Redesigning the jMetal Multi-Objective Optimization Framework

Antonio J. Nebro
Departamento de Lenguajes y Ciencias de la Computación
Ada Byron Research Building
University of Málaga, Spain
antonio@lcc.uma.es

Juan J. Durillo
Distributed and Parallel System Group
University of Innsbruck, Austria
juan@dps.uibk.ac.at

Matthieu Vergne
Center for Information and Communication Technology, FBK-ICT, Italy
vergne@fbk.eu
Doctoral School in Information and Communication Technology, Italy
matthieu.vergne@unitn.it

ABSTRACT

jMetal, an open source, Java-based framework for multi-objective optimization with metaheuristics, has become a valuable tool for many researches in the area as well as for some industrial partners in the last ten years. Our experience using and maintaining it during that time, as well as the received comments and suggestions, have helped us improve the jMetal design and identify significant features to incorporate. This paper revisits the jMetal architecture, describing its refined new design, which relies on design patterns, principles from object-oriented design, and a better use of the Java language features to improve the quality of the code, without disregarding jMetal ever goals of simplicity, facility of use, flexibility, extensibility and portability. Among the newly incorporated features, jMetal supports live interaction with running algorithms and parallel execution of algorithms.

CCS Concepts

•Software and its engineering → Software libraries and repositories; •Computing methodologies → *Heuristic function construction;*

Keywords

jMetal; Optimization Framework; Multi-Objective Metaheuristics; Open Source

1. INTRODUCTION

Multi-objective optimization with metaheuristics is an active research area since early 2000, when algorithms that rapidly become widely used were proposed (NSGA-II [3], SPEA2 [16], PAES [10]) and the books about evolutionary algorithms for solving multi-objective problems of K. Deb [9] and C. Coello *et al* [2] were published.

The availability of software tools, including state-of-the-art metaheuristics, has become an important factor in the field of multi-objective optimization for promoting (1) research of new techniques and (2) the adoption of these algorithms in real scenarios. One of these tools is jMetal [5][4][6].

The jMetal project started in 2006 to cover our needs for researching in multi-objective optimization with metaheuristics. The absence at that time of an easy-to-use, flexible, extensible and portable software framework led us to design and develop a new tool from scratch. In 2008, jMetal was hosted on SourceForge[1], becoming freely available to the research community for multi-objective optimization. Since then, jMetal has become a popular software in the field [8] and our three papers describing the jMetal framework [5][4][6] sum up more than 500 citations at the time of writing this paper[2].

The object-oriented architecture of jMetal and its code readability are very appreciated features for many users who have provided feedback about the tool. However, as regular jMetal users, we have detected several issues that could be improved in its nine years old design, many of which have also been pointed out by other users. For this reason, we have taken the decision of carrying out the first major revision of jMetal since its initial version. The first step in this direction has been to switch to GitHub for the versioning of the code[3]. This has allowed us to get very valuable ideas from other people who have notably influenced the new release architecture.

In this paper, we analyse first those relevant aspects of jMetal that we have reconsidered (Section 2) and then we describe how we have dealt with them (Section 3). Finally, the conclusions of the paper are summarized in Section 4.

We have to note that at the time of writing the paper, the new jMetal version is not finished yet, so what we present here is a work in progress. The next major release, jMetal 5.0, is expected to be finished by the end of July 2015.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

[1]jMetal on SourceForge: http://jmetal.sourceforge.net
[2]Source: Google Scholar
[3]jMetal on GitHub: https://github.com/jMetal/jMetal

2. RELEVANT ISSUES

A critical review of jMetal identified some issues in our core class hierarchy that make some recurrent actions, such as creating new types or operators, not as simple as it would be desired. Although one of the original goals of jMetal was to pay attention to code quality, there is room for improvement here, for example the implementation of unit tests. Another point is the project delivery strategy consisting in only providing the source code, which does not facilitate the use of jMetal as a dependency in other projects. In this section we analyse several points to elaborate on these ideas.

2.1 Too Many Entities to Represent Solutions

Solution encoding is a fundamental aspect when solving an optimization problem with a metaheuristic because it determines the variation operators that can be applied. Encoding is managed in jMetal through three classes: `Solution`, `SolutionType` and `Variable` (the architecture of the last jMetal release, version 4.5, is depicted in Figure 1). This way, a solution has a type and it is composed of a number of variables. The solution type is used to determine which operators are available for manipulating the solutions, and the variables are the containers of the solution values. If we consider evolutionary algorithms, a solution represents an individual, the variables constitute the chromosome, and each variable is a gene.

In [7] we studied the behaviour of several state-of-the-art multi-objective metaheuristics when solving parameter scalable problems. The targets were the continuous problems of the ZDT benchmark [15]. In this study we addressed problems having up to 2048 `Real` variables, meaning that the solutions were composed of 2048 `Variable` objects in some experiments, thefore introducing a significant performance overhead. We introduced then an "optimized encoding" called `ArrayReal`, in such a way that a real solution is composed of a single variable which contains an array of real values. This scheme showed more efficiency, but it led to two ways of using real coded solutions. To simplify the management of this duplication, we used a wrapper class called `XReal`. As a consequence, up to five classes can be used to work with real coded solutions: `Solution`, `Real` (variable type), `RealSolutionType`, `ArrayReal` and `XReal`. A simpler scheme would be desirable.

Another consequence of our encoding was that, to access the value of its i^{th} variable, a solution s requires to use the following statement:

```
value = s.getDecisionVariables()[i].getValue()
```

which is rather cumbersome. It should be as simple as getting the value of the j^{th} objective:

```
value = s.getObjective(j)
```

2.2 Non-SOLID Core Classes

According to the SOLID principles [13], a class should fulfil five principles:

- Single Responsibility Principle (SRP): a class should have responsibility over a single part of the functionality provided by the software, and this responsibility should be entirely encapsulated by the class.

- Open-Close Principle (OCP): a class must be open to extensions and closed to modification.

- Liskov Substitution Principle (LSP): derived classes must be substitutable for their base classes.

- Interface Segregation Principle (ISP): no client should be forced to depend on methods or elements it does not use.

- Dependency inversion principle (DIP): classes must depend on abstractions, not on implementations.

One of the classes at the core of jMetal, the `Solution` class, does not fulfil some of these principles. Besides including the data that every solution must have, it includes numerous fields which are algorithm dependent (SRP violation), such as fields to store the ranking and the crowding distance of a solution, required by the NSGA-II algorithm but not used by algorithms such as PAES or SPEA2 (violation of the ISP). Similarly, a strength variable is included for the only need of the SPEA2 algorithm, and so on. In fact, the way to proceed every time a new metaheuristic is to be included consists in modifying the `Solution` class according to the needs of that algorithm (violation of the OCP), giving as a result a class populated with many fields.

The `Solution` class is not the only one violating these principles. The `Problem` class contains fields such as lower and upper bound, which are only needed by continuous problems, or a `getNumberOfBits()` method, which is only useful for problems that work with binary representations.

2.3 Unrelated Solution Types, Operators and Problems

Every optimization problem in jMetal is associated with a particular encoding, which is indicated by a `SolutionType` object. For example, continuous problems require real coded solutions, and combinatorial problems require other solution types such as binary types (like the OneMax problem) or permutations (such as the travelling salesman problem or *TSP*). The solution type is used to determine the variation operators that can be applied to a solution. For example, single point and BLX-α are crossover operators which should work only with binary and real representations, respectively.

In jMetal 4.5 and previous versions, these relations among different entities in the framework are hard coded and runtime checks are needed to validate the applicability of an operator to a given solution based on its type. As an example, the following code shows the constructor of the classical Schaffer problem. The problem constructor receives a string as parameter to indicate if the problem is continuous (`Real`), discrete (`Integer`) or of any other type. As we can see, an `if` statement is used to ensure that the solution type is correctly set:

```
public Schaffer(String solutionType) {
  numberOfVariables_   = 1;
  numberOfObjectives_  = 2;
  numberOfConstraints_ = 0;
  problemName_         = "Schaffer";

  lowerLimit_ = new double[numberOfVariables_];
  upperLimit_ = new double[numberOfVariables_];
  lowerLimit_[0] = -100000;
  upperLimit_[0] =  100000;

  if (solutionType.compareTo("Real") == 0)
    solutionType_ = new RealSolutionType(this) ;
  else {
    System.out.println("Error: solution type ") ;
    System.out.println("" + solutionType + " invalid") ;
```

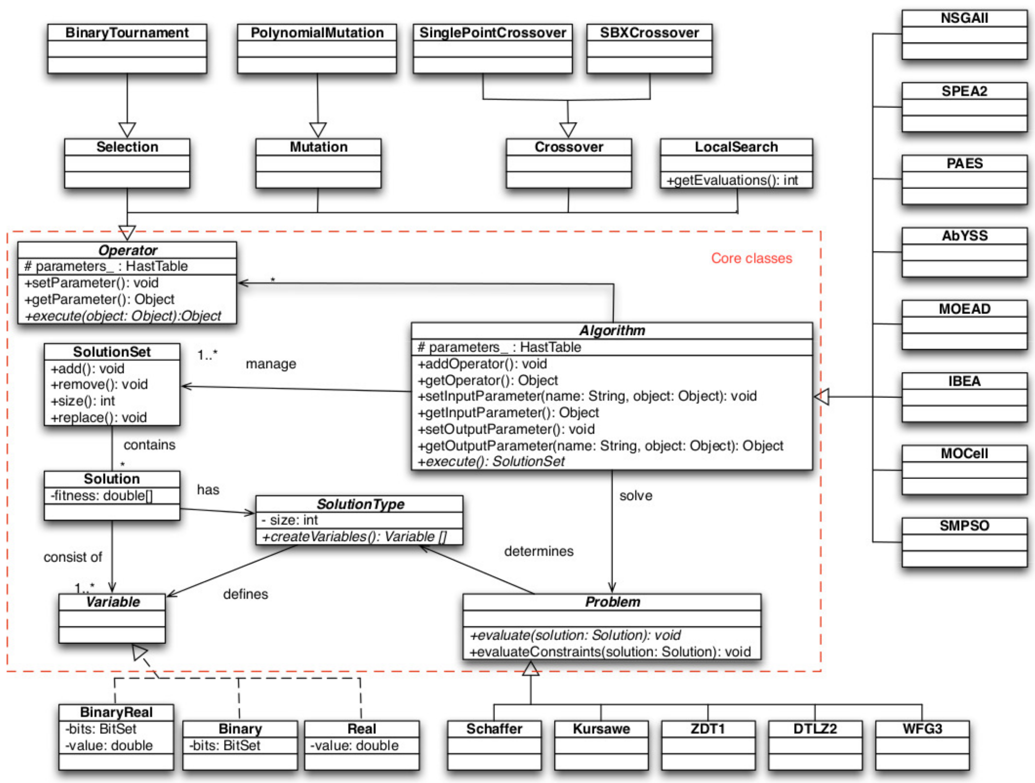

Figure 1: jMetal 4.5 architecture

```
    System.exit(-1) ;
  }
} // Schaffer
```

Part of the code of the `PolynomialMutation` class, which implements a mutation operator for continuous problems such as Schaffer, is included in the following lines. It can be also seen that this code checks whether the type of a solution is adequate for this operator.

```
public class PolynomialMutation extends Mutation {
  ...
  private static final List VALID_TYPES =
     Arrays.asList(RealSolutionType.class) ;
  ...
  public Object execute(Object object) {
    Solution solution = (Solution)object;
    if (!VALID_TYPES.contains(solution.
      getType().getClass())) {
        ... // error message
    } // if

  doMutation(mutationProbability_, solution);
  return solution;
} // execute
```

Such programming practice works, but it is error prone, so the execution of an algorithm may be interrupted at runtime if the solution type received by an operator is not the expected one.

2.4 Code Replication and Algorithm Templates

Most of metaheuristic families are characterized by a common behaviour which is shared by all the algorithms belonging to the family. This behaviour can be expressed as a pseudo-code that can be seen as a template. An example is shown in Algorithm 1, which represents Evolutionary Algorithms (EAs),

Algorithm 1 Pseudo-code of an evolutionary algorithm

```
1: $P(0) \leftarrow$ GenerateInitialSolutions()
2: $t \leftarrow 0$
3: Evaluate($P(0)$)
4: while not StoppingCriterion() do
5:    $Q(t) \leftarrow$ Variation($P(t)$)
6:    Evaluate($Q(t)$)
7:    $P(t+1) \leftarrow$ Update($P(t)$, $Q(t)$)
8:    $t \leftarrow t+1$
9: end while
```

Having a template for a basic EA would lead researches to focus on which part of a new EA are novel and what parts already exist. From the software engineering point of view, an algorithm whose behavior falls in a basic template would only require to implement some specific methods for the new technique (in the case of genetic algorithms, the recombination, selection, etc); the common behavior would not be needed to be programmed, therefore resulting in less code replication. jMetal does not contain any set of basic templates, so the users do not benefit of the aforementioned advantages.

2.5 No Run Time Information Support

By default, when the execution of an algorithm is launched (by calling its `execute()` method), the calling program has to wait until the method has finished to get the computed result. During this period of time, little or no information about the algorithm execution is provided. While this may be acceptable for executions that take a few seconds (even minutes), this behaviour is non-desirable for long execution that take hours or days to complete. In these cases, it may be necessary to be able to retrieve some information (such as the current iteration, the current set of non-dominated solutions, etc.) at any point of its execution in order to take further decisions.

In the previous version of jMetal, for obtaining such information during an algorithm execution, the only choice was to manually add log sentences by modifying the source code. This is an unattractive approach, especially if we are using several metaheuristics (e.g., in a comparative study). Frequently, these monitoring sentences have to be removed or, even worse, commented, leading to many commented lines of code.

2.6 Code Quality

One of our goals, when developing jMetal, was to pay attention to the quality of the code, a concern which has been well appreciated by many users who have provided feedback about our tool. However, our criteria to qualify the code quality were fuzzy and mainly based on our intuition. Indeed, the application of specific tools to measure the quality of code, such as SonarQube[4], revealed many issues to improve, such as:

- `System.exit()` is widely used when an error situation is found. This sentence shuts down the Java virtual machine, which is a very undesirable effect if jMetal is included into other software.
- Some methods have many lines of code, making them hard to read and understand and hinder to effectively apply unit tests.
- Many commented-out lines of code.
- Many names of fields, constants, methods, and classes do not comply with a naming convention.

We do not use unit nor integration testing in jMetal, so the possibility that some errors remain undetected is higher. For example, the behaviour of variation operators, like the polynomial mutation or SBX crossover applied by many algorithms, has not been tested to ensure that they are correctly implemented.

2.7 Source Code Delivery

jMetal is distributed by providing the source code through SourceForge. This means that, if a new version is released, a project using it has to be updated manually. jMetal 4.5 is also delivered as a jar file, but the problem is the same. With the purpose of simplifying the use of jMetal by other researchers, we did not use any external dependence. Therefore, we did not use packages of third parties that would undoubtedly help to offer more functionalities and accelerate the jMetal development.

[4]SonarQube: http://www.sonarqube.org

2.8 Weak Support for Parallelism

jMetal includes parallel variants of some metaheuristics, like pNSGAII, pSMPSO and pMOEAD. These versions are thread-based implementation of these algorithms built on top of the Java thread facilities. They are only intended to run on multi-core processors and are not usable to develop distributed versions of the algorithms.

3. REDESIGNING JMETAL

After analysing those issues that could be improved in jMetal, we detail in this section how we cope with them in the forthcoming jMetal 5.0 version.

3.1 New Architecture

The main goal of the new architecture is to provide a simpler design while keeping the same functionality as before. Figure 2 depicts the core classes of this architecture. This diagram, composed of only four interfaces, captures the typical functionality provided by jMetal: an `Algorithm` solves a `Problem` by manipulating a set of potential `Solution` objects through the use of several `Operators`.

Compared with the former architecture (see Figure 1), the new design introduces several obvious differences. First, the number of core classes has been reduced from seven to four. Second, all these core classes are now interfaces, and most of them have only few methods. Finally, we can observe the use of parametrized types to model the use of Java generics, which are now widely applied.

The absence of the `SolutionSet` class, that was used before for representing populations in EAs (or swarms in PSO), may be a bit surprising; however, our analysis revealed that its role can be fulfilled by a list of solutions, and therefore it can be represented by the `java.util.List` interface. The main advantage of using Lists for this purpose is that Java provides already several implementations with different performance and characteristics (arrays, linked list, etc).

3.2 Representing Solutions

In jMetal 5.0, there is no need for using the concepts of solution types nor variables to encode a solution. The former is not required thanks to the use of generics. Figure 3 shows how the solutions are defined now, where three representations are included: binary, real, and integer. By using this approach, many implementations can be provided for the same encoding, adding an extra degree of flexibility.

In the case of double solutions, we provide a default implementation where the variable values are stored in an array, so we avoid having to define something like the former `ArrayReal` encoding. This also eliminates the necessity of wrapper classes such as `XReal`. Now, getting the value of a variable and an objective is done in this way:

```
double value = solution.getVariableValue(i) ;
double objective = solution.getObjective(j) ;
```

which is simpler and more natural than the previous scheme commented in Section 2.1.

The use of generics also allows that an attempt to incorrectly assign the value of a variable results in a compilation error, e.g., trying to assign to an `int` variable the variable value of a `DoubleSolution`.

A feature of the new approach to represent solutions is the introduction of attributes, which are simply <key, value>

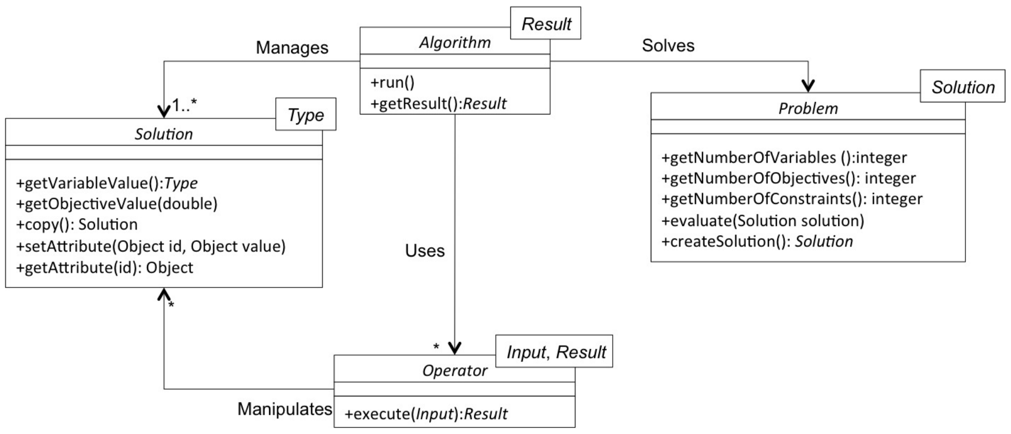
Algorithm
Result
+run()
+getResult():Result
Manages
Solves
1..*
Solution
Type
+getVariableValue():Type
+getObjectiveValue(double)
+copy(): Solution
+setAttribute(Object id, Object value)
+getAttribute(id): Object
Problem
Solution
+getNumberOfVariables ():integer
+getNumberOfObjectives(): integer
+getNumberOfConstraints(): integer
+evaluate(Solution solution)
+createSolution(): Solution
Uses
*
*
Operator
Input, Result
+execute(Input):Result
Manipulates

Figure 2: UML class diagram of jMetal 5.0 core classes.

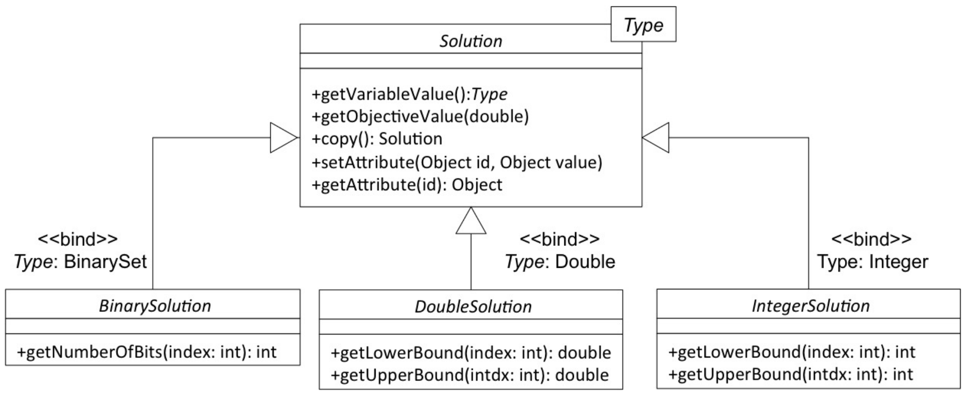
Solution
Type
+getVariableValue():Type
+getObjectiveValue(double)
+copy(): Solution
+setAttribute(Object id, Object value)
+getAttribute(id): Object
<<bind>>
Type: BinarySet
<<bind>>
Type: Double
<<bind>>
Type: Integer
BinarySolution
+getNumberOfBits(index: int): int
DoubleSolution
+getLowerBound(index: int): double
+getUpperBound(intdx: int): double
IntegerSolution
+getLowerBound(index: int): int
+getUpperBound(intdx: int): int

Figure 3: jMetal 5.0 Solution class diagram.

pairs. When a solution has to incorporate a field which is specific for a given algorithm, that field can be added as an attribute. This way, there is no need to modify the solution class, as commented in Section 2.2.

To avoid having to manage directly the solution attributes, we include this utility interface:

```
public interface SolutionAttribute
   <S extends Solution, V> {

  public void setAttribute(S solution, V value) ;
  public V getAttribute(S solution) ;
  public Object getAttributeID() ;
}
```

The use of solution attributes can be encapsulated. As an example, we have defined the following interface to assign a rank to a solution (i.e, NSGA-II's ranking):

```
public interface Ranking<S extends Solution>
  extends SolutionAttribute<S, Integer>{

  public Ranking computeRanking(List<S> solutionList) ;
  public List<S> getSubfront(int rank) ;
  public int getNumberOfSubfronts() ;
}
```

so a client class (e.g., the `NSGAII` class) can merely use:

```
Ranking ranking = computeRanking(jointPopulation);
```

This way, the solution attribute is managed internally by the class implementing the ranking and is hidden to the metaheuristic.

3.3 jMetal as a Maven Project

A request of some users was to make jMetal a Maven project[5]. This way it would be easier to incorporate it in any Maven project and also to use dependencies of third parties.

Our approach has been to partition jMetal into four packages:

- `jmetal-core`: Classes of the core architecture plus some utilities, including quality indicators.
- `jmetal-algorithm`: Implementations of the metaheuristics included in the framework.
- `jmetal-problem`: Implementations of the known problems.
- `jmetal-exec`: Executable programs to configure and run the algorithms.

These packages can be found in the Maven Central Repository[6], where the Maven dependencies can be obtained. For example, in the case of needing the `jmetal-core` package, the current dependence is:

```
<dependency>
    <groupId>org.uma.jmetal</groupId>
    <artifactId>jmetal-core</artifactId>
    <version>5.0-Beta-28</version>
</dependency>
```

[5]https://maven.apache.org/
[6]Maven Central Repository: http://search.maven.org/

3.4 Relating Solution, Operators, and Problems

By using generics in the new architecture, the compiler can ensure that, if an algorithm is configured to solve e.g, a continuous problem (a `DoubleProblem` in jMetal 5.0), then the operators must manipulate only `DoubleSolutions`.

Let us suppose that we want to use NSGA-II to solve a continuous problem. The configuration of the algorithm would contain the following code:

```
DoubleProblem problem;
Algorithm<List<DoubleSolution>> algorithm;
CrossoverOperator<
        List<DoubleSolution>,List<DoubleSolution>> crossover;
MutationOperator<DoubleSolution> mutation;

...

algorithm = new NSGAIIBuilder
  <DoubleSolution>(problem, crossover, mutation)
  .setSelectionOperator(selection)
  .setMaxIterations(100)
  .setPopulationSize(100)
  .build() ;
```

This way, operators such as the SBX crossover and polynomial mutation will be valid to solve a problem such as ZDT1 (continuous), but the code will not compile if there is any attempt to apply the single point crossover or bit-flip mutation binary operators, or the ZDT5 problem (binary).

Any attempt to use an operator with a non valid solution will be detected in compilation time, and advanced IDEs (e.g, Eclipse) can immediately find these errors.

3.5 Algorithm Templates

As commented in Section 2.4, providing algorithm templates can bring a number of advantages. In jMetal 5.0 we include a number of templates mimicking canonical metaheuristics. For example, the `AbstractEvolutionaryAlgorithm` class provides a template for EAs by including the following implementation of the `run()` method (`Algorithm` extends `Runnable`):

```
  @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);
      updateProgress();
    }
  }
```

To develop a new EA, the methods used by `run()` have to be implemented. This way, the flow control is in the abstract class, not in the algorithm implementation. This abstract class is the base class of genetic algorithms, evolution strategies, and differential evolution algorithms. An `AbstractParticularSwarmOptimization` class is currently provided as well.

Popular metaheuristics such as NSGA-II [3], PAES [10], SPEA2 [16] or SMS-EMOA [1] follow this template. Others, such as MOEA/D [12], cannot be implemented according to it, so they need to inherit directly from the `Algorithm` interface. Templates provide facilities to implement algorithms, but they are not mandatory.

3.6 Measures in Algorithms

A novelty in jMetal 5.0 is the inclusion of measures, which allow to obtain algorithm-specific information during its execution. The current implementation supports two types of measures: a `PullMeasure` provides the value of the measure on demand (synchronous), while a `PushMeasure` allows to register listeners (observer design pattern) to receive the value of the measure when it is produced (asynchronous).

One can transform a pull measure into a push one (e.g. with a period of refresh) as well as a push measure into a pull one (e.g. by storing the value for future demands), thus choosing one or the other is more a design choice depending on how the algorithm manage the measured values. For instance, if the value is always available in the algorithm (e.g. population size), then it is usual to use a pull measure to retrieve it on demand, while a value generated on the fly (e.g. solution evaluation) is prone to be managed by a push measure.

Finally, these measures are provided by the algorithm through a `MeasureManager`, which acts as a mapper between the available measures and keys which identify each of them. The measure manager provides all the available keys, allowing to retrieve and manage all the available measures for further automation.

To illustrate how measures work, jMetal 5.0 provides a version of NSGA-II endowed with pull and push measures. The next code shows an example of measures implementing both features and how they are used:

```
public class NSGAIIMeasures extends NSGAII {
  private CountingMeasure iterations ;
  private BasicMeasure<List<Solution>> lastEvaluatedPopulation ;

  ...

  iterations = new CountingMeasure(0) ;
  lastEvaluatedPopulation = new BasicMeasure<>() ;

  public NSGAIIMeasures(...) {
    ...
    measureManager = new SimpleMeasureManager() ;
    measureManager.setPullMeasure("currentIteration",
                                  iterations);
    measureManager.setPushMeasure("lastEvaluatedPopulation",
                                  lastEvaluatedPopulation);
  }

  @Override protected void initProgress() {
    iterations.reset(1);
  }

  @Override protected void updateProgress() {
    iterations.increment();
    lastEvaluatedPopulation.set(getPopulation());
  }

  ...
}
```

In this example, our algorithm declares a `CountingMeasure` (a measure intended to count events) called `iterations` and aiming at counting how many iterations has been made since the starting of the algorithm. Another one is a `BasicMeasure` (simply stores a value) called `lastEvaluatedPopulation` and aiming at providing the population considered at the evaluation time, thus giving a way to know how the population evolve between each round. The point with push measures is when to effectively push the value to the listeners, thus identifying the precise location where the value changes. Here, the iteration counting is reset at every call of `initProgress()`, which corresponds to a restart of the algorithm, and both measures are updated in `updateProgress()`, which is called at the end of each iteration (notice that these two methods are required by the evolutionary algorithm template defined in Section 3.5). Regarding the `MeasureManager`, we register both the measures as pull or push measures depending on how we want them to be used out of the algorithm. The client code can make use of these measures as follows:

```
...
 Algorithm<List<Solution>> algorithm;
 ...
  MeasureManager measureManager =
     algorithm.getMeasureManager() ;

  CountingMeasure iteration =
    (CountingMeasure) measureManager.getPullMeasure(
                            "currentIteration");
    BasicMeasure<List<Solution>> lastEvaluatedPopulationMeasure
        = (BasicMeasure) measureManager.getPushMeasure(
                            "lastEvaluatedPopulation");

    lastEvaluatedPopulationMeasure(new Listener());

    Thread algorithmThread = new Thread(algorithm) ;
    algorithmThread.start();

    while(iteration.get() < maxIterations) {
      TimeUnit.SECONDS.sleep(5);
      System.out.println("Iteration:  " + iteration.get()) ;
    }

    algorithmThread.join();
}
```

In this piece of code, the measure manager is obtained from the algorithm, so the measures are now accessible. In the case of the pull measures, the algorithm is executed in a thread and the client code prints every five seconds the current iteration while the algorithm is concurrently running.

We can observe that a listener is registered to use the push measure. An example listener class is shown next:

```
public class Listener
                   implements MeasureListener<List<Solution>> {
  private int counter = 0 ;

  @Override synchronized public void
          measureGenerated(List<Solution> solutions) {
   if ((counter % 10 == 0)) {
     // Do whatever action with the solution list (population).
   }
   counter ++ ;
  }
}
```

In this code, the `measureGenerated()` method is called whenever a push operation is invoked on the push measure, and an action is performed every 10 invocations. A typical action could be to write the current Pareto front approximation in a file for experimental evaluation.

The benefit of using a push measure in this example is that there is no need of modifying the algorithm code; all the logic is in the client side. The approach has also a minimum impact on performance: if no listeners are registered, the push operations only will make a check in an empty list.

In general, pull measures are useful when the client program needs to get information in a particular moment about the current state of the algorithm execution. On the contrary, pull measures are more useful when they are intended to inform the client code about relevant events that are produced in an asynchronous way. In the former example, that event is evaluating the population after an algorithm iteration, but more complex scenarios could be considered, e.g., the population has not changed in the last N iterations, or the population does not contain infeasible solutions in case of solving a constrained problem.

3.7 Improving Code Quality

A reason to start the jMetal project was that we found out that existing software for multi-objective optimization with metaheuristics was difficult to understand and to reuse, so a design goal we imposed ourselves was to pay attention to code quality. This has been appreciated by many users, but when we analyzed the current implementation we were aware that there is room for improvement, as pointed out in Section 2.6.

In the new implementation, we follow the guidelines of Robert C. Martin's *Clean Code book* [13], with the goal of making the code more readable, extensible and maintainable. This is a work in progress, which implies a deep refactoring of most of the code.

Directly related to code quality, an important issue is to apply unit and integration testing. We use jUnit[7] complemented with Mockito[8] when mocks are needed.

Because metaheuristics are stochastic methods relying on randomized components, writing unit tests for these components is not trivial. Focusing on variation operators such as crossover and mutation, our approach is to test them in two steps. Step 1 consists in checking that the parameters they receive are correct (e.g., the probabilities of applying them is between 0.0 and 1.0) and that the produced values are valid. Because these tests cannot ensure that the operator works properly, step 2 is to write a program that generates several solutions, applies the operator to them, and writes the results in a file; then, by plotting the obtained values we can visually check whether or not the result is the expected one.

Our approximation for integration testing is to run the algorithms to solve known benchmark problems. The expected values of the tests can be as simple as checking the number of returned solutions; for example, solving the Kursawe problem with NSGA-II configured with standard settings (a population size of 100, a stopping condition of 25'000 evaluations, a probability of crossover (SBX crossover) of 0.9, and a probability of mutation (Polynomial mutation) of 1/`NumberOfVariables`) should return 100 solutions, so the test checks that at least 99 solutions have been returned. The tests can be more accurate by knowing in advance the average values of a quality indicator (e.g, the Hypervolume) when solving a given problem and checking for this value with an error margin; this information can be obtained from pilot experiments and from studies published in the literature.

3.8 Parallelism Support

The inclusion of algorithm templates in jMetal 5.0 (Section 3.5) allows to provide parallelism support in an easy way when a list of solutions has to be evaluated. The approach taken is to define an interface called `SolutionListEvaluator`:

[7]jUnit: http://junit.org/
[8]Mockito: http://mockito.org

```
public interface SolutionListEvaluator
        <S extends Solution> extends Serializable {
  public List<S> evaluate(List<S> solutionSet, Problem problem);
}
```

This way the techniques based on the abstract evolutionary algorithm template can implement the `evaluate()` method in this way:

```
@Override
protected List<S> evaluatePopulation(List<S> population) {
  population = evaluator.evaluate(population, problem);

  return population;
}
```

By adopting this scheme, the evaluation of a list of solutions is encapsulated. We currently provide two implementations of such evaluators: sequential and multi-threaded. This scheme does not require to modify the code of the algorithm to use the evaluators, and the following metaheuristics use it: NSGA-II, SMPSO [14], SPEA2 and GDE3 [11]. More implementations are in progress to provide distributed versions.

4. CONCLUSIONS AND FUTURE WORK

In this paper we have identified and analysed a number of issues that can be improved in the jMetal framework for multi-objective optimization with metaheuristics. These issues are related to the class architecture, the quality of the code, the organization of the project, the lack of templates for algorithms, the absence of features for getting information at runtime, and the lack of parallelism support.

To cope with the identified points, we have introduced the new architecture of the next release of jMetal and the adopted solutions to solve those issues. As a result, the redesigned jMetal should lead to a significantly improved tool aimed at being useful to researchers of the multi-objective optimization community.

The future work is directly focused in finishing the new jMetal version and releasing it to the community.

5. ACKNOWLEDGMENTS

This work is partially funded by Grants TIN2014-58304-R (Ministerio de Economía y Competitividad), TIN2011-25840 (Ministerio de Ciencia e Innovación), P11-TIC-7529 and P12-TIC-1519 (Plan Andaluz de Investigación, Desarrollo e Innovación).

6. REFERENCES

[1] N. Beume, B. Naujoks, and M. Emmerich. Sms-emoa: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653–1669, 2007.

[2] C. Coello, D. Van Veldhuizen, and G. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Genetic Algorithms and Evolutionary Computation. Kluwer Academic Publishers, 2002.

[3] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.

[4] J. Durillo, A. Nebro, and E. Alba. The jmetal framework for multi-objective optimization: Design and architecture. In *CEC 2010*, pages 4138–4325, Barcelona, Spain, July 2010.

[5] J. Durillo, A. Nebro, F. Luna, B. Dorronsoro, and E. Alba. jMetal: a Java framework for developing multi-objective optimization metaheuristics. Technical Report ITI-2006-10, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S.I. Informática, Campus de Teatinos, 2006.

[6] J. J. Durillo and A. J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011.

[7] J. J. Durillo, A. J. Nebro, C. A. Coello Coello, J. García-Nieto, F. Luna, and E. Alba. A study of multiobjective metaheuristics when solving parameter scalable problems. *IEEE Transctions on Evolutionary Computation*, 14(4):618 – 635, August 2010.

[8] I. Giagkiozis, R. Lygoe, and P. Fleming. Liger: an open source integrated optimization environment. In *Genetic and Evolutionary Computation Conference, GECCO '13, Amsterdam, The Netherlands, July 6-10, 2013, Companion Material Proceedings*, pages 1089–1096, 2013.

[9] D. K. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[10] J. D. Knowles and D. W. Corne. Approximating the nondominated front using the pareto archived evolution strategy. *Evolutionary Computation*, 8(2):149–172, 2000.

[11] S. Kukkonen and J. Lampinen. GDE3: The third evolution step of generalized differential evolution. In *IEEE Congress on Evolutionary Computation (CEC'2005)*, pages 443 – 450, 2005.

[12] H. Li and Q. Zhang. Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii. *IEEE Transactions on Evolutionary Computation*, 12(2):284–302, April 2009.

[13] R. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[14] A. Nebro, J. Durillo, J. García-Nieto, C. Coello Coello, F. Luna, and E. Alba. Smpso: A new pso-based metaheuristic for multi-objective optimization. In *2009 IEEE Symposium on Computational Intelligence in Multicriteria Decision-Making (MCDM 2009)*, pages 66–73. IEEE Press, 2009.

[15] E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195, Summer 2000.

[16] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. In K. Giannakoglou, D. Tsahalis, J. Periaux, P. Papailou, and T. Fogarty, editors, *EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, pages 95–100, Athens, Greece, 2002.