Natural Computing manuscript No.
(will be inserted by the editor)

A Pattern-Driven Solution for Designing Multi-Objective Evolutionary Algorithms

Giovani Guizzo · Silvia R. Vergilio

Received: date / Accepted: date

Abstract Multi-Objective Evolutionary Algorithms (MOEAs) have been widely studied in the literature, which led to the development of several frameworks and techniques to implement them. Consequently, the reusability, scalability and maintainability became fundamental concerns in the development of such algorithms. To this end, the use of Design Patterns (DPs) can benefit, ease and improve the design of MOEAs. DPs are reusable solutions for common design problems, which can be applied to almost any context. Despite their advantages to decreasing coupling, increasing flexibility, and allowing an easier design extension, DPs have been underexplored for MOEA design. In order to contribute to this research topic, we propose a pattern-driven solution for the design of MOEAs. The MOEA designed with our solution is compared to another MOEA designed without it. The comparison considered: the Integration and Test Order (ITO) problem and the Traveling Salesman problem (TSP). Obtained results show that the use of this DP-driven solution allows the reuse of MOEA components, without decreasing the quality, in terms of hypervolume. This means that the developer can extend the algorithms to include other components using only object-oriented mechanisms in an easier way, while maintaining the expected results.

Keywords Metaheuristic Ddesign Pattern · Multi-Objective Evolutionary Algorithm · Software Testing · Hyper-Heuristic

Mathematics Subject Classification (2000) 68N30 · 68T20

This work is supported by the Brazilian funding agencies CAPES and CNPq under the grants: 307762/2015-7 and 473899/2013-2.

G. Guizzo · S. R. Vergilio
DInf - Federal University of Paraná, CP: 19081, CEP 19031-970
Curitiba – PR, Brazil
E-mail: {gguizzo, silvia}@inf.ufpr.br

1 Introduction

Evolutionary Algorithms (EAs) (Eiben and Smith, 2003) and Multi-Objective EAs (MOEAs) (Coello et al, 2007) are based on the theory of evolution, where the fittest solutions reproduce more often and survive by means of natural selection. In MOEAs, an individual is a solution from a population of solutions being evolved. The evolution process employs different procedures, such as fitness assignment, selection of the fittest individuals, crossover, mutation, replacement and other components for improving the final outcome.

EAs have been widely used to solve hard computational problems of diverse domain fields (Eiben and Smith, 2003). This fact led to the development of several frameworks and techniques to ease the implementation of these algorithms (Nebro et al, 2015; Ochoa et al, 2012; Tsyganov and Bulychov, 2012; Ventura et al, 2007; Fortin et al, 2012; Cahon et al, 2004). However, since EAs have several components and a lot of parameters, the maintainability and extensibility of the algorithms design became factors of interest. The issue is how to develop frameworks that can accommodate new algorithms and their components, without decreasing the software reusability. For example, if a developer wants to change the selection procedure of the algorithm by adding a new selection mechanism to his/her implementation, how can he/she do it without recoding the whole algorithm? To answer this question, the area known as Meta-heuristic Design Patterns (MDP) proposes solutions based on Design Patterns (DPs) (Gamma et al, 1995) to design metaheuristics, including EAs.

DPs (Gamma et al, 1995) are reusable solutions for common design problems that can be used to decrease the coupling and increase the cohesion between elements. The benefits of DPs are directly related to the scalability, maintainability and reusability of the software. Another benefit is that these patterns are usually abstract solutions and can be adapted to almost any object-oriented software (Gamma et al, 1995), including frameworks for evolutionary optimization.

We found in the literature some works regarding the application of design patterns with EAs and more particularly MOEAs (Nebro et al, 2015; Patelli et al, 2015; Wick and Phillips, 2002; Woodward and Swan, 2014; Woodward et al, 2014). These works show how DPs are in fact not only reusable solutions used to increase the software flexibility, but also how they play an important role in the robust implementation of frameworks. However, this is still an underexplored field when compared to the design pattern literature. We acknowledge that this field can benefit from the usage of design patterns. In order to contribute to such a field, in a previous work (Guizzo and Vergilio, 2016) we proposed a DP based solution for designing genetic operators for EAs. We showed how Visitor (Gamma et al, 1995) is able to decouple the genetic operators from the problem representations. In this preliminary work, a genetic operator behaves like a visiting operation that operates over a visited element (representation). The idea was to decouple the operator from the representation, and let them interchange freely without concerning about constraints

specific to each representation. For allowing this, each operator implements a method for each representation, complying with the respective constraints. Instead of letting the operator decide which method must be executed in which representation, this Visitor structure demands that the representation object invokes the operator method which is specific for such representation. In this sense, the representation manages how the visiting object will act over itself.

Motivated by the successful results, now in this paper we extend our previous solution by introducing a broader one for MOEA design. While our previous work only acknowledged the design of a single component of MOEAs (operators and representation interaction), the extended solution is capable of encompassing all components of a MOEA in order to improve the MOEA design as a whole. It is important to note that it does not necessarily improve the overall results of such algorithms, but mainly their structural quality. The proposed solution is based on four DPs: Visitor (as done in Guizzo and Vergilio (2016)), Bridge, Factory Method, and Builder. This solution was created and tested in a real development scenario, when we were designing and implementing a hyper-heuristic that automatically generates MOEAs for solving the integration and test order problem (ITO) (Mariani et al, 2016).

Hyper-heuristics are defined as heuristics used to select or generate low-level heuristics (Burke et al, 2013). They operate over the heuristic space instead of operating on the solution space directly. Hence, the main goal of hyper-heuristics is to find the right heuristics to be used in a given situation rather than trying to directly solve the problem (Burke et al, 2013). This is specially useful for developers that are not always familiar with optimization algorithms, sometimes lack on expertise to properly configure them, or simply do not have time for a task that may be automated by hyper-heuristics. For a more detailed overview of the hyper-heuristic field, we recommend the paper of Burke et al (2013).

The ITO problem is a software testing problem that consists in finding the best unit (the smallest part of a software) order for testing and integrating them (Assunção et al, 2014). This order can impact on the stubbing cost, which is the cost related to the creation of stubs for emulating required units that are not yet tested or implemented. As a consequence, a good unit ordering can reduce the overall testing cost.

For assessing the correctness of the proposed solution, we conducted in this paper a comparison between the MOEA designed with our DP-based solution and another MOEA designed without it. The results of the correctness test on two problems, ITO and the Multi-Objective Traveling Salesman Problem (TSP) (Paquete et al, 2004), show no statistical difference between the algorithms. This means that the results are statistically identical, but the MOEA designed with the DP-based solution takes advantage of the positive consequences of the DP application: i) the proposed solution enables the automatic design of MOEAs without requiring source-code manipulation, but only using object-oriented mechanisms; ii) it allows reuse of components by different MOEAs; and iii) the developer can extend his/her framework in a more de-

coupled way, by only adding new concrete classes to the structure and not changing the template of the algorithms.

This paper is organized as follows. Section 2 gives a brief background on the used DPs. Section 3 reviews some works in the literature regarding the usage of design patterns with metaheuristics and hyper-heuristics. Section 4 describes the proposed solution, presenting the problem, how the solution is designed, and its main consequences. Section 5 presents the evaluation in two scenarios: i) the original context in which the solution was designed (hyper-heuristic to solve the ITO problem); and ii) for solving the Multi-Objective TSP problem, a common and well-known combinatorial optimization problem of the Artificial Intelligence (AI) literature (Paquete et al, 2004). Finally, Section 6 concludes the paper and discusses future work.

2 Design Patterns

In the object oriented design, the developer must address some issues, such as how to grant reuse of artifacts, easy maintainability, good organization, and decoupled addition and removal of software components. If not addressed earlier in the development process, these issues may increase the final cost of the product and affect its quality. Design Patterns (DPs) are elegant solutions for these and other problems in the software development (Gamma et al, 1995). DPs are defined as description of interacting objects and classes that need to be personalized to solve a general design problem in a given context (Gamma et al, 1995). In other words, DPs are common solutions for common design problems, but at the same time that they provide well-defined solutions for the problems, they also need to be adapted for the particular context in which they are applied.

DPs are usually extracted from existing software and described into catalogs, such as the Gang of Four (GoF) catalog (Gamma et al, 1995). A DP names, abstracts, and identifies the main characteristics of the problem in which it is applied, in order to make it useful for almost any design. A DP is composed by four main elements: i) Name – the name given to describe the DP; ii) Problem – describes what are the common problems in which the DP is applicable; iii) Solution – describes the solution for the problem, which contains the elements that compose the pattern, their responsibilities, interaction and relationships; and iv) Consequences – the advantages, disadvantages and results of the pattern application.

The main benefits of using DPs are the improvement of some important software attributes, such as reusability, maintainability, understandability, extensibility, scalability, and others. This improvement can be achieved by carefully selecting the most appropriate DP for solving a design problem. In other words, a misapplied DP can cause the deterioration of the software design, hence the evaluation of which DP is the most appropriate requires some effort, but not as much as it would require the maintenance of a software with a bad solution.

In this paper we use four GoF DPs, which are described next according to Gamma et al (1995).

Visitor – Represents an operation to be executed over the elements of an object structure. The operations are decoupled from the visited objects, in a way that the visited objects decide how to receive the operation instead of letting the visitor operation decide how to visit an object. In this sense, the visited object invokes the most appropriate method of the operation object. When adding new operations, the existing visited objects will not have their classes changed, since the only structure that has to be adapted is of the new operation. Even though the visitor operation has methods that are somewhat coupled to the visited objects, this coupling is weaker than having a visitor operation checking each visited object to decide what is the best way to use it.

Bridge – Detaches the object abstraction from its implementation in two (or more) separated hierarchies. While the abstraction defines how something must be executed, the implementation defines what must be executed. Each concrete implementor class must implement a single variation of a functionality, while each refined abstraction must implement a variation of how to use each implementor. When a client object invokes an operation using the abstraction class, this operation is delegated to the aggregated implementor. By decoupling both, they can be separately and transparently exchanged for the client object.

Factory Method – Defines an interface for the creation of new objects and let the concrete classes to define how to create such objects. This pattern is specially useful when several objects of a given "family" can be instantiated in a given point, but the decision of which object to create is made at runtime. Hence, a different creation result can be obtained by changing an instantiation parameter or the whole concrete factory object.

Builder – Detaches the representation of a complex object from its creation process, such that the same construction process can be used to generate different objects. In other words, the Builder pattern defines a class to gradually build a complex object, part by part, but abstracting the whole construction process. In this sense, by changing one part during the building process, a different object is obtained and the differences are not explicitly visible to the objects that use it.

3 Related Work

Some authors (Lones, 2014; Mannava and Ramesh, 2012) point out that DPs are a useful tool for designing and implementing metaheuristics and evolutionary algorithms. One of the main motivations is that DPs standardize how

algorithms are defined and expressed, while also provide ways of code reuse and wisdom sharing between the evolutionary computation community. Some frameworks have already used DPs in their implementation (Alba et al, 2007; Nebro et al, 2015; Tsyganov and Bulychov, 2012; Ventura et al, 2007) and the consequences are beneficial. Therefore, the documentation of DPs can be very advantageous for metaheuristic and framework developers seeking for reusable solutions. Next, we present works that document DPs and are mainly used to improve the design of metaheuristics and hyper-heuristics.

Raidl (2014) reviews eight kinds of DPs for developing hybrid metaheuristics. Some patterns are used to help the developer to decide which component to use. Other ones are directly used to design better metaheuristics. Fernandez-Marquez et al (2013) present a catalog of bio-inspired mechanisms for self-organizing systems. Wick and Phillips (2002) do a comparison between Strategy and Bridge (Gamma et al, 1995) for developing a Genetic Algorithm (GA). The paper aims at giving experience with DPs to students by means of implementing GAs, since, as the authors stated, implementing GAs can be very interesting and enjoyable.

Besides DPs for conventional metaheuristics, some works, most related to ours, use DPs for improving the design of hyper-heuristics. Patelli et al (2015) extracted two anti-patterns (i.e. common bad solutions for a problem) used in hybrid metaheuristics and propose two new DPs called *Simple Black Box (Two-B)* and *Utility-based Black Box (Three-B)*. Woodward et al (2014) treat hyper-heuristics as a metaheuristic based on the *Composite* pattern (Gamma et al, 1995). By using a composite structure, a hyper-heuristic can take metaheuristics, search operators or even other hyper-heuristics as low-level heuristics and apply the search on any of them. Woodward and Swan (2014) use the *Template Method* pattern (Gamma et al, 1995) to represent metaheuristics and hyper-heuristics, similarly to the template of EAs as described in Krasnogor (2012). Therefore, hyper-heuristics can easily generate and configure new implementations for these abstract methods. This is useful when the user does not want to change the structure of the algorithm, but rather its behavior at runtime.

Most of the above-mentioned works are published in the "Metaheuristic Design Patterns" workshop (MetaDeeP) held at the Genetic and Evolutionary Computation Conference (GECCO) in 2014 and 2015. The goal of this workshop was to provide an open forum for researchers to contribute to this field by demonstrating the usefulness and advantages of such patterns in designing metaheuristics. Some other works on metaheuristics but non-related to EAs can be found in the proceedings of such workshop[1].

The solution proposed in this paper differs from other works in the literature (Nebro et al, 2015; Tsyganov and Bulychov, 2012; Woodward and Swan, 2014; Cahon et al, 2004), including our previous work (Guizzo and Vergilio, 2016), mainly on the algorithm structure. Usually a MOEA structure is defined

[1] More info at `http://www.sigevo.org/gecco-2014/workshops.html#mdp` and `http://www.sigevo.org/gecco-2015/workshops.html#wmdp`.

as a Template Method, where an abstract evolution procedure is hard-coded and any concrete class must implement all abstract methods. Thus, an algorithmic skeleton is defined which calls each abstract method in a predefined sequence. On the other hand, in this paper we propose the detachment of abstractions and implementations using the Bridge pattern, where each one can be independently exchanged. Therefore, instead of hard-coding an algorithmic structure and forcing the developer to recode common concrete components, our design technique lets each of these be freely interchanged and reused.

4 Proposed Solution

This section presents a solution for designing MOEAs, based on DPs (Gamma et al, 1995). The idea is to use this solution to decrease the coupling between the MOEA components. The next subsections present the problem, the solution and its main consequences.

4.1 The Problem

EAs are optimization algorithms based on the theory of evolution (Eiben and Smith, 2003). An individual (solution) is represented by a chromosome, which in turn is recombined with other chromosomes for the generation of children. The offspring undergo mutations for diversity, but only the fittest ones survive for the next generations and, consequently, reproduce to spread their genes. Therefore, several procedures such as replacement, crossover, mutation, population initialization and other ones must be chosen by the software developer and configured during the algorithm design phase. This design task is even more tiresome when MOEAs are needed.

Besides all these components and their parameters, MOEAs usually take into account the Pareto dominance concept for fitness evaluation (Coello et al, 2007). A solution dominates another when it is better or equal in each one of the objectives, and better in at least one. Otherwise, these solutions are called non-dominated and cannot be directly compared. Even so, a MOEA must compare them to decide about, for example, which solution should survive to the next generation. For this end, MOEAs use complex fitness assignment procedures that take into account the convergence and the diversity of solutions in the objective space. Usually, a MOEA or an EA has an abstract structure similar to the one presented in Algorithm 1.

What differs from an algorithm to another is the implementation of each procedure, such as the evaluation, selection, crossover, replacement, and others. Actually, there are several components and parameters for each step as described by Eiben and Smith (2003); Coello et al (2007). The choice of these details is a hard task and can be considered an optimization problem itself (Eiben and Smit, 2011), however it is still something that directly impacts the performance of the algorithms. When designing such algorithms or designing a

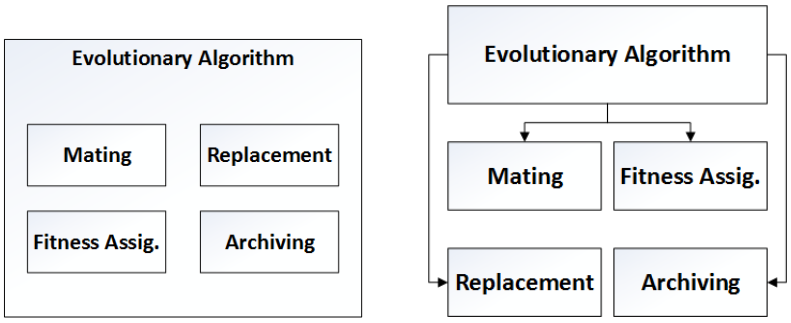
Algorithm 1: Common EA/MOEA template (Mariani et al, 2016)

```
1  begin
2      population ← Initialize population;
3      Evaluate(population);
4      Archive(population);
5      while stop criteria is not achieved do
6          mating ← Selection(population);
7          offspring ← Crossover(mating);
8          offspring ← Mutation(offspring);
9          Evaluate(offspring);
10         population ← Replacement(offspring, population);
11         Archive(population)
12     return population;
```

framework for working with them, the software developer must take that into consideration, or else the faced problem might not be solved in the most efficient way. We can observe a great interest in reusable and robust design methods for such algorithms. For instance, among the most used optimization algorithms to solve software engineering problems, EAs and MOEAs stand out as some of the most preferred (Harman et al, 2012).

An example of a framework that aids the development of MOEAs is the jMetal framework (Nebro et al, 2015). This framework has several algorithms available for execution, such as Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al, 2002) and Strength Pareto Evolutionary Algorithm 2 (SPEA2) (Zitzler et al, 2001). In its latest version, the jMetal framework employs the Template Method Design Pattern (Gamma et al, 1995) to create a method template for evolutionary algorithms in an abstract class, such as the one presented in Algorithm 1. The Template Method is already cataloged as a Metaheuristic Design Pattern by Woodward and Swan (2014), similarly as it is implemented in the jMetal framework.

However, even though Template Method presents a way to define how algorithms should behave, it is not entirely dynamic and reusable. This pattern defines a structure for the execution of the algorithms, which means that only the abstract execution steps are reused and each component of an algorithm is probably defined and implemented in the body of the concrete classes. Therefore, if the idea is to change the behavior of one method during execution, the whole algorithm object must be exchanged. Another problem is the duplication of common code for similar algorithms. One example is the random population initialization for the NSGA-II and SPEA2 algorithms. Following the Template Method pattern, the developer would be tempted to copy and paste the same code in the population initialization methods of both algorithms. This can lead to fault replication and a greater maintenance effort. This problem becomes even more impacting when there are no DPs involved. In the worst case scenario, each algorithm is hard-coded as a single function in a separated class and without a common interface. In such a scenario, not



(a) EA has components (b) EA uses components

Fig. 1 EA views

even the object exchange is possible, and tasks like refactoring may become error-prone and exhaustive for the software developer.

4.2 The Solution

As mentioned by Gamma et al (1995), it is preferable to use object delegation instead of class extension. This is due to the greater flexibility of being able to change each aggregated object during execution. In the context of meta-heuristics, it would be more useful to have an algorithm aggregating its functionalities/components instead of having them fixed at code time. The MOEA Framework[2] does something similar to this, but each algorithm has its own iteration/generation method, even though most MOEAs perform essentially the same steps at each generation.

We advocate that the problems presented in the previous section occur mostly due to our (not wrong) view of an EA structure. For instance, Fig. 1(a) shows a common representation of an EA structure. It is reasonable to think that an EA "has" procedures/components such as replacement, archiving, mating and fitness assignment, but arguably a more reusable interpretation of an EA structure would be the one depicted in Fig. 1(b), where the components are independent, and the EA merely uses their functionality in a certain order (sometimes the same as other EAs).

In our view, the ideal scenario would be the one of the latter figure, where an algorithm can change its components in a decoupled way like the MOEA Framework enables, while also reusing a skeleton method for its execution like the jMetal framework enables. This can bring more flexibility to the MOEA design and can increase the reuse of components.

The solution described in this section was motivated by the good outcomes of our previous work (Guizzo and Vergilio, 2016) and by a new hyper-heuristic

[2] `http://moeaframework.org/`

proposed by Mariani et al (2016). In Guizzo and Vergilio (2016), we only addressed the design of genetic operators, but now the problem faced demanded a different design encompassing several components of an EA. The proposed and extended solution is based on four DPs: i) Bridge – for defining a decoupled structure of components and MOEA execution; ii) Factory Method – for instantiating components; iii) Builder – for building MOEAs; and iv) Visitor – for decoupling the solution representation from the genetic operators, as proposed in Guizzo and Vergilio (2016). Following the EA view presented in Fig. 1(b), Fig. 2 presents the main structure (conceptual view) of our solution and how these patterns interact.

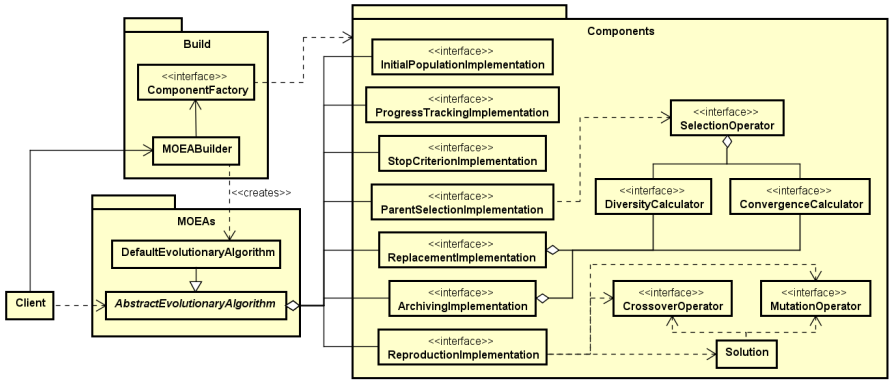


Fig. 2 Proposed solution (conceptual view)

The "Client" element is any object/entity that executes a MOEA using its abstract class "AbstractEvolutionaryAlgorithm". It can be another class, a module, a whole system, or any other element that can use a MOEA. "AbstractEvolutionaryAlgorithm" has several methods for invoking each of its components. The components on the other hand are abstracted by interfaces. While the "AbstractEvolutionaryAlgorithm" only uses the operations of the components, the "DefaultEvolutionaryAlgorithm" implements a method that assembles the operations together to execute a complete evolutionary procedure. For building such an algorithm, the "Client" class uses a builder object ("MOEABuilder"), which in turn uses some factory objects to instantiate the components. In this sense, the client does not instantiate directly each component in order to avoid coupling. The next subsections present in more detail this structure and what patterns are involved in each core substructure.

4.2.1 Bridge – Main Structure

The main structure of an EA/MOEA can be designed using the Bridge pattern (Gamma et al, 1995). Fig. 3 presents an excerpt of the proposed solution with the Bridge Pattern. The figure only shows two examples of implementa-

tion hierarchies (initialization and replacement), but, as seen in Fig. 2, there can be as many as needed.

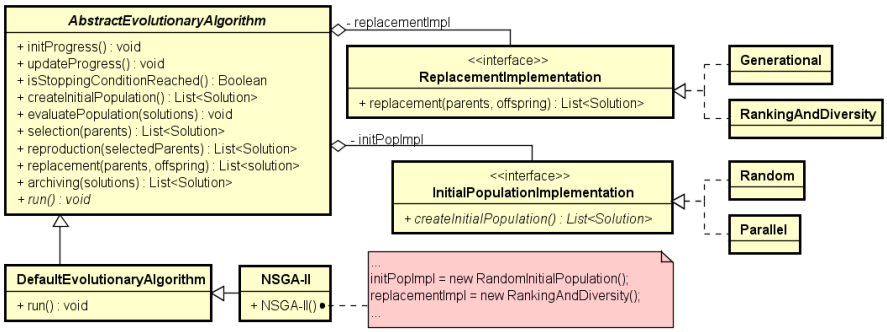


Fig. 3 Evolutionary Algorithm Bridge Structure

The abstract class "AbstractEvolutionaryAlgorithm" contains a set of steps performed by almost any EA, such as: i) "createInitialPopulation()" that creates the first population; ii) "evaluatePopulation()" that performs the fitness evaluation of a given set of solutions; and iii) "replacement()" that decides from the parents and children sets which individuals must survive for the next generation. Instead of coding each method directly in the concrete classes, "AbstractEvolutionaryAlgorithm" delegates the responsibility for the proper implementor. When a request for the population initialization comes, the aggregated object "initPopImpl" is called and this responsibility is given to it. Because the abstraction uses an interface, both initialization strategies "Random" and "Parallel" can be exchanged in runtime by changing the aggregated object. The same occurs to the Replacement object.

As depicted in Fig. 3, the concrete class "DefaultEvolutionaryAlgorithm" implements the abstract method "run()". This method calls the methods of the superclass to execute each step in the sequence defined in Algorithm 1. If a different variation is desired, the developer may only need to create a new class implementing a different "run()" method. This is not the case for conventional MOEAs, such as NSGA-II and SPEA2. Both algorithms have a default MOEA behavior and they only differ in the implementor objects. Therefore, NSGA-II and SPEA2 should both extend "DefaultEvolutionaryAlgorithm" and only instantiate different implementors. For instance, NSGA-II uses a Ranking (convergence attribute) and Diversity replacement, while the SPEA2 uses a Generational one due to its archiving strategy. Actually, in our implementation of this solution, the only methods coded in the NSGA-II and SPEA2 classes are their constructors. For implementing a new MOEA, the developer can extend the "DefaultEvolutionaryAlgorithm" class and change which implementors are instantiated in the constructor of the new algorithm class. Everything else is reused.

This solution has seven different implementation hierarchies, one for each kind of MOEA component: i) initialization – the population initialization procedure; ii) progress tracking – how the progress is computed, e.g., fitness evaluations, generations, time, etc.; iii) stop criterion – the type of stop criterion used to stop the evolutionary process, e.g., maximum time, maximum generations, maximum generations without fitness improvement, etc.; iv) parent selection – how the parents are selected, e.g., selecting only from the population, selecting only from the archive, or selecting from both archive and population; v) reproduction type – if the reproduction generates two children each mating, one child each mating, or if it is a steady state reproduction (one solution per generation); vi) replacement – how the replacement is done; and vii) archiving – how the archiving is done, if any. So, with this Bridge structure, the developer can instantiate an algorithm with any combination of components and without recoding a single component. The greatest effort is the first and only implementation of a new component.

Besides this first level Bridge structure for the MOEA components, there are some components that use other variable components as part of their execution. In this sense, each component can have a Bridge structure itself. For instance, Fig. 4 depicts the structure of the selection procedure as designed in this work.

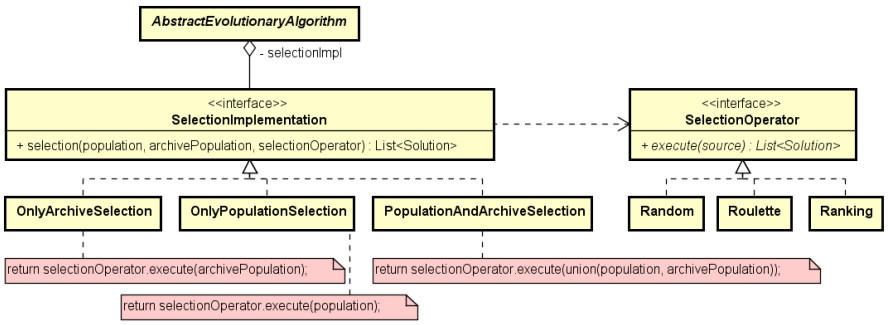


Fig. 4 Selection procedure using Bridge

As done for the initialization and replacement procedures, the selection procedure is also abstracted by an implementation interface: "SelectionImplementation". This interface has only one method with the intent of selecting parents from either the main population or the archive, or even both. The implementor classes of "SelectionImplementation" define from where this is done. But the actual selection is performed by a selection operator that implements the "SelectionOperator" interface, which receives only one list of solutions and returns the selected parents from this list according to its own selection procedure. The main idea is to decouple the "SelectionImplementation" object from the "SelectionOperator" object, letting both interchange freely. In the example of Fig. 4, we can compose nine different selection procedures without creat-

ing more classes or adapting code. Even though it is not a traditional Bridge structure (with abstract classes), it can still be considered a Bridge application since it defines abstraction hierarchies for the "how" ("SelectionImplementation") and "what" ("SelectionOperator") elements.

The same structure is used to decouple the diversity and convergence assessment operations from the selection operators, replacement procedure, and archiving procedure. Each of these components may use one or more of such assessment operations to evaluate the quality of solutions in a multi-objective problem and make a decision based on this quality. For example, the replacement procedure usually favors the survival of the best solutions, whereas the selection operator will try to select the best parents for mating. By decoupling the assessment operations, we can easily reuse them in these three components.

4.2.2 Visitor – Genetic Operators

We first introduced the usage of the Visitor pattern (Gamma et al, 1995) for designing genetic operators in Guizzo and Vergilio (2016), as shown in Fig. 5.

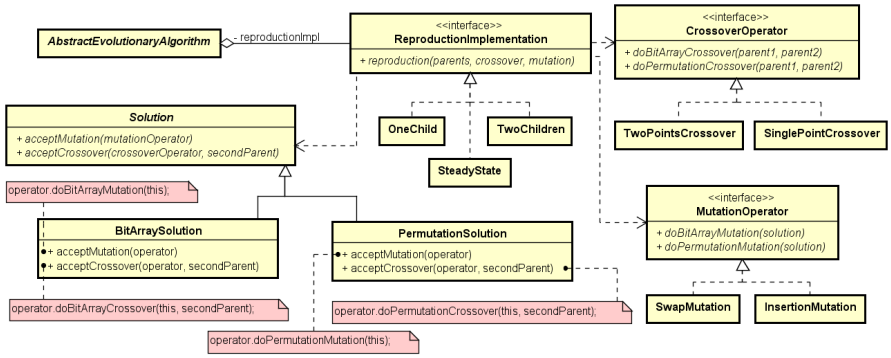


Fig. 5 Visitor for genetic operators. Adapted from Guizzo and Vergilio (2016)

Similarly to the other components, the "ReproductionImplementation" interface abstracts the reproduction procedure of the EAs. In the example there are three main reproduction procedures: i) "OneChild" – generates one child per mating; ii) "TwoChildren" – generates two children per mating; and iii) "SteadyState" – generates only one child per generation. Each of these reproduction strategies can use different crossover and mutation operators, which in turn operate over a solution object. The problem is that usually the operators are coupled to the solution representation but they still can be applied to different representations, i.e., the operators must comply with the restrictions and internal characteristics of the compatible representations to ensure a reliable solution generation. For example, the "Single Point Crossover" cuts the parent chromosomes in a single point and generates children interchanging the chromosome segments. When used in a bit array type of problem this is

straightforward, but when applied to a permutation problem some intrinsic characteristics must be addressed: i) genes cannot repeat in the child; and ii) the original gene ordering of the parents should be maintained as much as possible to avoid losing genetic load. Even though these details are different from one representation to another, the main procedure of the operator is the same: cut and interchange. Hence, this interaction must be carefully designed to prevent code replication. One may be tempted to create one class for each operator/representation combination, and thus to copy common code excerpts into each class. This is not a good practice, since it can cause the project to lose extensibility, reusability, and replicate bugs when the copied code is bugged in the first place.

By using the Visitor DP, we force the solution to decide how it must be manipulated instead of letting the operator decide it. In order to do this, first the evolutionary algorithm object calls the aggregated "reproductionImpl" object by its "accept" method. If the desired operation is the crossover, then the "acceptCrossover" method of the solution is called, or "acceptMutation" if the algorithm needs to apply a mutation operation. These methods receive as input the operator of the algorithm and any other solution involved in the mating. The solution object then invokes the crossover or mutation method based on the representation. For instance, if the representation is permutation, then the solution object uses the operator given as parameter to invoke the "doPermutationCrossover" or "doPermutationMutation" method, while giving itself as input parameter to the operator. If the representation used is a bit array, then the methods to be invoked by the "acceptCrossover" and "acceptMutation" methods are "doBitArrayCrossover" and "doBitArrayMutation". By changing the solution object, we transparently change the methods of the operators which are called to comply with the constraints of the representations.

The operators implement one method for each compatible representation. If a new representation is added to this design, then a new method for each operator is created to address the new constraints and to be invoked by the representation object during the search process. The main advantage is that the core functionality of the operators can be reused by these methods. Furthermore, the operators become less coupled to the representations, since they do not need to identify which representation is used and how to comply with the constraints.

4.2.3 Builder and Factory Method – MOEA Instantiation

In order to decrease the coupling between client objects and the MOEA objects during instantiation, we used the Builder and Factory Method DPs to abstract this process. Fig. 6 presents a communication diagram depicting how this instantiation is done.

For the instantiation of the implementor objects (components), we use the Factory Method DP. For each type of component, a factory class was created to instantiate the concrete objects. "InitializationFactory" and "ReplacementFactory" are both examples of these classes. In this structure, instead of directly

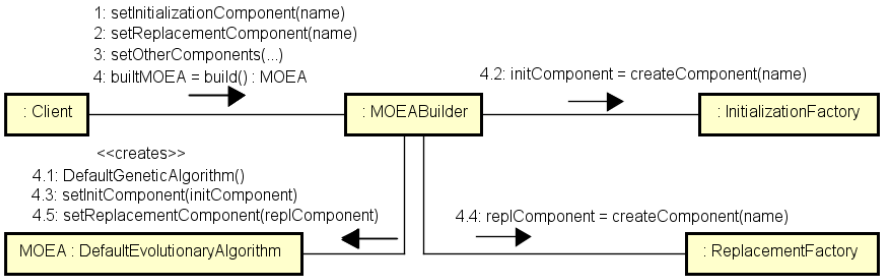


Fig. 6 Communication diagram for the interaction between the client and the pattern-driven solution

calling the constructor methods of the components, the objects that need this instantiation ("MOEABuilder") call the factory methods giving as parameter the name of the components or any other information about which concrete component must be instantiated.

However, because MOEAs are complex objects that aggregate several components, they can be built in a more robust way. For this, we applied the Builder DP. The builder class "MOEABuilder" constructs a "DefaultEvolutionaryAlgorithm" object by parts. If the developer wants to instantiate a custom MOEA, then he/she must call each method of the builder object giving as parameter the name of the components and then call its "build" method, just like "Client" does. The builder uses the factory classes to instantiate each part (component) of the MOEA, and at the end, after every component was instantiated, the build method assembles everything together and returns a fully-functional MOEA.

4.3 The Consequences

We designed this pattern-driven solution when we faced the necessity to reuse components and abstractions, while also enabling them to independently exchange. According to our experience, the main consequences of using this DP solution are described next.

- Code Reuse – Some algorithms may have common components, e.g., the random initialization of NSGA-II and SPEA2. In this case, both algorithms can instantiate the "Random" class and use this instantiation. When a software bug must be fixed in a common functionality such as this one, only the concrete class of the component must be changed;
- Easy Introduction of New Functionalities – Suppose that the developer needs to implement a new replacement component or a new operator. He/she only needs to create a class for this component and instantiate it for the algorithms that use it;
- Easy Change of Algorithm Steps – One common functionality of EAs is the fixed sequence of steps: i) selection; ii) reproduction; iii) evaluation;

iv) replacement; and v) archiving. If the developer wants, for example, to perform two replacements, then this can be done by creating an alternate concrete class for the abstraction class. The only required change is the addition of another replacement call in a new class, and by changing a different concrete class, existing classes (such as NSGA-II) will not have their functionality affected, as it happens when a template class is shared by all algorithms. Because the abstraction is detached from the implementation hierarchy, the implementation classes can be used in this new kind of EA. The difficulty in designing and maintaining such algorithms is specially prevailing when the developer is not familiar with software engineering design techniques, e.g., a beginner Artificial Intelligence researcher with little experience on software engineering;
- Runtime Exchange of Algorithm Behavior – If the developer must change the functionality of an algorithm at runtime, then the desired functionality must be instantiated and replaced in the abstraction class by simply exchanging objects;
- Decreased Coupling – It is common for EAs to have a certain budget for their execution. This budget can be a number of maximum fitness evaluations, a number of generations or even a certain amount of execution time. Instead of creating similar algorithms by changing only the stop condition method, the developer can vary the strategy by changing the object for this component. Thus, the algorithm is not coupled to its components;
- Increased Cohesion – Because the algorithm abstraction classes are only responsible for calling the functionalities and do not implement any component functionality, each implementor does a single operation and the algorithm only needs to handle how it is executed.

We also observed some negative consequences of the proposed solution.

- Decrease Understandability – The proposed solution has a structure which is not clear at first sight due to the great number of classes and interfaces to compose an algorithm. This can hinder the implementation of the first components of the MOEA;
- Greater Amount of Code – The amount of code is greater when compared to a straightforward implementation of an EA. This is mainly due to the definition of several interfaces and abstract classes that delegate method calls or simply declare methods;
- Empty Methods – Some representation specific methods might be left empty in the operators body, because an operator class must implement all representation specific methods and not be compatible with them all. This can difficult the debug activity.

5 Case Studies

We created the solution during the implementation of a hyper-heuristic named GEMOITO (Grammatical Evolution for the Multi-Objective Integration and

Test Order problem) (Mariani et al, 2016). GEMOITO automatically generates MOEAs for solving the ITO problem. In this paper we conducted a brief test to evaluate the correctness of the proposed solution. For this end, we compared the quality of the results obtained by an algorithm generated by GEMOITO with a traditional implementation of the same algorithm with the same parameters. We executed the algorithms on two problems: i) ITO; and ii) Multi-Objective TSP. We added the experimentation for the latter one to show that the proposed solution is not exclusive for Software Engineering problems, and that it can be used by AI researchers for solving common AI problems as well. This section first introduces GEMOITO, how it was implemented with the proposed solution and then the obtained results.

5.1 GEMOITO

GEMOITO (Mariani et al, 2016) is an off-line hyper-heuristic that uses Grammatical Evolution (GE) (Ryan et al, 1998) to automatically generate MOEAs for solving the Integration and Test Order problem (ITO). The ITO problem (Assunção et al, 2014) consists in finding an order to integrate and test the units of a system, in a way that the stubbing cost is minimized. A stub is constructed to emulate the behavior of a unit A that is not available during the test of another unit B, such that B is dependent on A. The stub is discarded when A is integrated and tested. In this sense, the development of unnecessary stubs can increase the testing cost. This is a hard permutation problem and may have several objectives, such as the number of attributes, number of methods, number of parameters and number of return types to be emulated. More details about the problem can be found in Assunção et al (2014).

During the training phase, GEMOITO uses the GE algorithm to evolve a set of MOEAs. GE is a type of Genetic Programming (GP) algorithm (Koza, 1992) that uses a context-free grammar to build semantically correct programs and works like an EA by evolving such programs. Because a MOEA is also a program, GE can be used as hyper-heuristic for constructing them. GEMOITO uses a grammar containing several common components and uses these components to automatically build different MOEAs. In this build process, the components and their parameters are selected from the available grammar options using the genes of the GE solution (integer array). For instance, a gene is used to decide which initialization component will be used among several ones available in the grammar. Once built, the MOEAs are then executed and their result are evaluated using the hypervolume indicator (Zitzler et al, 2003). The hypervolume indicator measures the area/volume of the objective space that is dominated by a given Pareto front (Zitzler et al, 2003). At the end, the best MOEA is given as result and it can be used to solve other instances of the integration and test order problem. GEMOITO obtained the best results when compared to the conventional algorithms NSGA-II and SPEA2, and to another hyper-heuristic proposed for the same problem (Guizzo et al, 2015).

5.2 GEMOITO and the Proposed Pattern-Driven Solution

GEMOITO was implemented with our pattern-driven solution, and used as a case study for evaluation. The idea is that such a hyper-heuristic can generate MOEAs using solely object-oriented mechanisms and without code manipulation. Because the proposed solution is flexible enough, it can be used in this context without much adaptation. For this adaptation, we considered that GEMOITO plays the role of a "Client" element.

For instance, Fig. 6 shows the interaction between GEMOITO ("Client") and the proposed solution during the instantiation of a MOEA. GEMOITO selects options from its grammar that represent the names of the components and can be given as parameter to the factory classes. Following the example of Fig. 6, GEMOITO first chooses the initialization component and gives its name to the MOEA builder object "MOEABuilder" through the "setInitializationComponent" method. Then, it chooses a replacement component from the grammar and gives it to the MOEA builder using the "setReplacementComponent" method. This is repeated for every component and parameter being selected. At the end, GEMOITO invokes the "build" method and the builder object starts invoking every factory method to build the components. Then, the builder object assembles the built components into an instance of "DefaultEvolutionaryAlgorithm" and returns this MOEA to GEMOITO, which in turn executes and evaluates the quality of the built MOEA through the "AbstractEvolutionaryAlgorithm" class. These are the exact same steps that a developer would have taken to instantiate a MOEA with our solution if he/she was the client in this situation.

Before executing GEMOITO, we had to implement once each component used in the grammar. Then, GEMOITO was able to instantiate the MOEAs reusing the components according to the solutions and grammar manipulated by the GE. All of this without recoding a single component or manipulating code. If the developer wants to extend the grammar to allow the instantiation of other different components, then he/she needs to adapt the grammar, code only once each new component and define an instantiation statement in the respective Factory Method class of the component. Using a less extensible structure, such as the Template Method one, this functionality extension would rely on other mechanisms that would be harder to implement and maintain. For example, GEMOITO would have to assemble pieces of code together into a single class and compile the source-code, thus the developer would have to concern about the manipulation of source-code that can be troublesome for algorithms such as these. One kind of problem faced in that case are compilation errors due to duplicate variables.

A possible improvement for GEMOITO is to allow the generation of different "run" methods for the MOEAs. With the provided solution, this can be done independently of the components and parameters selection procedure. Because Bridge detaches the abstraction from the implementation, GEMOITO can generate a different order of component calls inside the "run" method of the abstraction class "DefaultEvolutionaryAlgorithm". For doing so, other DPs

or even architectural patterns (Garlan and Shaw, 1994) should be investigated. A possibility is the usage of the Pipes and Filters architectural pattern, which states that the output of an element is the input of another. In this case, a structure of component calls could be assembled where the output of each component is the input of another. Moreover, GEMOITO can be applied to other problems. To adapt this hyper-heuristic, the developer must change the grammar to only allow operators that comply with the representation of this new problem. Because GEMOITO and the proposed pattern-driven solution are decoupled from the problem domain, both can be reused without further effort.

5.3 Correctness Test

In order to evaluate if the generated MOEAs, designed with our pattern-driven solution, work as intended, we conducted a correctness test guided by the following research question: "Does the proposed pattern-driven solution impact on the overall quality of the results?". For answering this question, we executed GEMOITO using the proposed solution to generate a version of NSGA-II and compared this generated NSGA-II with the NSGA-II implemented in jMetal 5.0 (Nebro et al, 2015) using the exact same parameters. As mentioned before, we tested both algorithms on two different problems: ITO and Multi-Objective TSP. Despite having different structures, we expect to obtain the same results for both algorithms, since the proposed solution only presents a different and more flexible way of designing the same MOEAs, focusing solely on improving the structural quality of the algorithms and not their results.

GEMOITO evolved just some components and parameters of NSGA-II in order to maintain its key features, such as dominance depth and crowding distance replacement, binary tournament selection, and others. The evolved components are: population initialization, crossover operator and mutation operator. The resulting NSGA-II has the following configuration: i) random initialization; ii) PMX Crossover with 100% probability; iii) Swap Mutation with 1% probability; and iv) population size of 50. The same configuration was given to the NSGA-II implementation of the jMetal framework (Nebro et al, 2015), a well known framework for MOEAs. Both algorithms were executed for 60,000 fitness evaluations in 30 independent runs.

We tested the algorithms in seven instances of the ITO problem (Assunção et al, 2014) and in 12 instances of the Multi-Objective TSP problem (Paquete et al, 2004)[3]. The results were evaluated using the hypervolume indicator (Zitzler et al, 2003) to measure the quality of the generated fronts, and the Kruskal-Wallis (Corder and Foreman, 2009) statistical test with 95% of significance. Kruskal-Wallis is a non-parametric statistical test used to compare two or more groups of values (Corder and Foreman, 2009), in order to

[3] The benchmarks for the Multi-Objective TSP were obtained at `https://eden.dei.uc.pt/~paquete/tsp/`.

assess if there is any statistical difference between the groups. The hypervolume was computed using as reference set the Pareto set containing all the non-dominated solutions found by both algorithms. The objectives were normalized accordingly. These results are presented in Tables 1 and 2 for the ITO and Multi-Objective TSP problems respectively. The first columns show the name of the instances used in this case study. The second columns show the results for the NSGA-II implementation generated by GEMOITO; the third columns show the results for the NSGA-II implementation coded in the jMetal framework; and the fourth columns show if there is statistical difference ($\neq$) or not ($=$).

Table 1 Average hypervolume values by algorithm implementation for the ITO problem

Instance	GEMOITO NSGA-II	jMetal NSGA-II	Kruskal-Wallis
MyBatis	0.705	0.678	=
AJHsqldb	0.515	0.521	=
AJHotDraw	0.405	0.410	=
BCEL	0.680	0.683	=
JHotDraw	0.401	0.348	=
HealthWatcher	0.832	0.890	=
TollSystems	0.676	0.765	=
JBoss	0.817	0.799	=

Table 2 Average hypervolume values by algorithm implementation for the Multi-Objective TSP problem

Instance	GEMOITO NSGA-II	jMetal NSGA-II	Kruskal-Wallis
kroAB100	0.523	0.551	=
kroAB150	0.483	0.534	=
kroAB200	0.439	0.503	=
kroAC100	0.555	0.598	=
kroAD100	0.604	0.595	=
kroAE100	0.540	0.546	=
kroBC100	0.604	0.592	=
kroBD100	0.568	0.599	=
kroBE100	0.553	0.575	=
kroCD100	0.538	0.534	=
kroCE100	0.585	0.576	=
kroDE100	0.532	0.556	=

For all instances of both problems, the results were statistically equivalent, regarding hypervolume. In fact, although omitted in this paper, the obtained fronts overlap almost entirely. As seen with these results, the overall quality of the results obtained by the generated algorithm does not differ from the same algorithm designed using another implementation. Therefore, we can conclude that the proposed pattern-driven solution does not present any significant quality loss or gain (as expected) in terms of hypervolume for the

problems tested. In the end, we were able to improve the structural quality of our MOEA design using this DP-based solution, without affecting the quality of the results. Moreover, with this solution, it was easier to extend GEMOITO to support new concrete components.

6 Concluding Remarks

In this paper, we present a pattern-driven solution for the design of MOEAs. In this solution we use the Bridge design pattern to define a decoupled structure between the algorithm abstraction and its execution components. The operators and representations classes are designed using the Visitor DP in order to allow a better code reuse and to decrease their coupling. We also implemented the Builder and Factory Method patterns to aid the instantiation of components and MOEAs in an easier way. As a consequence, we obtained a more extensible and flexible design.

The solution was created for the implementation of the hyper-heuristic GEMOITO, which automatically generates MOEAs to solve the integration and test order problem. The solution allows an easy extension of the hyper-heuristic to include new components and to generate new kinds of MOEAs. We performed a correctness test to asses if there is any significant difference between the results obtained by an algorithm designed with the proposed solution and the results obtained by the same algorithm implemented in the jMetal framework without the proposed solution. We tested in two different problems: the ITO and Multi-Objective TSP problems. As this correctness test shows, there is no statistical difference between the results of both algorithms in terms of hypervolume. In this sense, the pattern-driven version of the algorithm performed similarly to the unchanged version of the algorithm in terms of hypervolume, but the former carries all the advantages of a pattern-driven structure.

As future work we intend to implement other DPs to improve other parts of the MOEA design, such as to decouple the problem and operators from the chromosome representation, and to evaluate the solution for other kinds of problems.

References

Alba E, Luque G, Nieto JG, Ordonez G, Leguizamon G (2007) MALLBA: a software library to design efficient optimisation algorithms. Int J Innovative Comput Appl 1(1):74–85

Assunção WKG, Colanzi TE, Vergilio SR, Pozo A (2014) A multi-objective optimization approach for the integration and test order problem. Information Sciences 267(0):119–139

Burke EK, Gendreau M, Hyde M, Kendall G, Ochoa G, Özcan E, Qu R (2013) Hyper-heuristics: A survey of the state of the art. J Oper Res Soc 64(12):1695–1724

Cahon S, Melab N, Talbi EG (2004) ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. Journal of Heuristics 10(3):357–380

Coello CAC, Lamont GB, Veldhuizen DAV (2007) Evolutionary Algorithms for Solving Multi-Objective Problems Second Edition, 2nd edn. Springer Science

Corder GW, Foreman DI (2009) Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach. New Jersey: Wiley

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evol Comput 6(2):182–197

Eiben AE, Smit SK (2011) Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm Evol Comput 1(1):19–31

Eiben AE, Smith JE (2003) Introduction to evolutionary computing. Springer Science & Business Media

Fernandez-Marquez JL, Di Marzo Serugendo G, Montagna S, Viroli M, Arcos JL (2013) Description and composition of bio-inspired design patterns: A complete overview. Natural Computing 12(1):43–67

Fortin FA, De Rainville FM, Gardner MAG, Parizeau M, Gagné C (2012) DEAP: Evolutionary Algorithms Made Easy. Journal of Machine Learning Research 13(1):2171–2175

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc.

Garlan D, Shaw M (1994) An introduction to software architecture. In: Advances in Software Engineering and Knowledge Engineering, Publishing Company, pp 1–39

Guizzo G, Vergilio SR (2016) Metaheuristic Design Pattern: Visitor for Genetic Operators. In: Brazilian Conference on Intelligent Systems

Guizzo G, Fritsche GM, Vergilio SR, Pozo ATR (2015) A Hyper-Heuristic for the Multi-Objective Integration and Test Order Problem. In: Genetic and Evolutionary Computation Conference, ACM, pp 1343–1350

Harman M, Mansouri SA, Zhang Y (2012) Search-based Software Engineering: Trends, Techniques and Applications. ACM Comput Surv 45(1):11–61

Koza JR (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press

Krasnogor N (2012) Memetic Algorithms, Springer Berlin Heidelberg, pp 905–935

Lones MA (2014) Metaheuristics in nature-inspired algorithms. In: Genetic and Evolutionary Computation Conference, ACM Press, pp 1419–1422

Mannava V, Ramesh T (2012) Load Distribution Design Pattern for Genetic Algorithm Based Autonomic Systems. Procedia Eng 38:1905–1915

Mariani T, Guizzo G, Vergilio SR, Pozo ATR (2016) A Grammatical Evolution Hyper-Heuristic for the Integration and Test Order Problem. In: Genetic and Evolutionary Computation Conference, ACM, pp 1069–1076

Nebro AJ, Durillo JJ, Vergne M (2015) Redesigning the jMetal Multi-Objective Optimization Framework. In: Genetic and Evolutionary Compu-

tation Conference, pp 1093–1100
Ochoa G, Hyde M, Curtois T, Vazquez-Rodriguez JA, Walker J, Gendreau M, Kendall G, Parkes AJ, Petrovic S, Burke EK (2012) HyFlex: A Benchmark Framework for Cross-domain Heuristic Search. In: European Conference on Evolutionary Computation in Combinatorial Optimization, Springer-Verlag, pp 136–147
Paquete L, Chiarandini M, Stützle T (2004) Pareto Local Optimum Sets in the Biobjective Traveling Salesman Problem: An Experimental Study, Springer Berlin Heidelberg, pp 177–199
Patelli A, Bencomo N, Ekárt A, Goldingay H, Lewis P (2015) Two-B or not Two-B? Design Patterns for Hybrid Metaheuristics. In: Genetic and Evolutionary Computation Conference, ACM Press, pp 1269–1274
Raidl GR (2014) Decomposition based hybrid metaheuristics. Eur J Oper Res 244:66–76
Ryan C, Collins JJ, Neill M (1998) Grammatical evolution: Evolving programs for an arbitrary language. In: Genetic Programming, vol 1391, Springer, pp 83–96
Tsyganov AV, Bulychov OI (2012) Implementing Parallel Metaheuristic Optimization Framework Using Metaprogramming and Design Patterns. Applied Mechanics and Materials 263-266:1864–1873
Ventura S, Romero C, Zafra A, Delgado JA, Hervás C (2007) JCLEC: a Java framework for evolutionary computation. Soft Computing 12(4):381–392
Wick MR, Phillips AT (2002) Comparing the template method and strategy design patterns in a genetic algorithm application. ACM SIGCSE Bulletin 34(4):76–80
Woodward J, Swan J, Martin S (2014) The 'composite' design pattern in metaheuristics. In: Genetic and Evolutionary Computation Conference, ACM Press, pp 1439–1444
Woodward JR, Swan J (2014) Template Method Hyper-heuristics. In: Genetic and Evolutionary Computation Conference, ACM, pp 1437–1438
Zitzler E, Laumanns M, Thiele L (2001) SPEA2: improving the strength Pareto evolutionary algorithm. Technical report, Department of Electrical Engineering, Swiss Federal Institute of Technology
Zitzler E, Thiele L, Laumanns M, Fonseca CM, da Fonseca VG (2003) Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans Evol Comput 7(2):117–132