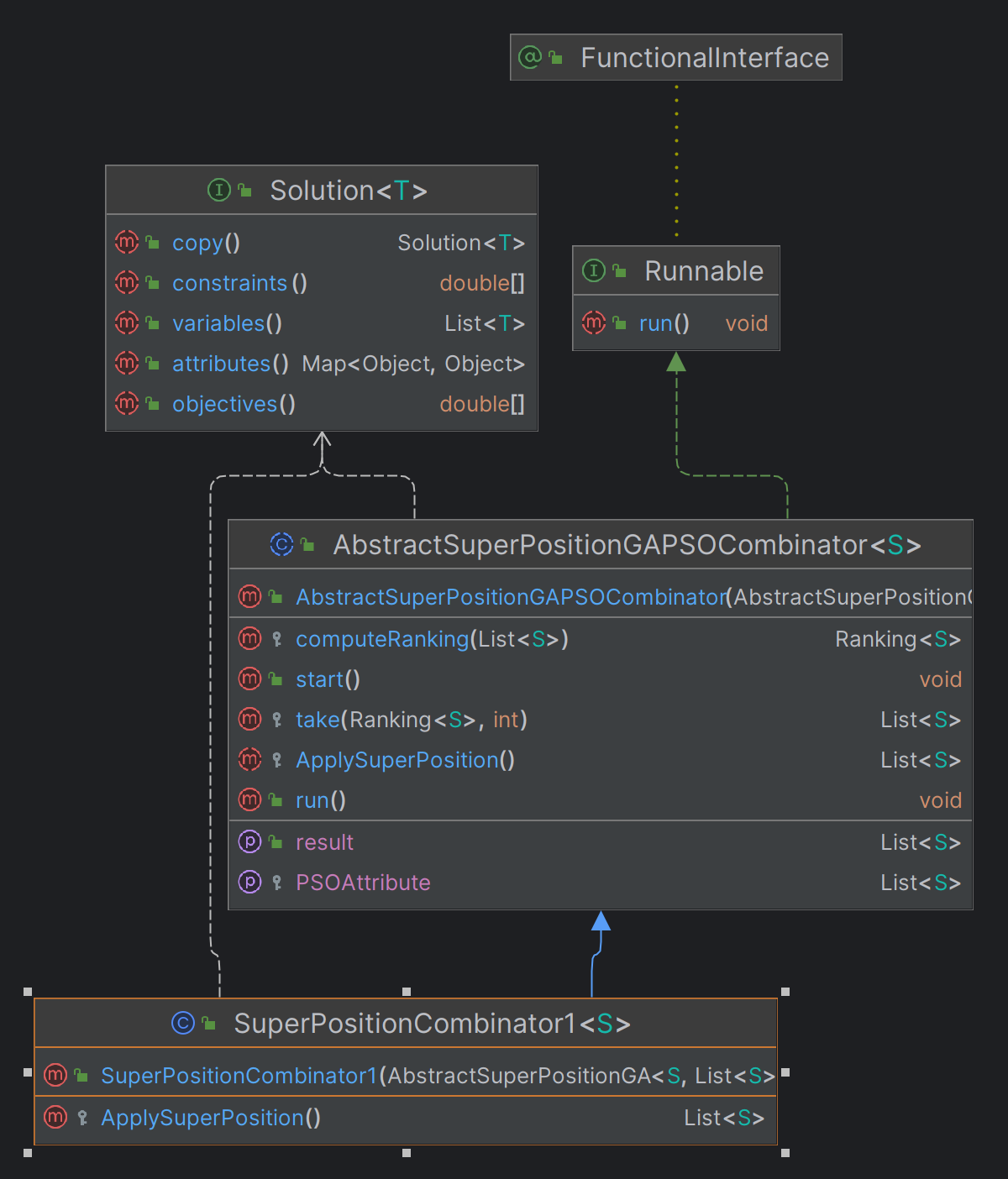


Fig.

### 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**;** |

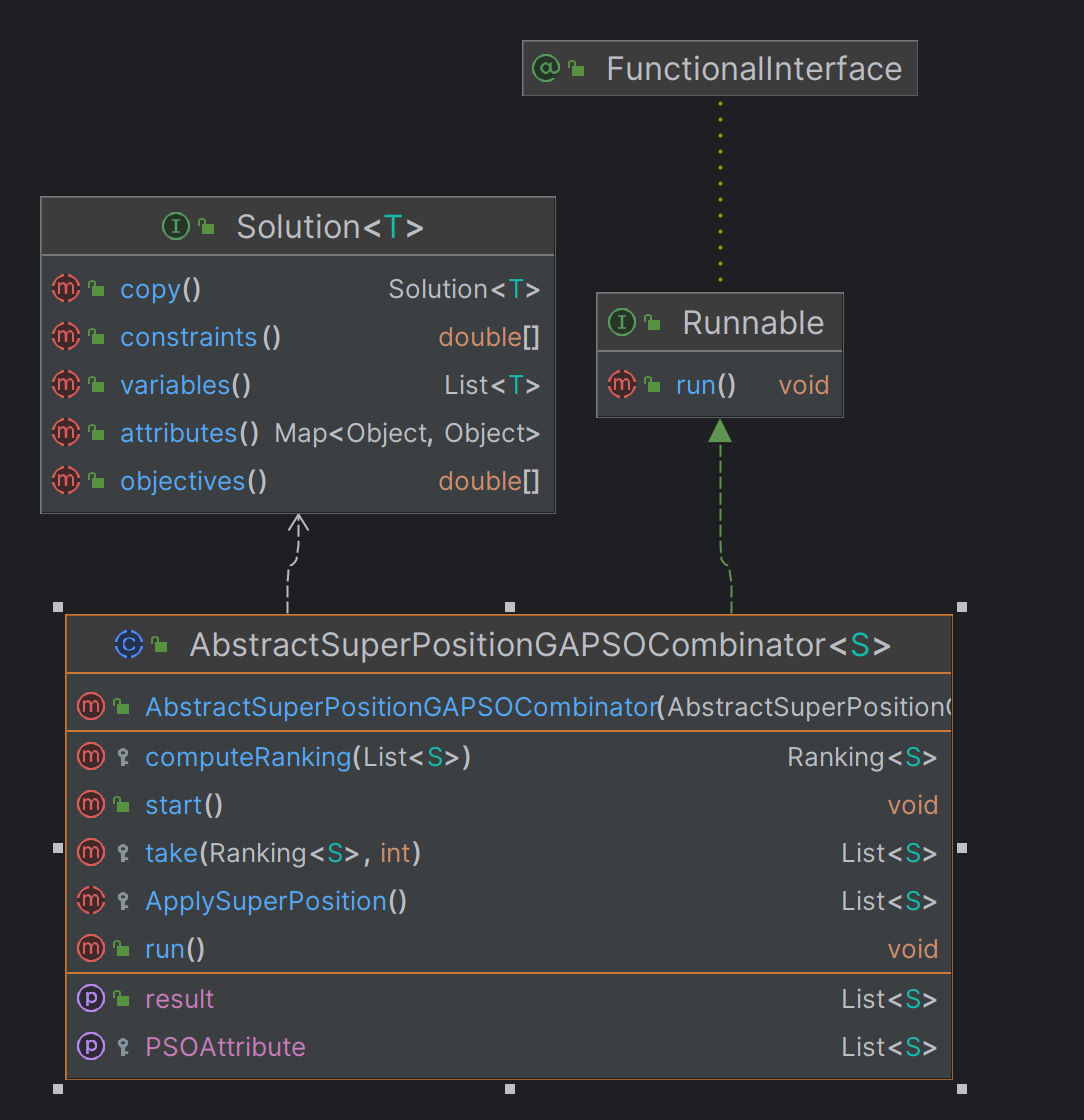


Fig.

## 4.5. Simulations

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. 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.

The coverage result if 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 sample a larger portion of the search space, potentially covering more regions 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 increases.
* 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.
* Integration of GAP simulator:
* GAPProblem
* GAPRunner
* GAP structure
* Config file
* Output file
* TSHD implementation
* Integration of benchmarks

# Chapter 5. Results

# Chapter 6. Conclusions

# Chapter 7. References