JCLEC-MO

JCLEC for multi-objective and many-objective optimisation

Design specification and programming guidelines

Version 1.0

Aurora Ramírez, José Raúl Romero, Sebastián Ventura
KDIS Research Group. University of Córdoba.
http://www.uco.es/grupos/kdis

March 9, 2018

Citing JCLEC-MO

Please, use the following citation to refer to JCLEC-MO:

A. Ramírez, J.R. Romero, S. Ventura. An Extensible JCLEC-based Solution for the Implementation of Multi-Objective Evolutionary Algorithms. *In Proceedings of the Companion Publication of the 17th Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*, pp. 1085-1092, ACM, 2015.

If you want to cite this document, you can also use the following citation:

A. Ramírez, J.R. Romero, S. Ventura (2018). JCLEC-MO: Design specification and programming guidelines. Version 1.0. Available for download from *http://www.uco.es/grupos/kdis/jclec-mo*

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

JCLEC-MO [1] is an extensible Java (JRE 8+) library aimed at providing specific algorithms and functionalities for both multi-objective and many-objective optimisation. Reusing general functionalities provided by JCLEC *core* (v4.0) [2], the framework includes implementations of well-known approaches like SPEA2 and NSGA-II, as well as other recently proposed methods based on decomposition techniques or landscape partition. In addition, JCLEC-MO provides a set of utilities to create experiments, report outcomes, evaluate the performance of algorithms in terms of quality indicators, as well as the necessary support to solve real-world optimisation problems. JCLEC-MO is compatible with other Java applications and analytical tools like datapro4j [3] and R[1].

The main reason behind the creation of this software was the growing interest in the resolution of multi-objective problems (MOPs) and many-objective problems (MaOPs). Separating the required functionalities to address multi-objective optimisation (MOO) from JCLEC *core* would help controlling their development in a more independent way and, at the same time, would allow an independent adaptation of those previously available elements of the library. As a result, JCLEC-MO better covers the specific necessities of both MOO practitioners and researchers.

1.1 Purpose

This document provides the structural design of JCLEC-MO in terms of its classes and interfaces. The reader can find several diagrams showing the different elements that take part in the definition of experiments. Explanations about these diagrams, code snapshots and additional considerations that developers should take care about are also included with the aim of providing the necessary information to understand how the classes interact and how they can be extended. It should be noted that private properties and methods are omitted. Protected properties and methods are shown when useful for external programmers.

JCLEC-MO is conceived to provide support to the development of new multi-objective optimisation approaches on the basis of the JCLEC framework. This document does not assume that the programmer has previous experience with JCLEC. Consequently, a brief introduction to the necessary concepts related to the library core is provided.

[1] `https://www.r-project.org`

The rest of the document is strictly focused on the JCLEC-MO classes that might be of interest to the programmer in the context of multi-objective optimisation research. It is worth mentioning that JCLEC-MO has been designed to be easily extended, adding new algorithms, utilities, etc. All these aspects are independent of each other and the existing *extension points* are clearly stated.

1.2 Licence

This software was developed by members of the Knowledge Discovery and Intelligent Systems (KDIS) Research Group at the University of Córdoba, Spain. For further information on the library and modifications, please visit the URL `http://www.uco.es/grupos/kdis/jclec-mo`.

1.3 Software distribution

JCLEC-MO is provided for download in different ways:

- The runnable *jar* file, *jclec-mo.jar*.
- The binary *jar* file, *jclec-mo-1.0.jar*.

The following external libraries are required:

- JCLEC *base* (v4.0+), either in binary form or as source files, which can be obtained from `http://jclec.sourceforge.net`.
- datapro4j library core (v1.0+), and the additional module to access to R[2]. They are available for download from `http://www.uco.es/grupos/kdis/jclec-mo`. *datapro4j* is only required by some *reporters* and *handlers* (see Section 4).

[2]R and *rJava* package should be installed. They can be obtained from `https://www.r-project.org` and `http://www.rforge.net/rJava`, respectively.

- Apache Commons libraries, which can be obtained from `https://commons.apache.org`. The following libraries are required: *collections* (v3.2.2), *configuration* (v1.10), *lang* (v2.6) and *logging* (v1.2+)[3].
- JUnit (v4.12+), as well as harmcrest-core (v1.3+), which are available for download from `http://junit.org`. They are only required by test classes.

Both JCLEC-MO code and documentation, as well as external dependencies, can be downloaded from the project website: `http://www.uco.es/grupos/kdis/jclec-mo`. The API is also accessible via `http://www.uco.es/grupos/kdis/jclec-mo/v1/api`.

1.4 Best practices

The JCLEC-MO project is being developed following Software Engineering best practices:

- Modular and extensible design. The library has been designed in such a way that it remains, as much as possible, independent from the library core. In addition, extension points are clearly stated to facilitate adding new elements without requiring the recompilation of the library.
- Use of design patterns. The following design patterns [4] have been included or adapted to solve specific design issues with respect to some core restrictions or new required functionalities:
  - Command. Recurrent operations like objective transformations are provided as independent *commands*, which are highly configurable and interchangeable.
  - Strategy. Multi-objective algorithms are defined as *strategies*, allowing easily interchanging them by means of a configuration file. In addition, this pattern servers to clearly specify the steps that should be implemented by any developer.
  - Prototype. Fitness objects are based on this pattern in order to include those specific properties that some algorithms consider beyond the objective values in a flexible way. It also allows their adaptation to new variants or hybrid proposals.
  - Chain of responsibility. Post-processing operations are defined as *handlers* that can be connected in chains in order to create different types of experimental studies.

[3]Due to compatibility issues between JCLEC *base* and Apache Commons latest versions, JCLEC-MO is required to include *collections* v.3.2.2, *configuration* v1.10 and *lang* v2.6.

- Clean and fully documented source code. The source code has been carefully documented using *Javadoc* tags.
- Use of unit test. JUnit has been used to define and execute test cases, assessing the correctness of the most important features of the library.

2. OVERVIEW OF THE ARCHITECTURE

This chapter explains the design of JCLEC-MO, starting from a brief summary of JCLEC. Next, the classes that have been extended and adapted to address the specific requirements of MOO research are detailed. Most of them also constitute the extension points for further developments and contributions. Finally, some details about the execution workflow and the Contextualisation mechanism are explained in order to clarify the interaction between core classes.

2.1 JCLEC core

JCLEC core provides a variety of encodings and genetic operators that can be combined to solve different kinds of optimisation problems. Experiments are created using a configuration file in XML format, whose tags and elements can be adapted by researchers in order to include their own parameters. As a result, customised operators or new optimisation problems can be easily added without requiring the code to be recompiled. The set of available evolutionary techniques is comprised of generational schemes, steady-state evolution, niching methods and genetic programming. For a detailed explanation of the software, the reader is referred to [2].

Figure 2.1 shows the core classes and interfaces of JCLEC. `RunExperiment` is the class in charge of starting the execution of an algorithm, which is represented by the `IAlgorithm` interface. The `IAlgorithmListener` interface specifies the set of methods used to report the outcomes and get information about the algorithm execution during the search process.

In JCLEC, new algorithms should be created extending from the abstract class that implements the `IAlgorithm` interface, i.e. `AbstractAlgorithm`. `PopulationAlgorithm` specifies the steps that should implement any evolutionary algorithm (EA). More specifically, every EA evolves a set of individuals (`IIndividual`), i.e. the population, and makes use of a number of tools that serve to conduct the different steps of the optimisation process: the initialisation (`IProvider`), the application of genetic operators (`IRecombinator` and `IMutator`), and the selection of parents (`ISelector`).

An evolutionary algorithm also requires the assignment of an evaluation mechanism to calculate the fitness of individuals according to the optimisation problem being solved.

In JCLEC, the interface `IEvaluator` declares the evaluation method, `evaluate`, and two other additional abstract classes, `AbstractEvaluator` and `AbstractParallelEvaluator`, define how this evaluation task has to be performed, sequentially or in parallel, respectively. In any case, the *evaluator* is in charge of assigning a fitness value to each individual using a subclass of `AbstractFitness`, an implementation of `IFitness`.

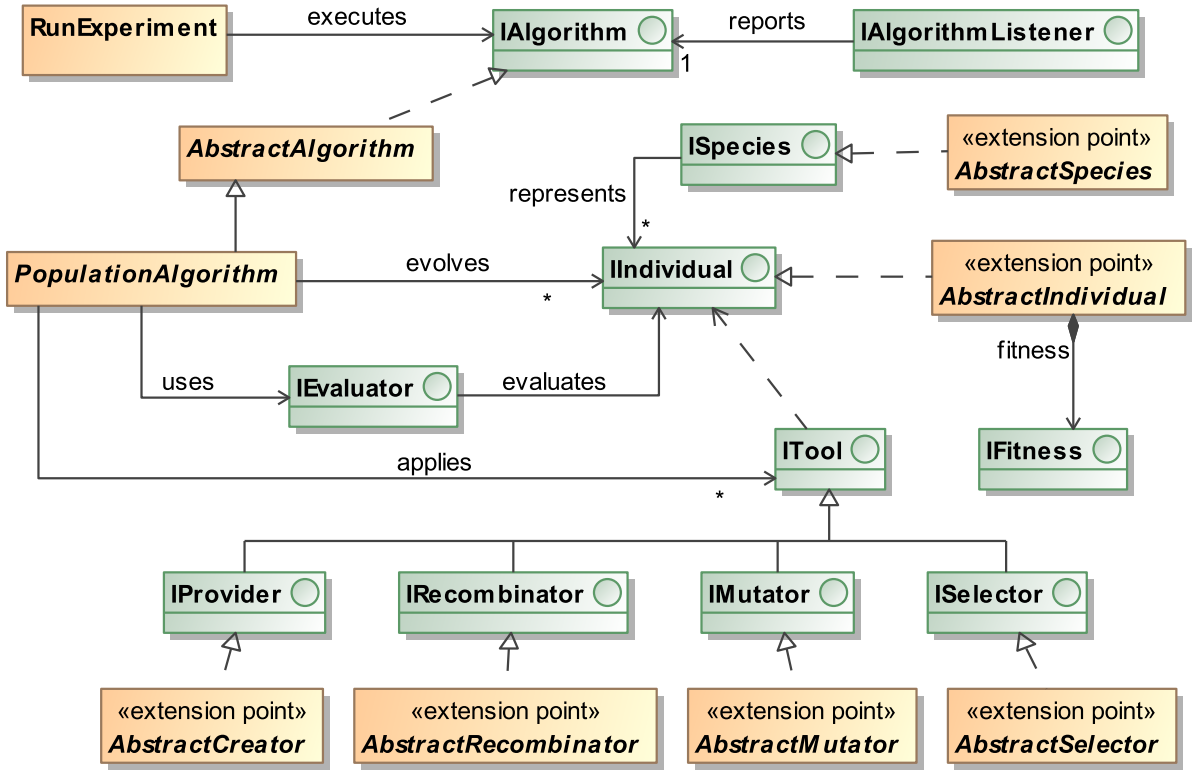

Figure 2.1: Core classes and interfaces in JCLEC

Most of the aforementioned classes represent extension points of JCLEC that are still valid for JCLEC-MO, since new domain-specific elements, i.e. tools and encodings, can be developed just by extending these classes. Abstract classes that have not been marked as "extension points" in Figure 2.1, as well as other classes that do not appear in the diagram, might represent extension points of JCLEC, but they have to be previously adapted to the specific requirements of MOO research. Thus, new extension points will be provided in order to include new *algorithms*, *evaluators* and *listeners*, among other elements, as detailed in Section 2.2.

A more detailed explanation of the `PopulationAlgorithm` class might help in understanding how JCLEC defines the iterative process of an evolutionary algorithm. Figure 2.2 depicts the control flow performed by this class, where italics indicate abstract operations. Subclasses of `PopulationAlgorithm`, such as `SG` (simple generational genetic algorithm), `SS` (steady-state genetic algorithm) or the previous versions of SPEA2 and NSGA-II, implement the overall search process, comprising the following key methods: `doSelection`, where a *selector* creates the mating pool; `doGeneration`, where genetic operators like

recombinators and *mutators* are executed; `doReplacement`, where survivors are selected among the current population and offspring; and `doUpdate`, responsible for creating the next population. Several different sets of individuals, referring to the current population (`bset`), parents (`pset`), offspring (`cset`), and survivors (`rset`), are managed by this class along the evolution. The search can be stopped in JCLEC when a maximum number of generations or evaluations is reached. Additionally, the execution could be interrupted if the best individual in the population achieves an acceptable value for the problem being solved.

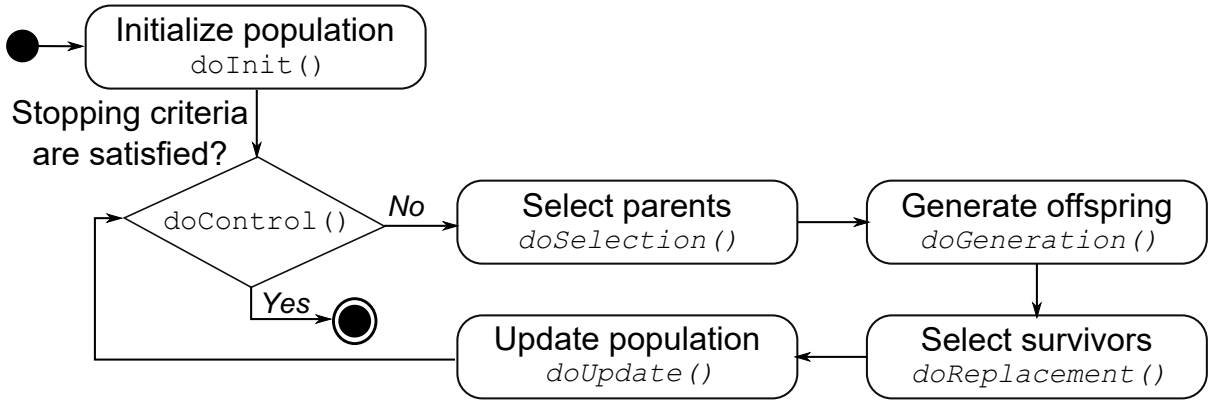

Figure 2.2: Iterative process of `PopulationAlgorithm`

2.2 JCLEC-MO main classes

Figure 2.3 shows an overview of the main classes and interfaces that compose JCLEC-MO, including those required to wrap JCLEC in order to adapt its functionality to the specific requirements of MOO. Classes that can be extended by the external developer to continue expanding the JCLEC-MO functionalities are identified in the diagram as "extension points". A brief description of the functionality of each class is provided next:

`MOECAlgorithm` and `MOPSOAlgorithm`. These abstract class represent the general structure of multi-objective evolutionary algorithms and multi-objective particle swarm optimisation algorithms. `MOECAlgorithm` inherits from `PopulationAlgorithm` and `MOPSOAlgorithm` has `PSOAlgorithm` as its base class[1]. They separate the search steps that correspond to domain-specific elements from those steps really attributable to the multi-objective algorithm, as detailed in Section 3.1. Both classes implement the interface `IMOAlgorithm`, which specifies the common operations of any multi-objective algorithm.

[1]This class has been defined in JCLEC-MO to isolate the general properties and elements that would participate in a PSO algorithm, regardless the number of objectives. It also serves to maintain the consistency with respect to the hierarchy of algorithms.

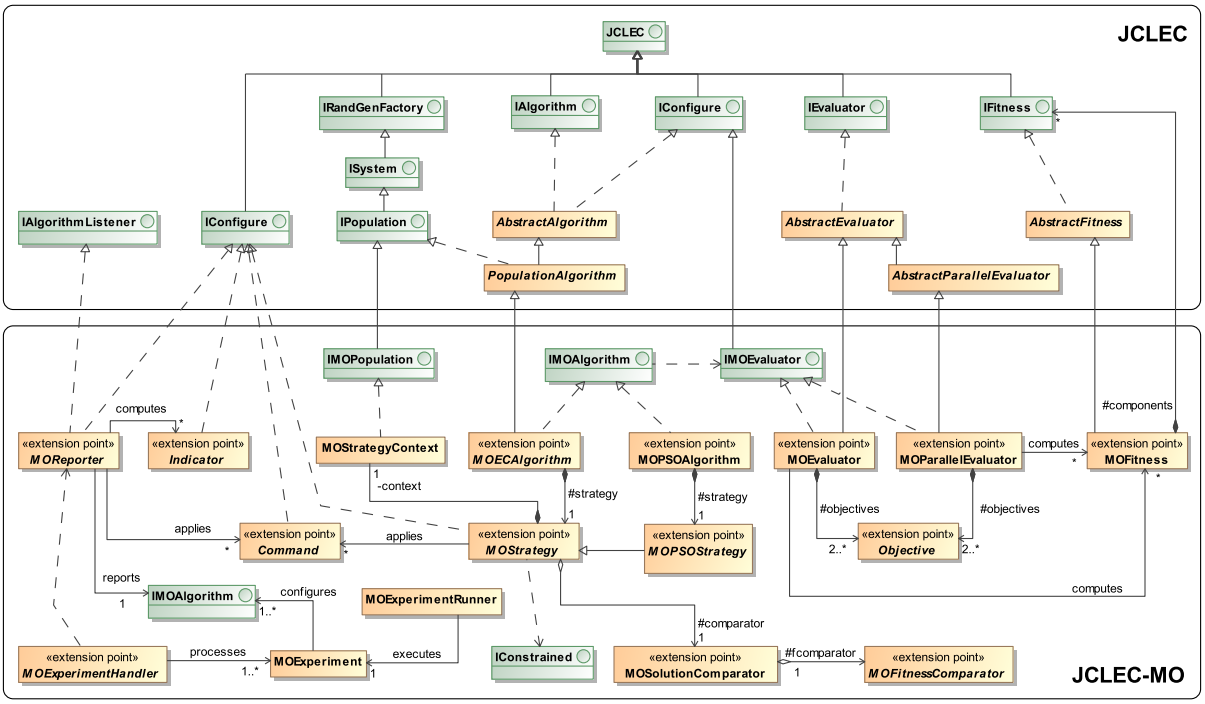

Figure 2.3: Core classes and interfaces in JCLEC-MO

`MOStrategy` and `MOPSOStrategy`. These abstract classes specify the set of methods that any multi-objective algorithm should implement, on the basis of the *Strategy* design pattern. `MOPSOStrategy` defines the additional methods required by MOPSO approaches. Information about the overall search process can be retrieved from the execution context, as will be explained in Section 3.2.

`MOEvaluator` and `MOParallelEvaluator`. These concrete classes implement the evaluation mechanism in two different ways: sequentially or in parallel. These classes inherit their properties and methods from their respective classes in the JCLEC layer and handle the set of objectives functions required to evaluate solutions. Both *evaluators* implement the interface `IMOEvaluator`, which declares the public methods that any *evaluator* should implement.

`Objective`. It is the abstract class that serves as a basis for defining the necessary information about each objective function of the MOP.

`MOFitness`. This class encapsulates the fitness of an individual, which should be computed by an *evaluator*. Each objective value should be stored using a class implementing the `IFitness` interface defined in JCLEC. Usually, `SimpleValueFitness` will be selected.

`MOSolutionComparator` and `MOFitnessComparator`. These classes serve to perform comparisons between solutions or between their fitness objects. They implement the `Comparator<T>` Java interface, where T refers to `IIndividual` and `IFitness`, respectively.

`MOReporter.` This class represents a general reporter that will act as a *listener* of an multi-objective algorithm. More specifically, it provides the facilities to extract information from the algorithm outcomes during and after its execution.

`Command.` Based on the design pattern with the same name, this class specifies the general structure of recurrent operations that can be performed on a set of solutions, e.g. objective transformations. Properly configured by the user or the system itself, they can be executed by both *listeners* and *strategies*.

`Indicator.` This class constitutes the starting point of the hierarchy of quality indicators that can be used to assess the performance of the algorithms.

`MOExperiment` **and** `MOExperimentRunner.` The former class allows the definition of an experiment as a set of configuration files to be executed. The latter class is in charge of executing MOO experiments, using the class `RunExperiment` defined in JCLEC.

`MOExperimentHandler.` This abstract class specifies the general structure of post-processing operations that can be performed after the execution of experiments.

A more detailed description of these classes and their key interactions will be provided in the following chapters.

3. THE SEARCH PROCESS IN JCLEC-MO

An important feature of JCLEC-MO lies on the independence between the different stages of the search process. In [5], the authors identify three different elements of a general stochastic search algorithm: the memory that stores the current solutions, the selection module and the variation module. Frequently, MOEAs only define those procedures aimed at selecting and replacing individuals, also referred to as *mating selection* and *environmental selection*, respectively. Nevertheless, these methods can be reinforced by some kind of evaluation method, named *fitness assignment* in [5], which is executed before sampling individuals. Finally, MOEAs can manage an external archive of solutions in order to promote characteristics like elitism and diversity preservation. Influenced by MOEAs, MOPSO algorithms have adopted a similar structure, since they frequently use external archives and include specific methods to select leaders based on diverse quality criteria [6]. JCLEC-MO adopts this schema, and delegates the control of MOO-specific steps to a class named `MOStrategy`, an adapted implementation of the *Strategy* design pattern. In this chapter, the execution workflow among *algorithms* and *strategies* is described in detail. Some issues regarding the contextualisation mechanism are also explained.

3.1 Execution workflow

In JCLEC-MO, *algorithms* are the classes that encapsulate the iterative process of a specific metaheuristic, e.g. evolutionary computation (EC) and particle swarm optimisation (PSO). Therefore, *algorithms* are responsible for invoking the multi-objective *strategy* at certain stages of the search process. Figure 3.1 shows how an *algorithm* and a *strategy* interact each other to perform the overall evolutionary process. Italic typeface is used to indicate abstract methods. Firstly, `MOECAlgorithm` implements the steps of an evolutionary search as follows:

1. `doInit`. After the creation of the population (`bset`), which is actually performed by `PopulationAlgorithm` (line 2), the *strategy* creates the initial archive (line 3).

```
1 protected void doInit() {
2     super.doInit();
3     this.archive = this.strategy. initialize (this.bset);
4 }
```

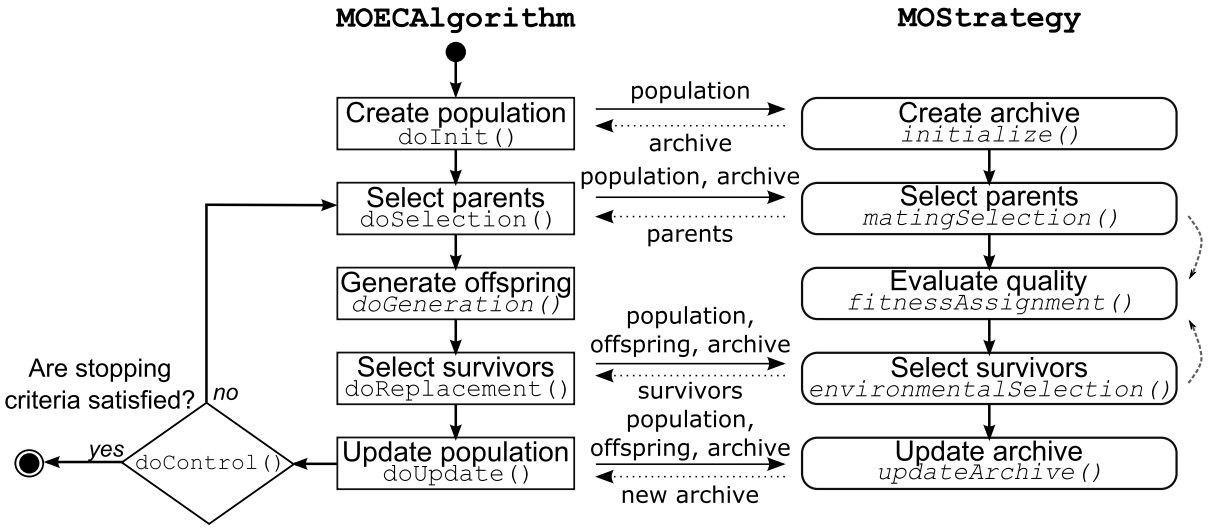

Figure 3.1: Interaction between an EA and the multi-objective strategy

2. `doSelection`. It corresponds to the *mating selection* mechanism in a MOEA, so the *algorithm* delegates its execution to the *strategy*. The set of individuals involved in the process are the current population (`bset`) and the current archive (`archive`), if exists. The returning object represents the set of parents (`pset`).

```
1 protected void doSelection() {
2     // Select parents from the current population
3     this.pset = this.strategy.matingSelection(this.bset, this.archive);
4 }
```

3. `doGeneration`. The variation mechanism remains abstract in `MOECAlgorithm`, so it will be implemented by the specific subclasses according to the corresponding evolutionary paradigm, i.e. genetic algorithms, evolution strategies, etc.

4. `doReplacement`. The selection of survivors (`rset`) is carried out by the *strategy*, choosing from individuals belonging to the current population (`bset`), the set of offspring (`cset`) and the archive (`archive`), if exists.

```
1 protected void doReplacement() {
2     // Select survivors
3     this.rset = this.strategy.environmentalSelection(this.bset, this.cset, this.archive);
4 }
```

5. `doUpdate`. In this phase the *algorithm* should update both the archive (line 3) and the population (line 5). The former is managed by the multi-objective approach, so the *strategy* should be invoked considering the current population (`bset`), the generated offspring (`cset`) and the current archive (`archive`). Next, the *algorithm* replaces the current population with the set of survivors (`rset`) obtained in the

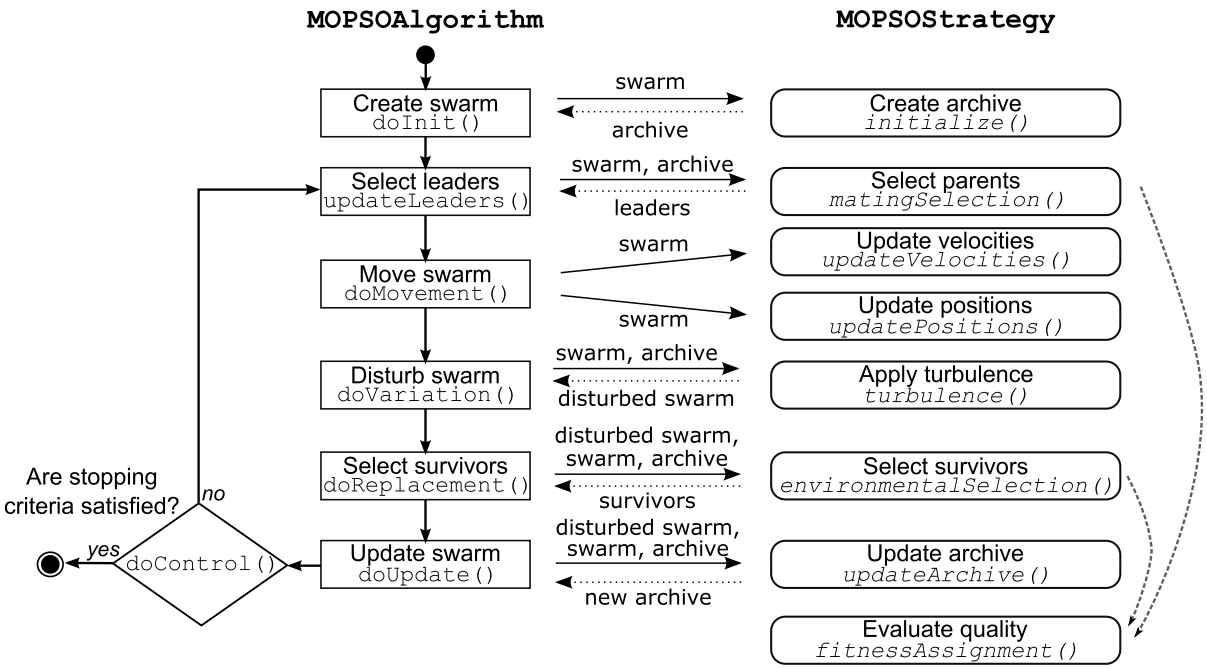

Figure 3.2: Interaction between a PSO algorithm and the multi-objective strategy

previous step. Finally, the *strategy* is also updated (line 7), allowing the execution of any additional operation to be performed before the end of the iteration.

```
1 protected void doUpdate() {
2     // Update archive
3     this.archive = this.strategy.updateArchive(this.bset, this.cset, this.archive);
4     // Update current population
5     this.bset = this.rset;
6     // Update strategy
7     this.strategy.update();
8     ...
9 }
```

In the case of MOPSO, the implementation of `doInit`, `doReplacement` and `doUpdate` methods is equivalent to that explained for MOEAs. However, a MOPSO algorithm will delegate to the *strategy* additional stages of the search process (see Figure 3.2):

- `updateLeaders`. In this step, the *algorithm* invokes the *strategy* to select the leaders (`leaders`) considering both the current swarm (`swarm`) and the archive (`archive`).

```
1 protected void updateLeaders() {
2     this.leaders = this.strategy.matingSelection(this.swarm, this.archive);
3 }
```

- `updateVelocities`. The *strategy* will update the velocity of each particle (`swarm`) considering the designated leader (`leaders`). The velocities are modified in the given swarm.

```
1 protected void updateVelocities() {
2     this.strategy.updateVelocities(this.swarm, this.leaders);
3 }
```

- `updatePositions`. This operation is delegated to the *strategy* because the MOPSO approach might determine its own mechanism to control that particle position do not exceed the bounds of the decision variables. The new positions should be fixed in the given swarm.

```
1 protected void updatePositions() {
2     this.strategy.updatePositions(this.swarm);
3 }
```

- `doVariation`. As MOPSO algorithms usually include a *turbulence* mechanism to promote diversity, the generation of a disturbed swarm (`disturbedSwarm`) should be performed by the *strategy*, receiving the current swarm (`swarm`) from the *algorithm* (line 3). Then, the algorithm should evaluate the new particles (line 5) and initialise their local memory (lines 7-13).

```
1  protected void doVariation() {
2      // Apply the turbulence mechanism
3      this.disturbedSwarm = ((MOPSOStrategy)this.strategy).turbulence(this.swarm);
4      // Evaluate the new solutions
5      this.evaluator.evaluate(this.disturbedSwarm);
6      // Set the best position and fitness
7      Particle particle;
8      int size = this.disturbedSwarm.size();
9      for(int i=0; i<size; i++){
10         particle = (Particle)this.disturbedSwarm.get(i);
11         particle.setBestPosition(particle.getPosition());
12         particle.setBestFitness(particle.getFitness());
13     }
14 }
```

Finally, it should be noted that the `fitnessAssignment` method is not really invoked by the *algorithms*, since each *strategy* might require the execution of the evaluation mechanism in a different stage of the process. Consequently, each concrete *strategy* should decide whether it requires this method, and when it should be executed.

3.2 Contextualisation

In JCLEC, the components of an experiment are controlled by the algorithm, so they cannot know by themselves neither what are the other components that have been configured nor what is the current state of the search. However, certain aspects of the experiment have to be shared, e.g. the random number generator, in order to ensure that the experiment remains consistent. To do this, JCLEC proposes the so-called *contextualisation* mechanism.

Three JCLEC interfaces, `IRandGenFactory`, `ISystem` and `IPopulation`, dictate the components of an experiment that should be accessible to others. `PopulationAlgorithm` implements all these interfaces, thus providing access to some of its properties:

1. The *species*, which contains information about the encoding and the optimisation problem.
2. The random number generator, which should be an implementation of `IRandGen`.
3. The *evaluator* of solutions, which also defines how the fitness of the individuals should be compared.
4. The current population, a.k.a inhabitants, and the size of the population as it was configured.
5. The number of the current generation.

When creating and configuring JCLEC *tools*, the algorithm is in charge of contextualising them by invoking the method `contextualize`, which is specified in the interface `ITool` [2]. More specifically, *tools* receive a reference to the algorithm. An example of how `PopulationAlgorithm` contextualises the *provider* is shown in the following code fragment:

```
1 public abstract class PopulationAlgorithm extends AbstractAlgorithm implements IPopulation {
2     ...
3     public final void setProvider(IProvider provider) {
4         this.provider = provider;
5         provider.contextualize(this);
6     }
7     ...
8 }
```

This mechanism, although effective, slightly hampers the code comprehensibility and overloads the definition of algorithms, specially when new properties are added or the

adaptation of the algorithm's structure is required, as happens with multi-objective algorithms. Therefore, JCLEC-MO defines a new Contextualisation mechanism, which is compatible with existing JCLEC *tools*. In JCLEC-MO, the context is explicitly represented by the `MOStrategyContext` class, a property of `MOStrategy`, that stores the references to the aforementioned general properties, as well as to the following JCLEC-MO specific elements:

- The current archive of solutions. Although the strategy receives the required set of solutions from the algorithm, this property has been included in the context in order to maintain consistency, since providing access to the current population through the context is mandatory.
- A prototype of the fitness object to be used during evaluation. In some cases, the *strategy* requires that solutions store their fitness using a specific fitness object, so it is now a parameter of the configuration file that is retrieved by the own *strategy*. However, the selected class should be also available to the *evaluator*, as it is the responsible of creating the fitness object during the evaluation phase. Including a prototype of the selected fitness class in the context will allow the *evaluator* to know what type of fitness object should create and assign to the solutions.
- The comparator of solutions. Operators such as crossovers and mutators might require making comparisons of either fitness objects or solutions. Since JCLEC-MO explicitly defines comparisons at the solution level, the *comparator* created by the *strategy* should be also accessible.

Although the context belongs to the *strategy*, the *algorithm* is the class in charge of creating and updating it, since it manages the rest of components of the search process. The initial context will be created during the configuration of the experiment, and the process has to follow a special order to ensure that all the elements have been already configured (see the code snapshot given below). After invoking the configuration method of `PopulationAlgorithm` (line 4), the *provider* and the *evaluator* have been created. The next step consists in configuring the *strategy* (line 5), in which the context will be also created (lines 22-29). The method `contextualizeStrategy` is declared in the interface `IMOAlgorithm`, so all the algorithms are expected to implement it. Once the context has been created, the algorithm can contextualise the *provider* (line 6). Next, the properties needed by the *evaluator* can be properly configured (lines 8-11).

```
1 public abstract class MOECAlgorithm extends PopulationAlgorithm implements IMOAlgorithm {
2     ...
3     public void configure(Configuration settings) {
4         super.configure(settings);          // Evaluator is configured in the super class
5         setStrategySettings(settings);      // Configure the multi-objective strategy
6         this.provider.contextualize(getContext()); // Set the context in provider
7         // Set the fitness comparator and the fitness prototype in the evaluator
8         if(this.evaluator instanceof IMOEvaluator){
```

```
9             ((IMOEvaluator)this.evaluator).setComparator(this.strategy.getSolutionComparator().
                  getFitnessComparator());
10            ((IMOEvaluator)this.evaluator).setFitnessPrototype(this.strategy.getContext().
                  getFitnessPrototype());
11        }
12        else
13            throw new IllegalArgumentException("Evaluator should extend IMOEvaluator interface");
14        ...
15    }
16    protected void setStrategySettings(Configuration settings){
17        ...
18        // Configure the execution context
19        contextualizeStrategy();
20        ...
21    }
22    public void ContextualizeStrategy(){
23        if(this.strategy!=null){
24            MOStrategyContext context =
25                    new MOStrategyContext(this.randGenFactory.createRandGen(), this.species,
26                        this.evaluator, this.strategy.getSolutionComparator(), this.populationSize);
27            this.strategy.setContext(context);
28        }
29    }
30    ...
31}
```

A reference to both the population and the archive should be included in the context after their creation. This process is done in the `doInit` method, as shown below:

```
1 public abstract class MOECAlgorithm extends PopulationAlgorithm implements IMOAlgorithm {
2     protected void doInit(){
3         // Initialize the populations
4         ...
5         // Set the populations in context
6         this.strategy.getContext().setInhabitants(this.bset);
7         this.strategy.getContext().setArchive(this.archive);
8     }
9 }
```

Similarly, the *algorithm* should update those properties of the context that change after each generation (see the code snapshot below). The number of the current generation should be set before starting a new iteration (line 5), whereas populations that survive to the next generation have to be updated at the end of the iteration (lines 11-12).

```
1 public abstract class MOECAlgorithm extends PopulationAlgorithm implements IMOAlgorithm {
2     ...
3     protected void doSelection() {
4         // Update the generation in the context
5         this.strategy.getContext().setGeneration(this.generation);
6         ...
7     }
```

```
8       protected void doUpdate() {
9           ...
10          // Update populations in context
11          this.strategy.getContext().setInhabitants(this.bset);
12          this.strategy.getContext().setArchive(this.archive);
13      }
14  }
```

The contextualisation process of MOPSO algorithms is equivalent to that explained above. In general, if programmers want to modify or extend these base classes, they should carefully check that contextualisation is properly performed. A recommended option is to call the implementation of the `configure` method in the super class before configuring any other component or parameter in the new algorithm, whenever it is possible. In addition, concrete algorithms should contextualise their specific *tools*, e.g. genetic operators, by providing them with a reference to the context instead of to themselves. For instance, the following code fragment shows how `MOESAlgorithm` contextualises the mutation operator:

```
1   public class MOESAlgorithm extends MOECAlgorithm {
2       ...
3       public void setMutator(AbstractMutator mutator) {
4           this.mutator = mutator;
5           this.mutator.contextualize(getContext());
6       }
7   }
```

4. JCLEC-MO PACKAGES

This chapter describes the packages comprising JCLEC-MO. Firstly, a package diagram is presented to introduce the general structure of the library. The following sections give more details about the main classes included in each package.

4.1 Package diagram

Figure 4.1 shows the package diagram of JCLEC-MO. Next, the content of each package and existing dependencies among them are explained:

- `net.sf.jclec.mo`. The base package includes four interfaces: `IMOPopulation`, `IMOAlgorithm`, `IMOEvaluator` and `IConstrained`.

- `net.sf.jclec.mo.algorithm`. This package contains the abstract classes representing *algorithms*, i.e. `MOECAlgorithm` and `MOPSOAlgorithm`, as well as the subclasses implementing different EC paradigms, e.g. genetic algorithms and genetic programming. *Algorithms* will use *strategies* and *evaluators* to perform the search.

- `net.sf.jclec.mo.command`. All the currently available *commands* are stored in this package. Operations to invert and scale objective values, among others, have been implemented.

- `net.sf.jclec.mo.comparator`. All the required *comparators* are grouped together into this package. *Comparators* at the fitness level will belong to the inner package.

- `net.sf.jclec.mo.distance`. JCLEC-MO provides implementations of the interface `IDistance`, already declared in JCLEC, to calculate three types of distances between solutions in the objective space: `ManhattanDistance`, `EuclideanDistance` and `EuclideanHypercubeDistance`.

- `net.sf.jclec.mo.evaluation`. This package includes all the classes related to the evaluation mechanism: `MOEvaluator`, `MOParallelEvaluator`, `Objective` and `MOFitness`. Subclasses of `MOFitness`, which are specifically designed to store additional properties defined by some *strategies*, are stored in an inner package.

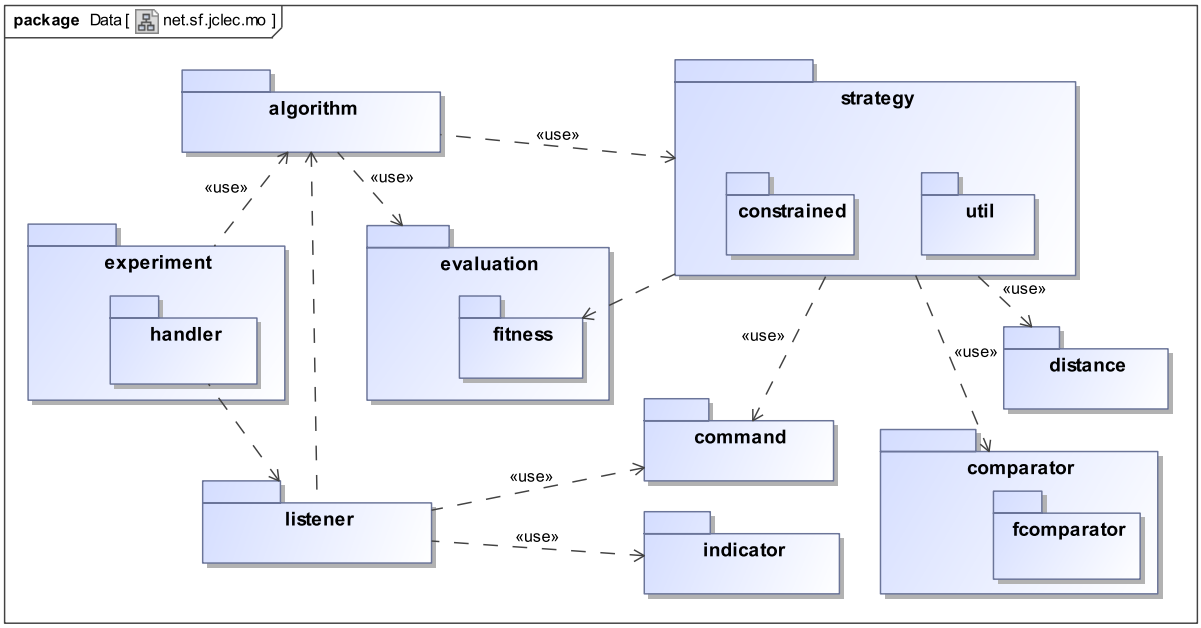

Figure 4.1: JCLEC-MO packages

- `net.sf.jclec.mo.experiment`. Classes related to the creation and execution of experiments are included in this package. These classes allow the execution of *algorithms*. The inner package contains the set of *handlers* that can be used to perform post-processing operations using the outcomes generated by *reporters*.
- `net.sf.jclec.mo.listener`. This package contains the abstract *reporter* and its subclasses, which will be used to report the outcomes of an *algorithm*. Some specific *reporters* might require the invocation of both *indicators* and *commands*.
- `net.sf.jclec.mo.strategy`. JCLEC-MO stores the multi-objective algorithms in this package, which is comprised of two subpackages: `constrained` and `util`. The former contains the constrained variants of all the available *strategies*. The latter includes auxiliary classes. Throughout its execution, *strategies* might calculate distances, execute *commands* or compare solutions. They can also compute additional properties for the solutions, which will be stored as part of their *fitness* object.

4.2 Package net.sf.jclec.mo.algorithm

Figure 4.2 shows a class diagram focused on the classes representing *algorithms*. Some JCLEC classes and interfaces, represented without background colour, have been included for the sake of comprehensibility. The following aspects should be taken into account:

- Each subclass of `MOECAlgorithm` implements the general behaviour of a different

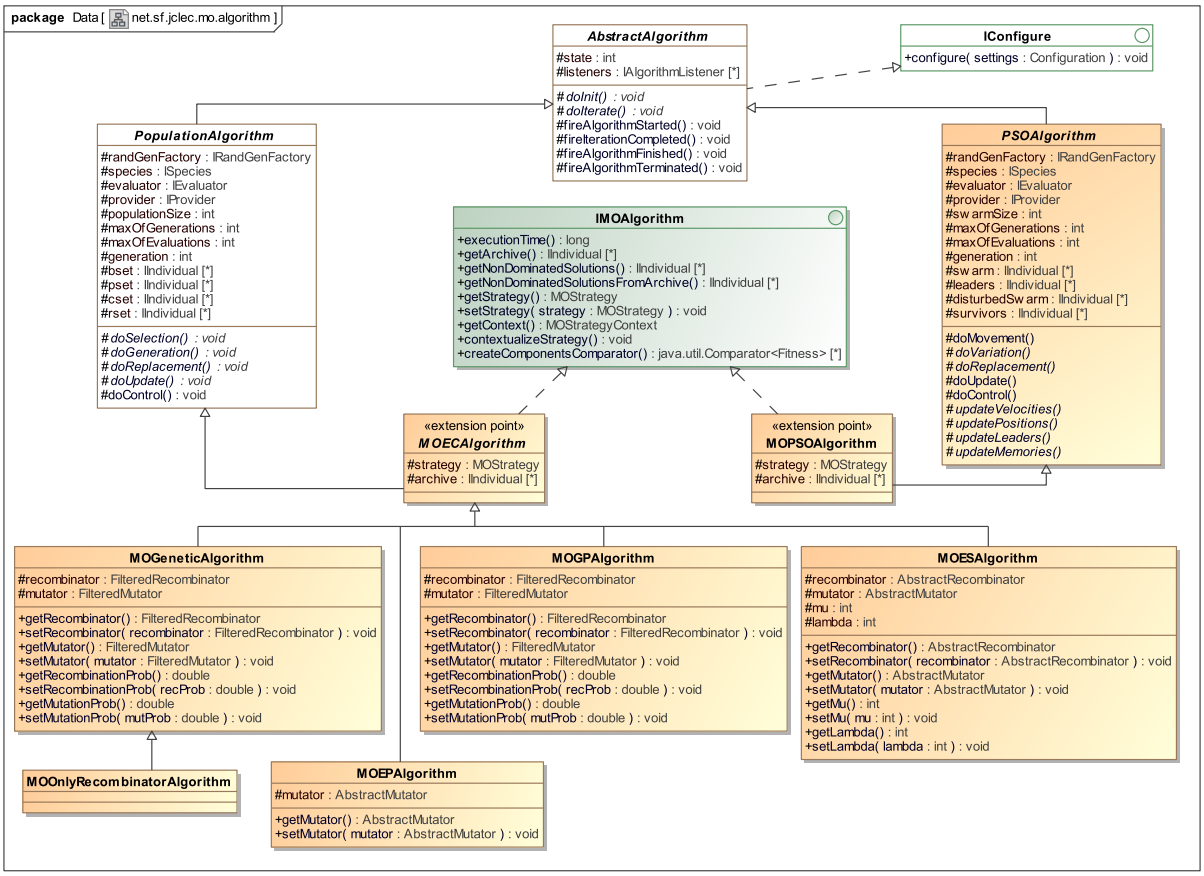

Figure 4.2: Class diagram: package net.sf.jclec.mo.algorithm

evolutionary paradigm regarding the variation mechanism, which remains as an abstract method in `MOECAlgorithm`.

- The *strategy* is a protected property of `MOECAlgorithm` and `MOPSOAlgorithm`. The *algorithm* will invoke the *strategy* when required, as explained in Section 3.1.

- `MOPSOAlgorithm` will delegate to the *strategy* all the abstract methods defined by `PSOAlgorithm` except `updateMemories`. In this method, the *algorithm* decides whether the new position is better than the previous one using the comparator of solutions created by the specific *strategy*. During the configuration process, this class checks that the *strategy* is an instance of `MOPSOStrategy`.

4.3 Package net.sf.jclec.mo.command

In JCLEC-MO, a *command* represents a recurrent operation that can be invoked by different elements of an experiment. Figure 4.3 shows the set of currently available *commands*.

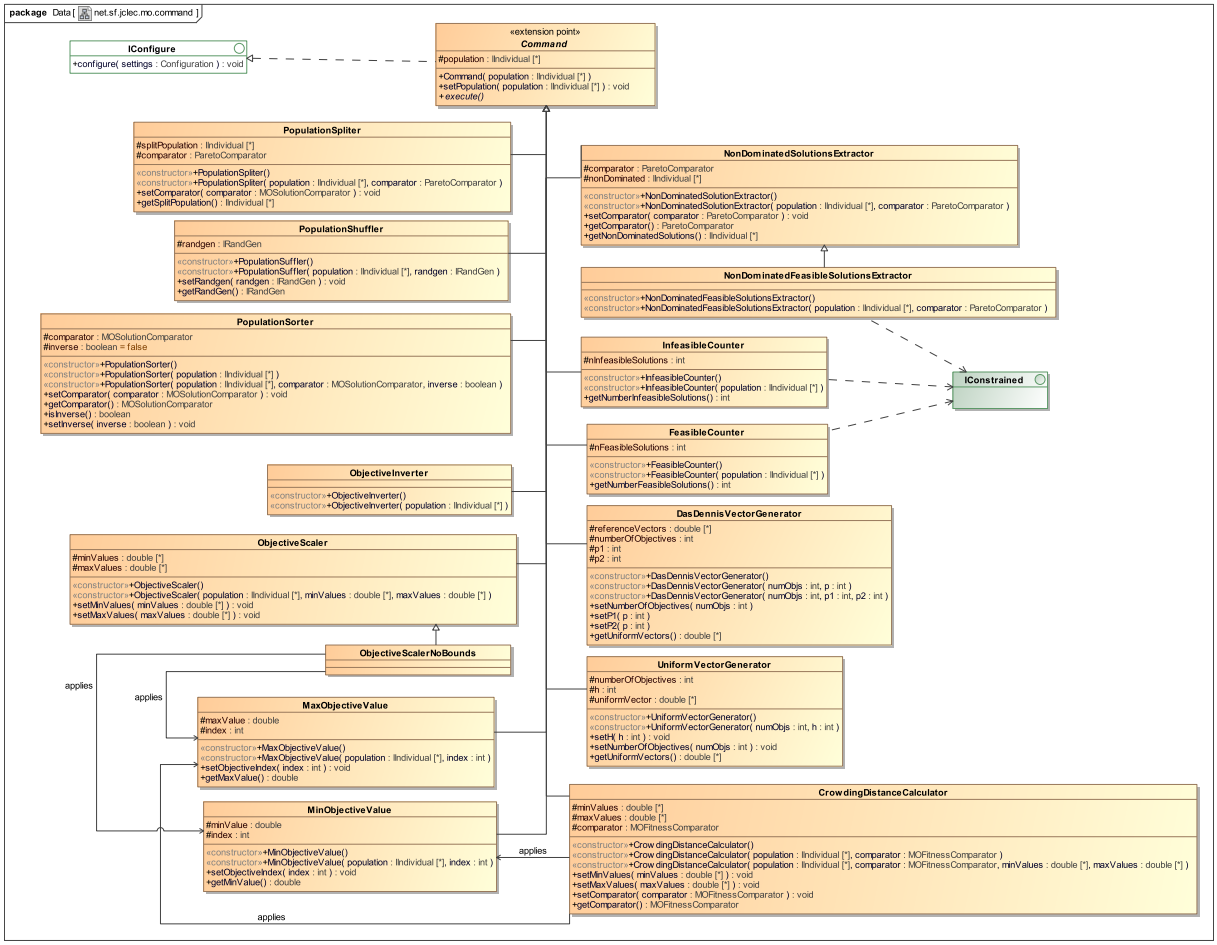

Figure 4.3: Class diagram: package net.sf.jclec.mo.command

The abstract class `Command` defines the general behaviour of this kind of operations, following the design pattern with the same name. The `execute` method performs a specific operation over a set of individuals (`population`), usually altering their properties.

JCLEC-MO provides subclasses that implement operations related to the objective functions, e.g. calculate bounds, scale and invert values, and other that manages the entire population in order to extract subsets, e.g. non-dominated solutions. On the one hand, most of the objective transformations directly modify the objective values of the solutions, not being a reversible operation. Only `MaxObjectiveValue` and `MinObjective` represent operations whose result does not imply the modification of the given population. In these cases, the result can be accessed using a *getter* method. On the other hand, `PopulationSpliter` and `NonDominatedSolutionExtractor` do not alter the population, since they are aimed at generating new sets of individuals according to different criteria. `PopulationSpliter` divides the population into fronts considering a Pareto dominance principle, which is represented by a `ParetoComparator`. `NonDominatedSolutionsExtractor` extracts a subset of the given population, which will be comprised of the non-dominated solutions according to the configured dominance criterion. In both cases, the result should

be retrieved, after invoking the `execute` method, using the corresponding *getter* method. `NonDominatedFeasibleSolutionsExtractor` is a variant that considers that only non-dominated individuals representing feasible solutions should be extracted, so it will make use of the `IConstrained` interface methods in order to know the sort of solutions encoded. In all these cases, the user can specify the *comparator* that has to be used.

Given that *commands* can constitute a part of the configuration of an experiment, subclasses should define empty constructors. In addition, *getters* methods have been defined to provide access to the results, whilst the inclusion of *setters* is aimed at allowing the configuration of parameters after its creation. This is not only useful when objects have been created using empty constructors, but also to reuse them along an iterative process, as frequently happens in search algorithms. All these circumstances imply that the design pattern has been slightly modified in order to adapt it to the JCLEC-MO requirements.

4.4 Package net.sf.jclec.mo.comparator

Comparisons between solutions are operations frequently performed during a search algorithm. In JCLEC, objects implementing the *Comparator*$<T>$ Java interface are included, where T can represent individuals (`IIndividual`) or fitness (`IFitness`) objects. JCLEC-MO follows the same idea, although the added *comparators* internally assume restrictions regarding the type of objects to be compared, e.g. `MOFitness` is expected as the more generic class to implement a fitness object. In addition, a clear relationship between a *comparator* of solutions and a *comparator* of fitness objects has been established. As can be seen in Figure 4.4, every `MOSolutionComparator` contains a `MOFitnessComparator`. Notice that the former has not been declared as an abstract class. It means that `MOSolutionComparator` implements the most general behaviour by default, i.e. solutions should be compared regarding their fitness objects, so it simply invokes its `MOFitnessComparator`.

Subclasses of `MOSolutionComparator` implement different types of comparisons that can be performed before comparing fitness values or even replacing that kind of operation. For instance, a search process might require a comparison method in which only one objective function should be considered (`ComparatorByObjectives`) or a method that can differentiate between feasible and infeasible solutions (`ConstrainedComparator`). *Comparators* can also be implemented to compare other properties defined by specific *strategies*. In these cases, they should only make use of the corresponding interface operations, e.g. `ICrowdingDistanceMOFitness` and `IHypercubeMOFitness`.

`ParetoComparator`, `EpsilonDominanceComparator` and `LexicographicComparator` are subclasses of `MOFitnessComparator` that use the set of objective values to compare solutions regarding the Pareto principle, the ϵ-dominance and the lexicographic ordering,

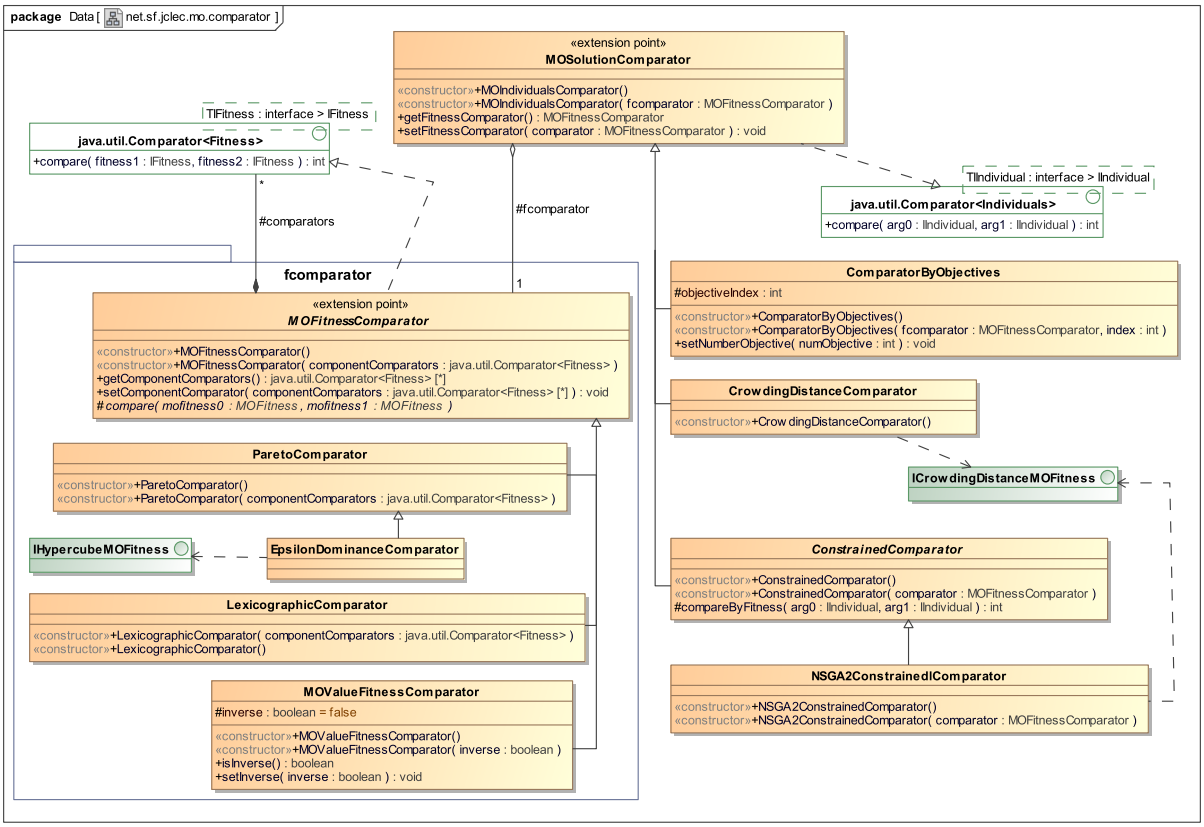

Figure 4.4: Class diagram: package net.sf.jclec.mo.comparator

respectively. On the contrary, `MOValueFitnessComparator` only considers the double value stored in `MOFitness`. This *comparator* includes a flag to determine whether the value should be maximised (`inverse=false`) or minimised (`inverse=true`).

Notice that both empty and parameterised constructors are defined. The formers are required in order to create objects using the Java reflection mechanism, as happens when classes are specified in the configuration file. In these cases, the developer is responsible of assigning the needed properties before making any comparison. To do this, *getter* and *setter* methods are provided. In other cases, parametrised constructors are recommended to guarantee that the *comparators* are properly created.

4.5 Package net.sf.jclec.mo.evaluation

Frequently, MOEAs define some properties to evaluate the quality of the solutions beyond their objective values. For instance, NSGA-II assigns a ranking front considering the dominance among solutions and computes a crowding distance to promote diversity. The values for these properties belong to each specific individual, but the properties are distinct

depending on the multi-objective approach. In JCLEC-MO, this information is stored in the fitness object, extending `MOFitness` when required. Figure 4.5 shows the definition of `MOFitness` and the set of already available subclasses. As can be seen, `MOFitness` includes the objective values, an additional `double` value that might be used to assign a unique value to the individual, e.g. the aggregated value defined by SPEA2, and a boolean value to indicate whether the fitness value is acceptable for the problem at hand.

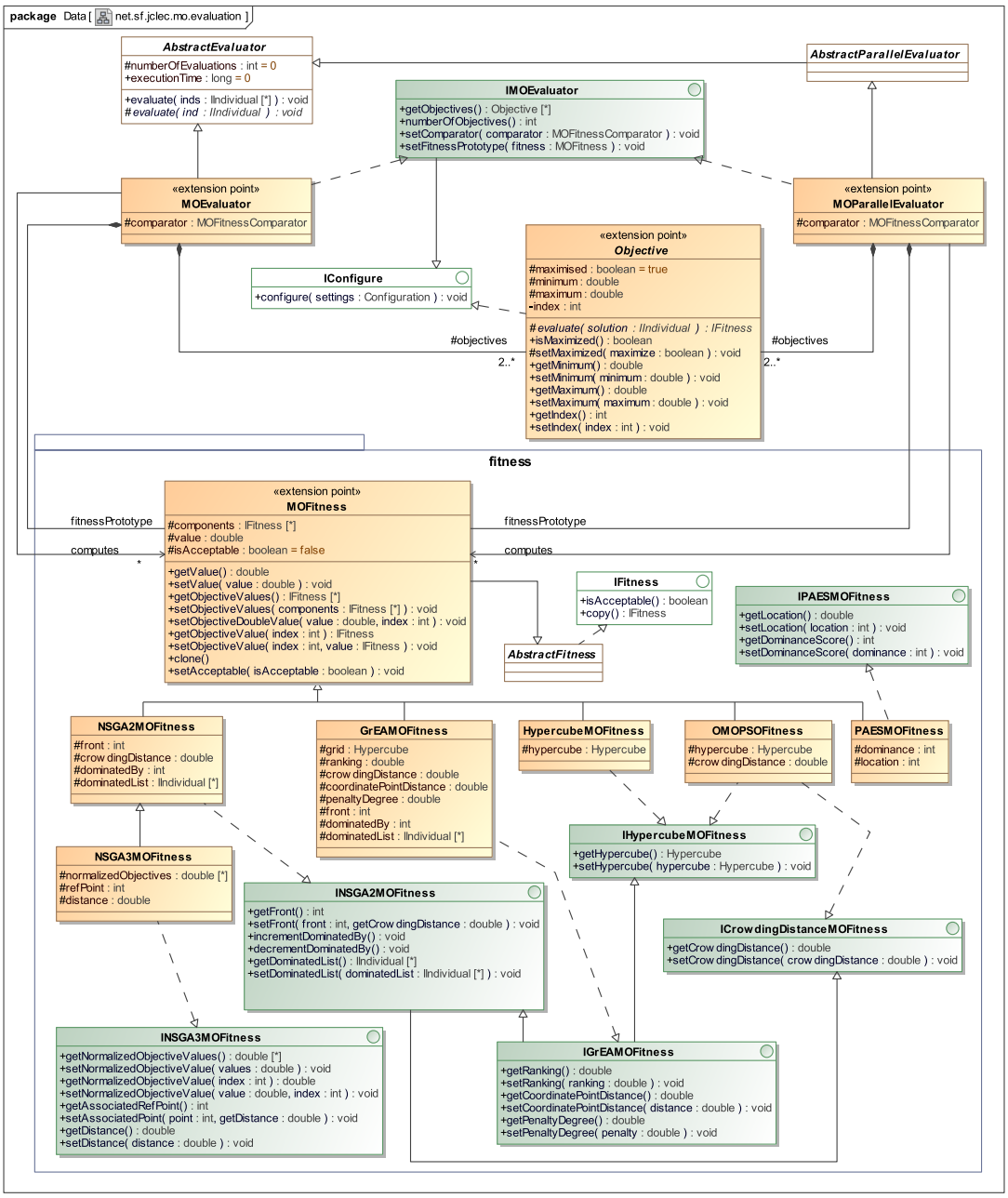

Figure 4.5: Class diagram: package net.sf.jclec.mo.evaluation

An important aspect is that `MOFitness` is an implementation of the *Prototype* design pattern, which allows not assuming the type of fitness object that the *evaluator* should use. More specifically, `MOFitness` implements the `Cloneable` Java interface, so new fitness objects will be created as copies of a given prototype, which might correspond to the generic class, a concrete subclass or even a new subclass defined by the developer. The *strategy* might impose some restrictions to the fitness object to be used, so it is the component in charge of creating the fitness prototype from the information contained in the configuration file (see lines 5-16 in the code fragment shown below). The created prototype is stored as part of the context (line 22), so it will be available to both the *algorithm* and the *evaluator*. If the fitness object is not specified in the configuration file, the generic class will be used (lines 18-20).

```
1  public abstract class MOStrategy implements IConfigure {
2      ...
3      public void configure(Configuration settings) {
4          // Fitness class name
5          String fitnessClassName = settings.getString("fitness[@type]");
6          MOFitness fitness = null;
7          // A specific fitness object must be used
8          if(fitnessClassName != null){
9              Class<? extends MOFitness> objClass;
10             try {
11                 objClass = (Class<? extends MOFitness>) Class.forName(fitnessClassName);
12                 fitness = objClass.newInstance();
13             }catch(ClassNotFoundException | InstantiationException | IllegalAccessException e){
14             throw new ConfigurationRuntimeException("Problems creating an instance of the fitness");
15             }
16         }
17         // The default fitness object
18         else{
19             fitness = new MOFitness();
20         }
21         // Store the fitness object in the evolution context
22         getContext().setFitnessPrototype(fitness);
23     }
24 }
```

During its configuration, the *evaluator* can receive the fitness prototype to be used, because every *evaluator* should implement the method `setFitnessPrototype` declared by the interface `IMOEvaluator`. When the evaluation method is invoked (see the code snapshot below), new instances of the fitness object can be created simply by cloning the prototype (lines 10-14). Although the `evaluate` method only has to store the objective values by invoking the configured objective functions (lines 16-20), this mechanism guarantees that the fitness structure assigned to every individual will be compatible to the one expected by the *strategy* (lines 22-24).

```
1  public class MOEvaluator extends AbstractEvaluator implements IMOEvaluator {
2      /** List of objectives to evaluate */
3      protected List<Objective> objectives;
```

```
4     /** The fitness object that has to be assigned to the individuals */
5     protected MOFitness fitnessPrototype;
6     ...
7     protected void evaluate(IIndividual solution) {
8         // Create an empty fitness
9         MOFitness fitness = null;
10        try {
11            fitness = (MOFitness)fitnessPrototype.clone();
12        } catch (CloneNotSupportedException e) {
13            e.printStackTrace();
14        }
15        // Evaluate the individual for each objective
16        int nObj = numberOfObjectives();
17        IFitness [] components = new IFitness[nObj];
18        for(int i=0; i<nObj; i++){
19            components[i] = this.objectives.get(i).evaluate(ind);
20        }
21        // Set components (objective values) in the fitness
22        fitness.setObjectiveValues(components);
23        // Set the fitness in the individual
24        solution.setFitness(fitness);
25    }
26 }
```

In JCLEC-MO, solutions are evaluated just after its creation and before invoking any method of the configured *strategy*. It guarantees that every solution has a fitness object that can be accessed by the *strategy*, if required. Nevertheless, to avoid a strong dependency between the *strategy* and the concrete class implementing the fitness prototype, the *strategy* only access to the fitness of a solution by means of interface methods. As an example, the code fragment below shows a part of the fast non-dominating sorting method implemented in `NSGA2`. Assuming that the dominance between all the solutions has been established (lines 5-7), the number of solutions that dominate each member of the population is checked (line 8), and those solutions that are non-dominated are assigned to the first front (lines 9-10).

```
1  public class NSGA2 extends MOStrategy {
2      public List<List<IIndividual>> fastNonDominatedSorting(List<IIndividual> population) {
3          List<List<IIndividual>> populationByFronts = new ArrayList<List<IIndividual>>();
4          int size = population.size();
5          ...
6          for(int i=0; i<size; i++) {
7              ...
8              if (((INSGA2MOFitness)population.get(i).getFitness()).getDominatedBy() == 0) {
9                  ((INSGA2MOFitness)population.get(i).getFitness()).setFront(1);
10                 populationByFronts.get(0).add(population.get(i));
11             }
12     }
13     ...
14     return populationByFronts;
15     }
16 }
```

Benefits of this approach are twofold. Firstly, the programmer can modify or extend the fitness object according to their needs, and the *strategy* will still work. This implies that variants can be created by combining methods of different *strategies*, since the developer only has to ensure that the fitness object stored in the solutions implements the expected interfaces. Secondly, concrete fitness objects can include properties usually defined by different families of algorithms. An illustrative example is GrEA (see Figure 4.5), which is based on NSGA-II but also incorporates a landscape partition technique. JCLEC-MO defines a class, named `Hypercube`, to store the position of an individual after dividing the objective space, which has been easily combined with the NSGA-II properties and additional grid measures defined by GrEA, resulting in the `GrEAMOFitness` class.

4.6 Package net.sf.jclec.mo.experiment

This package contains classes aimed at creating, executing and analysing JCLEC-MO experiments (see Figure 4.6). `MOExperiment` allows the management of all the configuration files comprising an experiment, whereas `MOExperimentRunner` receives either an experiment or a unique configuration in order to execute it. `MOExperimentHandler` represent a post-processing step that the programmer can use to define its own analysis under the *Chain of responsibility* design pattern. Since these operations are proposed to process the algorithm outcomes as generated by JCLEC-MO *reporters*, they assume data formats and file locations. In addition, it should be noted that the datapro4j library is used to load and manage all these files, so the use of *handlers* requires setting this external dependency in the classpath. The list of currently available *handlers* is detailed next:

- `ApplyStatisticalTest`. This *handler* applies statistical tests using the results in terms of quality indicators. It requires the names of the set of experiments (`experimentNames`), whose results should be stored in subfolders inside a common reporting directory (`directoryName`). If the number of algorithms is equal to 2, the Wilcoxon test is executed, otherwise, the Friedman test is performed.

- `ComputeIndicators`. This *handler* computes the given set of quality indicators (`indicators`) for all the algorithms whose results are located in a reporting directory (`directoryName`). It requires the path to the true Pareto front (PF) used by binary indicators (`referencePF`). If the PF is unknown, this parameter should be `null`, meaning that a reference PF (RFP) should be considered instead. The RPF can be automatically retrieved if `GenerateReferencePF` was previously executed. In that case, the programmer only has to specify if the RPF was generated considering the original objective values (`scaled==false`) or they were scaled in a previous step (`scaled==true`).

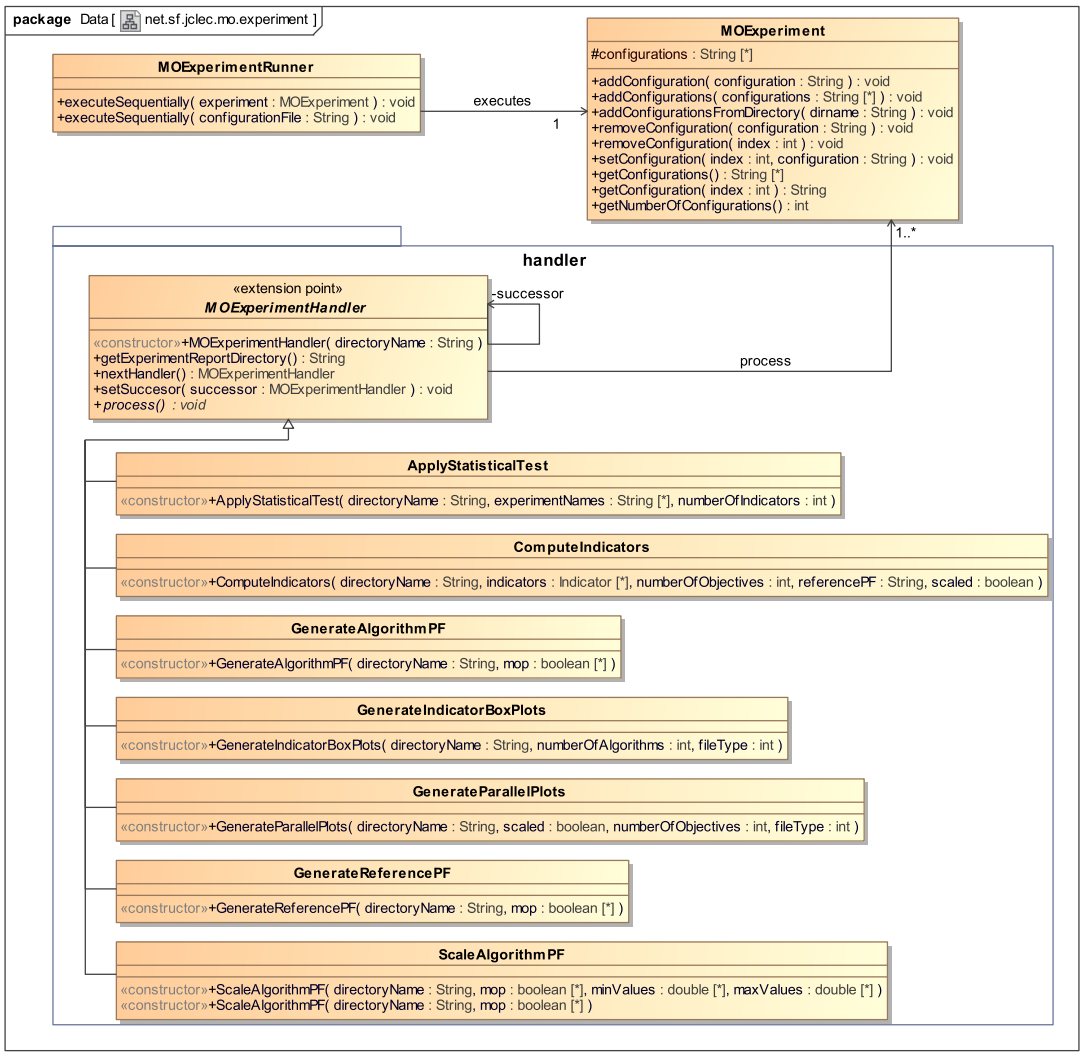

Figure 4.6: Class diagram: package net.sf.jclec.mo.experiment

- `GenerateAlgorithmPF`. This *handler* constructs a unique Pareto front from a set of fronts returned by different executions of the same algorithm. In order to check the dominance between all the solutions found, an array indicating whether the objective function at position i should be maximised (`mop[i]=true`) or minimised (`mop[i]=false`) should be specified.

- `GenerateIndicatorBoxPlots`. It creates a boxplot for each indicator computed during an experiment using the outcomes generated by `MOComparisonReporter`. The graphic will be saved in a file with the format specified in the constructor (`fileType`). Currently supported formats are: PNG (1), PDF (2), JPG (3), JPEG (4), EPS (5), PS (6), SVG (7).

- `GenerateParallelPlots`. It generates one parallel coordinates plot for each algorithm executed in an experiment. The PF represented is the one generated by

`GenerateAlgorithmPF`, either with the original objective values or after scaling them. The file format can be selected as in the previous *handler*.

- `GenerateReferencePF`. This handler constructs the RPF of an experiment from a set of fronts returned by different algorithms.

- `ScaleAlgorithmPF`. This handler scales the objective values of the solutions comprising the PFs generated by `GenerateAlgorithmPF`. The bounds of the objective functions can be specified in the constructor (`minValues` and `maxValues`). If not, they will be extracted from the given PF.

4.7 Package net.sf.jclec.mo.indicator

JCLEC-MO provides a wide set of *indicators*, each one implementing a performance measure that can be used to evaluate the quality of a Pareto front [7]. Figure 4.7 shows the hierarchy of classes that represent the types of *indicators* currently supported, i.e. unary, binary and ternary, and all the available measures. If the *indicator* is invoked along the search process, the PFs will be directly extracted from the set of solutions. Nevertheless, the measures can also be computed as part of external processes since *indicators* can load PFs from files using CSV format.

Indicators can present two pre-conditions regarding the sort of the objective space. In some cases, objective values in the range [0,1] are mandatory or, at least, recommended. In addition, some measures require all the objectives to be maximised or minimised. These requirements are controlled by two properties: `scaled` and `maximized`. If a specific *indicator* has no preference with regard to such conditions, *null* values might be assigned. It should be noted that the process invoking the *indicator* should guarantee that pre-conditions are satisfied. For instance, *reporters* in JCLEC-MO access to the aforementioned properties to check whether objective transformations are needed. Table 4.1 summarises the pre-conditions of each available *indicator*, where the symbol − is used to express that the *indicator* does not require the satisfaction of the corresponding pre-condition.

4.8 Package net.sf.jclec.mo.listener

Figure 4.8 shows the set of *reporters* provided by JCLEC-MO. `MOReporter` is a base class that can be extended in order to adapt it to the developer's needs. Despite being defined as an abstract class, `MOReporter` performs some operations during the execution of an experiment:

Table 4.1: List of available indicators and their prerequisites

Indicator	In [0,1]	Min./Max
Unary indicators		
Hypervolume (HV)	Yes	Max.
Overall Nondominated Vector Generation (ONVG)	-	-
Spacing	Preferable	-
Binary indicators		
I_ϵ	Recommended	-
$I_{\epsilon+}$	Recommended	-
Error Ratio (ER)	-	-
Spread	Recommended	Max.
Generalised Spread	Recommended	Max.
Generational Distance (GD)	Recommended	-
Inverted Generational Distance (IGD)	Recommended	-
Hyperarea Ratio (HR)	Yes	Max.
Maximum PF Error	Recommended	-
Nondominated Vector Addition (NVA)	-	-
ONVG Ratio (ONVGR)	-	-
R2 indicator	Recommended	Max.
R3 indicator	Recommended	Max.
Two Set Coverage (Coverage)	-	-
Ternary indicators		
Relative Progress	Recommended	-

1. General parameters, such as the report title and the report frequency, are retrieved from the configuration file. In this sense, any subclass inheriting from `MOReporter` should invoke the method `configure` of its super class.

2. Once the algorithm has started, `MOReporter` creates the reporting directory. Basic information about the experiment being executed will be stored in files, one per each *reporter* specified in the configuration file. Each one has its own identifier, which can be accessed using the method `getName`. In addition, some *reporters* are associated to a specific subfolder, which is automatically created inside the reporting directory.

3. After each generation, the *reporter* checks the report frequency to determine whether a new report should be generated. In such a case, the `doIterationReport` method is executed. This method simply prints the current generation (on console or on file) and then invokes the abstract method `doReport`, which should be implemented by every concrete *reporter*.

4. When the algorithm has finished, this class is in charge of closing output devices and showing the execution time.

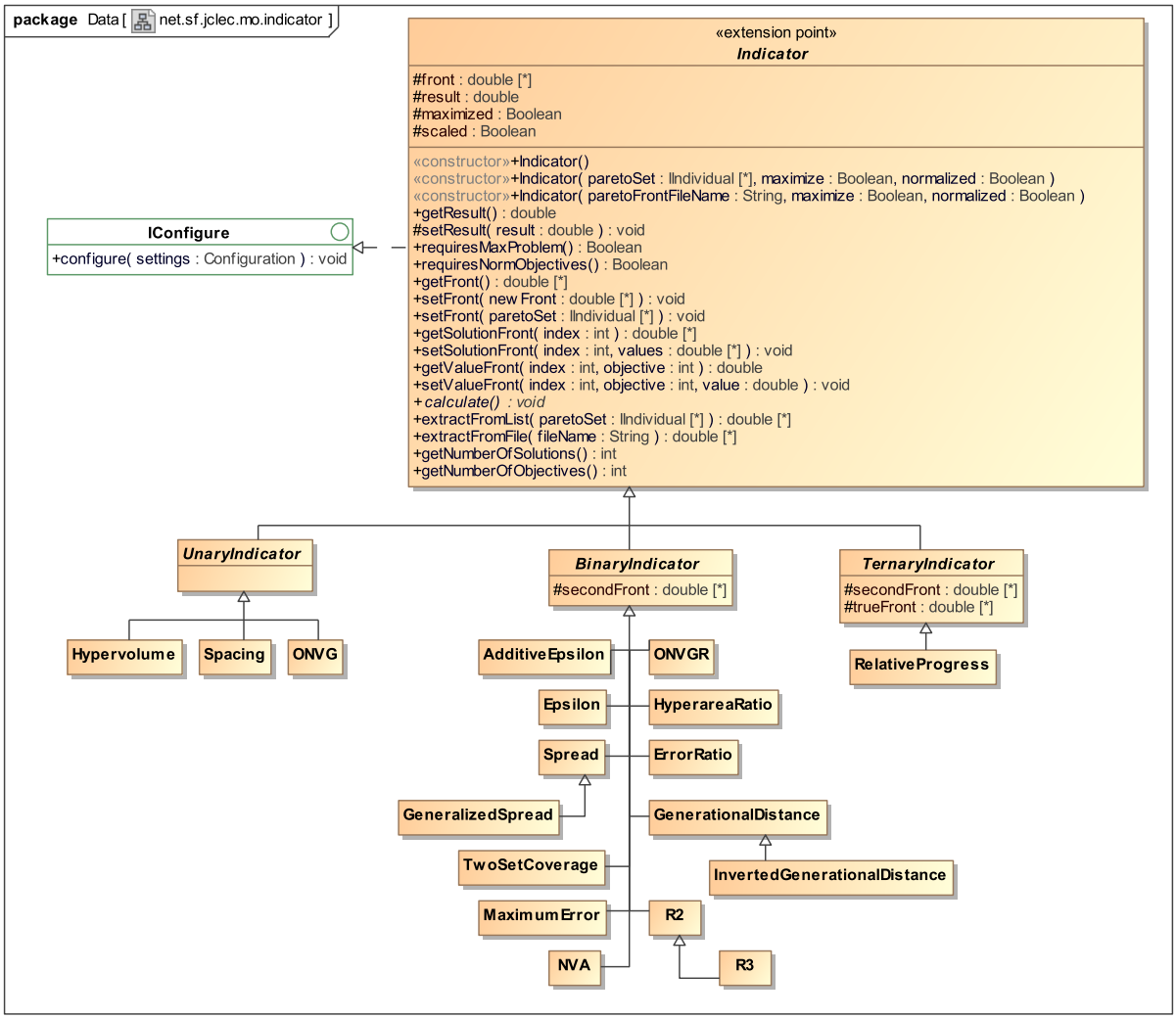

Figure 4.7: Class diagram: package net.sf.jclec.mo.indicator

Subclasses of `MOReporter` should maintain the general structure defined by the abstract class, although some methods can be carefully overridden, if required. For instance, *reporters* using *commands* should configure them before starting the execution of the algorithm. In these situations, calling the implementation of the super class before performing additional operations is highly recommended. The list of available *reporters* is given below:

- `MOPopulationReporter`. It reports the entire population and the external archive, if exists. The report files will be stored in a subfolder named "populations".

- `MOParetoFrontReporter`. This class creates a CSV file containing the current PF. To do this, it uses the datapro4j library. All the generated files are stored in a subfolder named "pareto-fronts".

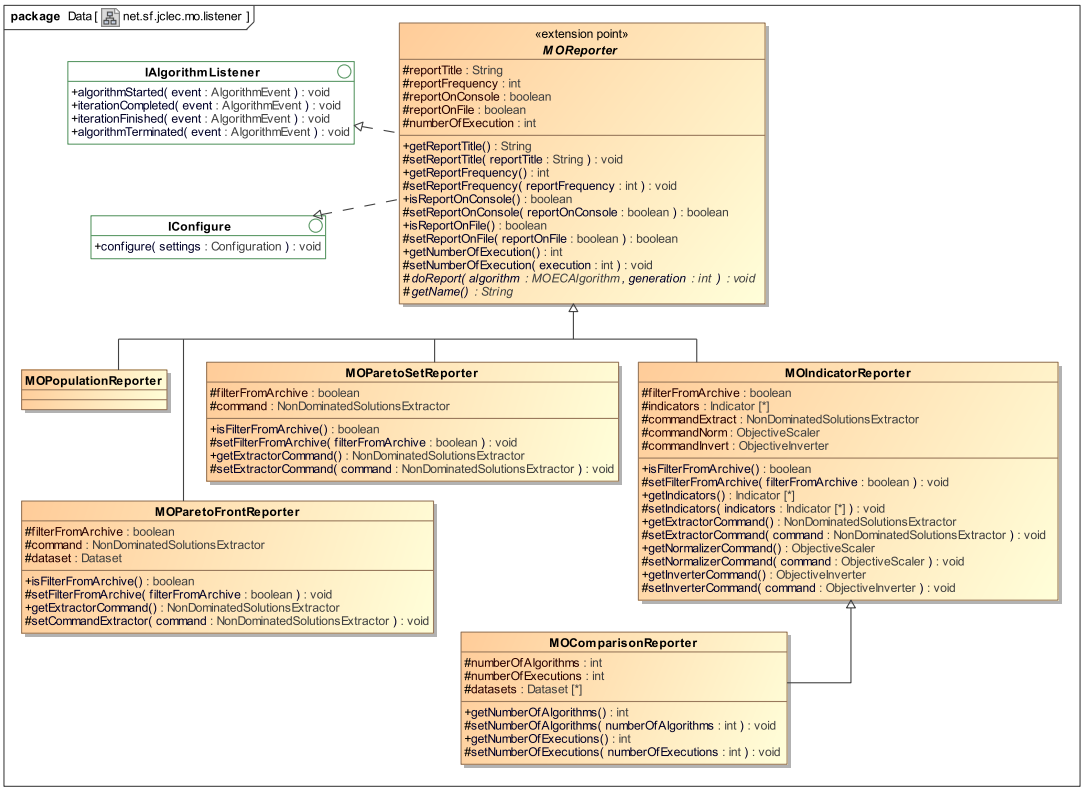

Figure 4.8: Class diagram: package net.sf.jclec.mo.listener

- `MOParetoSetReporter`. This *reporter* stores the solutions belonging to the Pareto set (in plain text) in a subfolder named "pareto-sets".
- `MOIndicatorsReporter`. A set of quality indicators can be computed using this class. The results are written to a unique file, the same that contains the general information of the report.
- `MOComparisonReporter`. It creates a file with CSV format using datapro4j, in which each column contains the result of a quality indicator for a different algorithm. This class generates one file for each measure specified in the configuration file, although all of them will be accessible from the same subfolder ("indicators").

4.9 Package net.sf.jclec.mo.strategy

Figure 4.9 shows the class diagram of the package *net.sf.jclec.mo.strategy*. Some details are explained below:

- `MOStrategy` defines a private property named *context*, which serves as the communication mechanism between the *strategy* and the *algorithm* (see Section 3.2).

- `MOPSOStrategy` specifies additional methods for MOPSO and should be used in combination with `MOPSOAlgorithm` (see Section 3.1).
- Each multi-objective algorithm is defined as a subclass of `MOStrategy`, providing concrete implementations of the abstract methods. Additional properties and operations might be required, although they are not shown in the diagram to save space. Both `IBEA` and `MOEA/D` are declared as abstract classes, since they can be customised with different mechanisms to evaluate solutions. Nevertheless, most of the implementation is provided by the corresponding abstract class.
- JCLEC-MO includes a variant of each *strategy* aimed at solving problems with constraints. These *constrained* variants assume that solutions that will be managed implement the `IConstrained` interface. This interface defines *getter* and *setter* methods to access the properties of solutions of constrained problems. The way in which properties are included in the solution relies on the programmer's criterion.

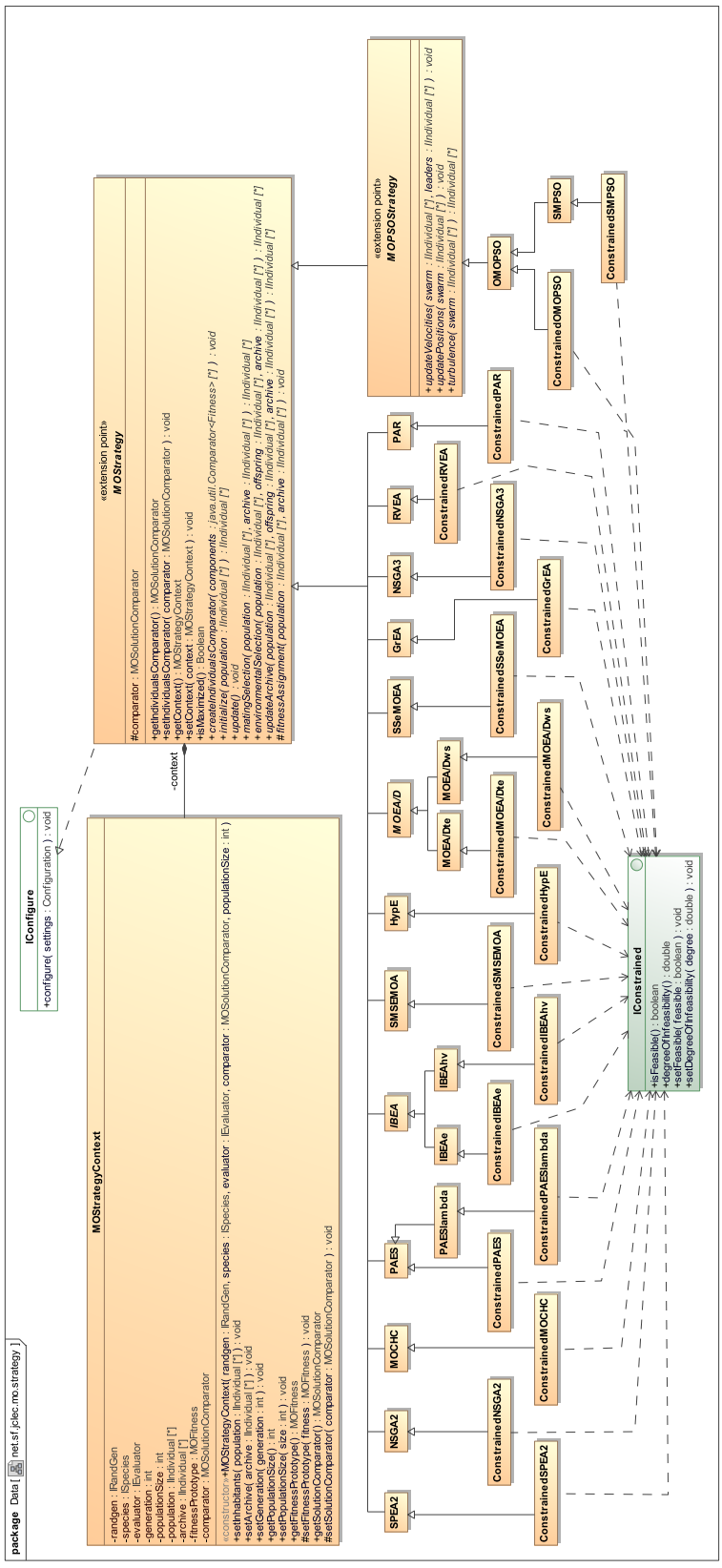

Figure 4.9: Class diagram: package net.sf.jclec.mo.strategy

REFERENCES

[1] A. Ramírez, J. R. Romero, and S. Ventura, "An Extensible JCLEC-based Solution for the Implementation of Multi-Objective Evolutionary Algorithms," in *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*, GECCO Companion '15, pp. 1085–1092, ACM, 2015.

[2] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a java framework for evolutionary computation," *Soft Computing*, vol. 12, no. 4, pp. 381–392, 2008.

[3] J. R. Romero, J. M. Luna, and S. Ventura, *datapro4j: the data processing library for Java*. Dept. of Computer Science and Numerical Analysis, University of Córdoba (Spain). Available for download from: http://www.uco.es/grupos/kdis/datapro4j, 2012.

[4] E. Gamma, R. Helm, R. Johnson, and D. A. Vlissides, J.M., *Design Patterns: Elements of Reusable Object-Oriented Software*. Prentice Hall, 1st ed., 1980.

[5] E. Zitzler, M. Laumanns, and S. Bleuler, "A Tutorial on Evolutionary Multiobjective Optimization," in *Metaheuristics for Multiobjective Optimisation*, vol. 535 of *Lecture Notes in Economics and Mathematical Systems*, pp. 3–37, Springer Berlin Heidelberg, 2004.

[6] J. J. Durillo, J. García-Nieto, A. J. Nebro, C. A. Coello Coello, F. Luna, and E. Alba, "Multi-Objective Particle Swarm Optimizers: An Experimental Comparison," in *Evolutionary Multi-Criterion Optimization*, vol. 5467 of *Lecture Notes in Computer Science*, pp. 495–509, Springer Berlin Heidelberg, 2009.

[7] C. A. Coello Coello, G. B. Lamont, and D. A. Van Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Springer, 2nd ed., 2007.