Metaheuristic Design Pattern: Visitor for Genetic Operators

Giovani Guizzo, Silvia R. Vergilio
Computer Science Department, DInf–UFPR, Curitiba–PR, Brazil
Email: {gguizzo, silvia}@inf.ufpr.br

***Abstract*—Metaheuristics, such as Genetic Algorithms (GAs), and hyper-heuristics have been widely studied and applied in the literature. This led to the development of several frameworks to aid the execution and development of such algorithms. Consequently, the reusability, scalability and maintainability became fundamental points to be attacked by developers. Such points can be improved using Design Patterns, but despite their advantages, few works have explored their usage with metaheuristics and hyper-heuristics. In order to contribute to this research topic, we present a solution based on the Visitor pattern used to design genetic operators. A case study is presented with the Hyper-heuristic for the Integration and Test Order problem (HITO). This case study shows that the proposed solution can increase the reusability of the implemented operators, and also enable easy addition of new genetic operators and representations.**

I. Introduction

Genetic Algorithms (GAs) [1], [2] have been widely used in the optimization literature. GAs are metaheuristics that employ the concept of genetic evolution and natural selection to find good solutions for hard problems. Therefore, the algorithms have some components such as fitness evaluation, crossover, gene mutation and population replacement. Each component may vary from one algorithm to another, but often some procedures are the same for all of them.

More recently, research on hyper-heuristics has raised interest of the optimization community [3]. Hyper-heuristics are heuristics used to select or generate low-level heuristics, such as genetic operators and metaheuristics. Hyper-heuristics act over the heuristic space instead of the search space directly, i.e., rather than searching for a solution, hyper-heuristics search for the best heuristic to find the solution.

This constant research in the evolutionary computation field led to the development of several frameworks used to aid the usage and design of new algorithms [4]–[7]. However, because GAs and hyper-heuristics may have complex structures, the code reuse, maintainability and extensibility became a concern for the developers. One issue is to design frameworks that can accommodate new algorithms, problems, chromosome representations, operators, and so on, without decreasing the software quality. Furthermore, hyper-heuristic adds another level of complexity to the framework, since it employs a higher level of optimization, e.g., automatic tuning of GAs and Adaptive Operator Selection (AOS) [3]. Hence, the framework code can become interlaced and hard to maintain. For example, considering a problem that has several representations and genetic operators for the optimization, a selection hyper-heuristic for AOS must select during its execution the specific operators for the representation chosen by the user. In a greatly coupled design, incompatible operators and representations may result in execution errors and compromise the optimization integrity.

Design Patterns (DPs) [8] are reusable solutions for common design problems that can be used for decreasing the coupling and increasing 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 [8], including frameworks for evolutionary optimization. Due to this we find some works [5], [9]–[12] devoted to the application of DPs with metaheuristics, GAs and hyper-heuristics in the area called Meta-heuristic Design Patterns (MDP). The researches show how DPs can help the developer to achieve more flexible algorithm designs. However, this is still an underexplored area when compared to the DP literature.

To contribute to the MDP area, this work presents a solution to improve the design of GAs and hyper-heuristics using the Visitor DP [8]. The idea is to freely interchange genetic operators and representations, and to provide a structure that is able to easily accommodate new components. In order to present how this solution is applied in a real world scenario, we describe a case study using the Hyper-heuristic for the Integration and Test Order Problem (HITO) [13]. This case study shows how this hyper-heuristic, designed for a specific problem, is easily extended to another context by using the proposed DP solution. HITO was chosen because it is an operator selection hyper-heuristic and uses GAs as main algorithms, which fits well in the context of this paper and can greatly benefit from the DP solution herein presented.

This paper is organized as follows. Section II describes DP concepts. Section III reviews related work from MDP. Section IV introduces the proposed solution. Section V presents the case study using HITO. Finally, Section VI concludes this paper and discusses future work.

II. 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 product final cost and affect its quality. Design Patterns (DPs) are elegant solutions for these and other problems in the software development [8]. 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 [8]. 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 [8]. 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 maintaining a software with a bad solution.

III. Metaheuristic Design Patterns

According to some authors [14], [15], DPs are useful for designing and implementing GAs. The main motivation is that DPs define a standard to design and develop algorithms, while also improve the code reusability and wisdom sharing between the evolutionary computation community. Frameworks in the GA literature already use DPs [4]–[7] and the observed consequences are mostly beneficial. Therefore, the documentation of DPs in this context can be advantageous for metaheuristics and frameworks developers seeking for reusable solutions. In this section we present some related work on this subject, focusing on DPs used in the design or development phases.

Raidl [16] reviews eight kinds of DPs for developing hybrid metaheuristics, which are used to help the engineer to decide the components used in the hybridization. Fernandez-Marquez et al. [17] present a catalog of bio-inspired mechanisms for self-organizing systems. Wick and Phillips [10] do a comparison between Strategy and Bridge [8] for developing a Genetic Algorithm (GA). The paper tries to give students some experience with DPs by means of implementing GAs, since, as the authors described, the GAs design and implementation can be very interesting and enjoyable.

Besides DPs for conventional metaheuristics, some works use DPs for improving the design of hyper-heuristics. Patelli et al. [9] extracted two anti-patterns (bad solution for a problem) commonly seen in hyper-heuristics. The authors then propose two new DPs called *Simple Black Box (Two-B)* and *Utility-based Black Box (Three-B)*. Woodward et al. [12] treat hyper-heuristics as a metaheuristic based on the *Composite* pattern [8]. Using this pattern, a hyper-heuristic can perform search on the metaheuristic, search operators or hyper-heuristic search space. Woodward and Swan [11] use the *Template Method* pattern [8] to represent metaheuristics, which allows hyper-heuristics to easily generate and configure new implementations for these metaheuristics. This is useful when the user does not want to change the structure of the algorithm, but rather its behavior at runtime.

We can see that MDP works have appeared recently. If we take the DP literature as a whole, only few works apply DPs to metaheuristics and (specially) hyper-heuristics. In short, we only found three works on DPs and hyper-heuristics, all of them were published in the Metaheuristic DPs Workshop (MetaDeeP), a workshop of the Genetic and Evolutionary Computation Conference (GECCO). We acknowledge that the field of hyper-heuristics and GAs can benefit a lot from the usage of DPs, and our work addresses this underexplored subject. The goal is to ease the design of more reusable, extensible and cohesive algorithms. For this end, in the next section, we present a solution based on Visitor [8], a pattern not yet explored with genetic algorithms.

IV. Visitor Design Pattern for Genetic Operators

This section presents a solution based on the Visitor DP [8] for designing crossover and mutation operators. The idea is to use this DP to decrease the coupling between genetic operators and the problem representation. The next subsections present the problem, the solution and its main consequences.

A. Problem

Genetic Algorithms (GAs), usually apply search operators such as crossover and mutation for generating new solutions [2]. These genetic operators use existing solutions (parents) to create similar new solutions (children). However, the solutions are modified in the genotype level instead of the phenotype level. That means that the chromosome of the solutions (decision variables) are changed, and these changes will reflect in different phenotypes (fitness values) for them.

A difficulty found manipulating chromosomes is the great gamma of possible representations for the chromosomes (solutions) of the problem [2]. For instance, a solution can be represented as an array of integers, an array of bits, an array of float numbers or many others. Each of these representations has some restrictions such as the maximum and minimum values that each gene can have, or even restrictions dictating that a value cannot appear more than once in the whole array (e.g. permutation representation). If these concerns are not taken into account when an operator is generating new solutions, then unfeasible solutions may be generated and the evolution may be negatively affected [1].

Due to these restrictions, when a genetic operator is implemented, usually it is applied to only one kind of representation. For example, the Two Points Crossover Operator [2] cuts two parents in two points, and all the genes contained between these points are copied from parent 1 to child 1 and from parent 2 to child 2. After this, the remaining genes are copied from parent 1 to child 2 and parent 2 to child 1. When this operator is applied to a bit array, the procedure goes as described. However, for a permutation array the genes cannot repeat. Even though the main steps are similar, they differ in the details.

When a developer is designing a genetic algorithm framework that supports several operators and problem representa-

tions, he/she may be tempted to create a class for each combination of operator-representation (e.g., "PermutationTwoPointsCrossover" and "BitArrayTwoPointsCrossover" classes). An improvement to this would be the addition of interfaces for each kind of operator and representation in order to enable the free exchange of operators. However, both situations can lead to code duplication, which hardens the maintenance of the algorithm and decreases reuse of source-code. Another problem is that runtime errors can happen when an operator is instantiated for an incompatible problem representation.

Furthermore, hyper-heuristics can manage several genetic operators in their procedure. This can be a complex algorithm to develop and, if the design flexibility is not treated earlier, the developer may deal with a low scalability later when trying to add new operators in the selection process.

B. Solution

The proposed solution is based on the Visitor DP [8]. Briefly, the Visitor pattern defines a set of visitor classes (the operators) that must behave in different ways depending on the concrete class of the visited elements (the problem representations). Both types are abstracted by interfaces or abstract classes, which eases the object interchange. Figure 1 illustrates the standard Visitor structure.

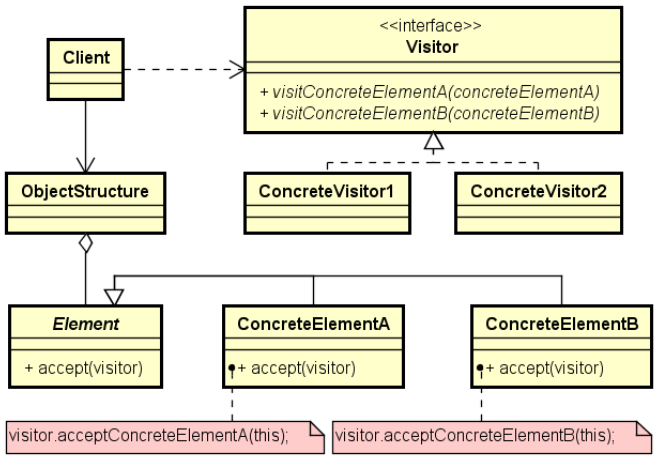

Fig. 1. Visitor Standard Structure

Instead of the visitor classes ("ConcreteVisitor1" and "ConcreteVisitor2") considering which elements ("ConcreteElementA" and "ConcreteElementB") they can operate upon, these classes will only hold the available operations ("visitConcreteElementA()" and "visitConcreteElementB()"). In turn, the concrete element classes have a common operation receiving a visitor, and with this operation the element will only call the operation that can be applied to itself. For instance, "ConcreteElementA" can only receive the application of the "visitConcreteElementA()" operation, thus it will only call this operation regardless of which visitor object is applied.

In terms of GAs, the Visitor classes (the ones holding the functional operations) are the Mutation and Crossover operators, whereas the visited elements are the solutions. Figure 2 depicts how we implemented this pattern with GAs.

In the figure, each crossover and mutation operator implements one operation per solution representation. For instance, the "TwoPointsCrossover" class implements both operations "doBitArrayCrossover()" and "doPermutationCrossover()" for "BitArraySolution" and "PermutationSolution" respectively. When an algorithm wants to apply this operator in two Permutation solutions, then this operator and a parent solution are given as parameters to the "acceptCrossover()" method of the other parent solution. The solution will then call the "doPermutationCrossover()" operation from the operator and pass itself and the other parent as parameter. At the end, the crossover operation is applied to the solutions, and the solution class is the only class that obtains any information about which operation must be executed for which kind of solution.

The same situation occurs when a mutation operator is applied to a solution. The only difference is that the solution will receive the visit of the operator through the "acceptMutation()" method. If a mutation operator cannot be applied to Bit Array solutions for example, then its "doBitArrayMutation()" method can be empty, throw an exception or be handled in other ways.

C. Consequences

The main consequences of using the proposed solution are:

• Decoupling of Operator and Solution – By using this pattern, the developer can decouple the operator from the problem solution type;

• Increased Cohesion – By giving the responsibility of only implementing the operations and not handling solution types, the crossover and mutation operators do not need to have solution type checking and only implement their main functionality. This better responsibility assignment decreases the scattering of similar functionalities along the program;

• Prevents Code Duplication – Instead of creating a class for each combination of operator-representation and implementing similar code for each one, the developer has a single class for each operator and can reuse common code with shared methods. Reusing shared methods and not recoding them is what really prevents the code duplication;

• Easy and Dynamic Interchange of Operators and Solutions – This structure eases how operators and solutions are handled by enabling their free interchange using interfaces. Also, it enables a dynamic interchange of objects during runtime, which is specifically useful for online selection hyper-heuristics;

• Easy Addition of New Operators and Solutions – To add a new solution type or a new operator, the developer just needs to create a concrete class for the interfaces and instantiate it whenever needed;

• Methods Without Implementation – Because some operators are very specific for a single solution representation (e.g. Bit Flip Mutation [2]), these operators have the declaration of methods for other representations, but no implementation. This can lead to methods being called and no operation being executed, which hardens the debugging and maintenance of the algorithm. This is a disadvantage of this pattern;

V. Case Study

In order to demonstrate how the proposed solution can be employed in a real scenario, this section presents a case study

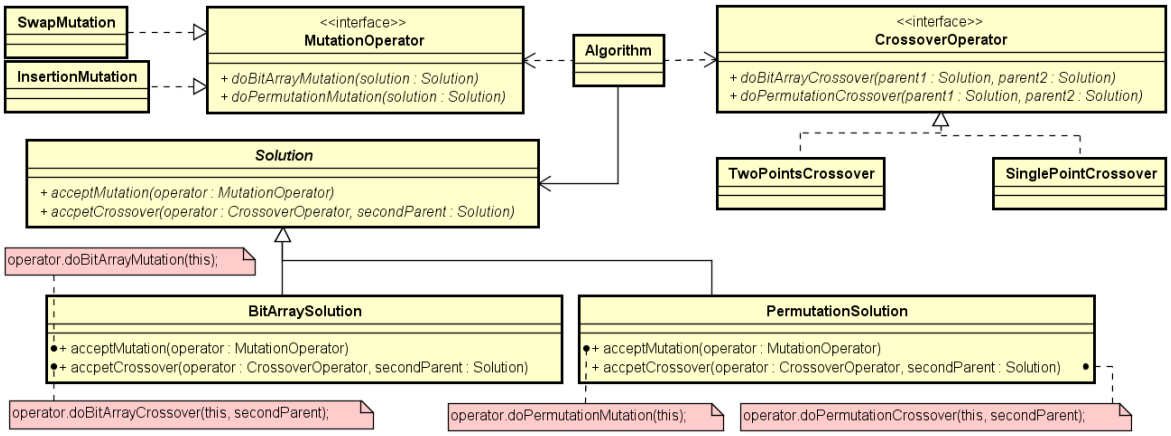

Fig. 2. Visitor Structure for GA Operators

conducted using the Hyper-heuristic for the Integration and Test Order Problem (HITO) proposed in [13]. We chose to present a case study using HITO and the ITO problem since it shows how the proposed pattern solution can improve the design of a hyper-heuristic based on GAs. In addition, HITO is a selection hyper-heuristic, uses several operators and is applied into two different representations, therefore it fits very well in the context of this paper. Furthermore, the ITO problem can be represented in many ways and has a variation (with clusters) that introduces different restrictions to the problem. Therefore, this problem is a good example to show how the change on its representation can be accommodated by the proposed solution. The next subsections present, respectively: i) the Integration and Test Order Problem; ii) HITO; and iii) how we applied Visitor with HITO.

A. The Integration and Test Order Problem

Briefly, for solving the Integration and Test Order (ITO) problem [18], the algorithm must find an order of units (the smallest part of a software, e.g., a class, a function or an aspect) to be integrated and tested. The cost of the integration and testing activities increases when a unit is integrated and tested after the units that require it, since in such situation a stub must be created to emulate this required unit. The issue with developing stubs is that, eventually, the units emulated by the stubs are developed and the stubs are discarded. Thus, the human effort allocated to the stubbing is simply wasted. By minimizing the stubbing cost, the cost of the whole software testing activity is also minimized.

The objective functions for this problem are designed to assess the cost of the required stubs. Some works just compute the number of stubs [19], however, other works such as [13], [18] use multi-objective approaches in order to increase the accuracy of the measurement and better guide the evolutionary optimization. In these works, the objective functions used are: i) A – Number of emulated attributes; ii) O – Number of emulated operations/methods; iii) R – Number of return types; and iv) P – Number of parameter types.

The representation of the problem is a simple permutation array of integers, where each number (gene) represents a unit. The order in which the genes appear in the chromosome is the order for the integration and testing of the respective units. As in other permutation problems, each number cannot repeat in the chromosome, because a unit must be integrated and tested only once. For this reason, the operators used for this problem must be adapted to comply with these restrictions.

There is also another instance of this problem. This instance is concerned with the integration of units in the presence of modularity constraints, introducing the idea of testing and integrating units in clusters [20]. A cluster can be a logical or an organizational grouping of units to define the ones that must be integrated and tested together. A common scenario is the distributed software development, where each unit cluster is assigned to one team. In addition, the units of a cluster are usually developed at the same time, thus it might be a good practice to test them at the same time too. For instance, if the developer must deploy a small release of the software, it would be wise to group the units of the requirements included in the plan for the release and test them before deploying.

Even though it appears to be a small change in the representation when compared to the problem without clusters, there are some issues that must be addressed by the genetic operators. One of them is to not mix units from different clusters, i.e., units of a cluster must stay in that cluster throughout the whole search process. In addition, not only the units have to be rearranged to decrease the testing cost, but also the search process must consider the ordering of the clusters. For example, a cluster with a lot of units required by units of different clusters would better fit at the start of the order. If these issues are not addressed, then the operators may generate infeasible solutions, which may delay the optimization and even decrease the quality of the results.

B. HITO

HITO is an online hyper-heuristic for dynamically selecting low-level heuristics for solving the Integration and Test Order Problem [13]. Each low-level heuristic is a composition of a crossover and a mutation operator, or only a crossover operator. These low-level heuristics are evaluated and selected at each mating, while the problem is being solved.

In order to evaluate the operators, HITO uses a function based solely on the Pareto dominance concept [1] that rewards

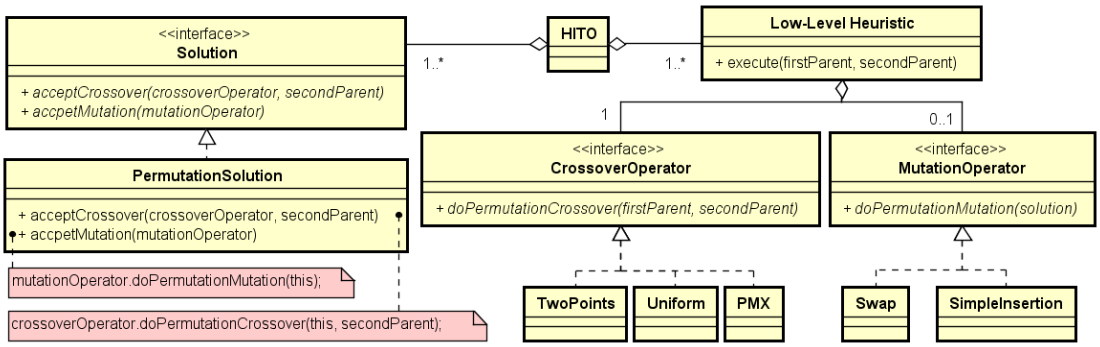

Fig. 3. HITO implemented with the Visitor design pattern

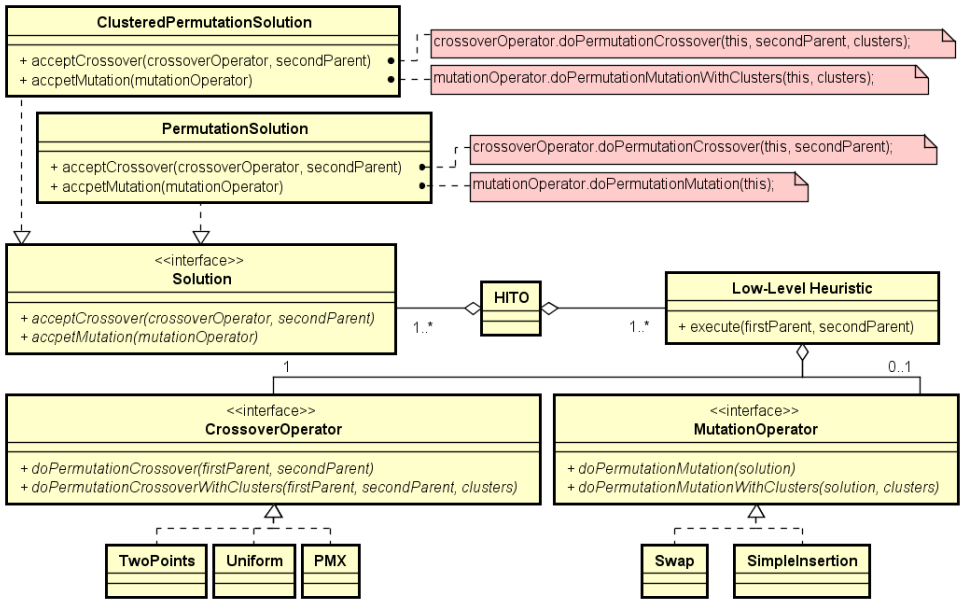

Fig. 4. HITO with both representations using Visitor

the heuristics when they generate good solutions, or punish them when they generate bad solutions. This performance value is later used as an exploitation factor by the Choice Function (CF) [21] to select the best heuristic to be applied. HITO also takes into account the number of matings since the last time each heuristic was applied, and uses this value as an exploration factor in CF.

In [13] we used 9 low-level heuristics, composed by three crossover operators and two mutation operators [2]: i) Two Points Crossover, PMX Crossover and Uniform Crossover; and ii) Swap Mutation and Simple Insertion Mutation. The representation is a permutation array of integer values and each operator was executed considering this. The PMX Crossover is an operator specific for permutation representations, which employs a more complex procedure when compared to the other operators. Therefore, it can be used in this problem and can depict how a specific operator can be included in a generic design such as the Visitor pattern.

C. Visitor Applied with HITO

To solve the ITO problem in the presence of modularity restrictions, it is necessary to use the HITO operators and also add the restrictions. To ease the implementation we used the solution based on the Visitor pattern, proposed before. Figure 3 presents the class diagram for the structure of HITO and the operators using the Visitor DP. For space reasons, some types and methods are omitted in the figure.

The structure presented in the figure is similar to Figure 2. HITO is the algorithm which has a set of low-level heuristics and a set of solutions. At each mating, HITO applies the low-level heuristic to two parents (although it can vary). The low-level heuristic then delegates the responsibility of the crossover to its aggregated crossover operator, and the responsibility of mutation to its aggregated mutation operator. However, because it is a Visitor structure, the low-level heuristic calls the methods “acceptCrossover()” and “acceptMutation()” of the interface “Solution”, instead of calling the methods of the operators directly. The concrete solution, in this case the class “PermutationSolution”, then accepts the operators by calling the methods “doPermutationCrossover()” and “doPermutation-Mutation()” for the parents and children respectively.

In this paper, we extended HITO to solve the integration and test order problem in the presence of modularity constraints. When dealing with clusters, the operators must behave

differently in order to not only reorder units, but also clusters. In order to do this, we had to add another representation to our framework. Figure 4 presents the new structure.

For this new structure, a new concrete class "ClusteredPermutationSolution" was created for the representation interface. This representation contains information about the clusters, which in turn are used by the operators to comply with the restrictions. Moreover, the interfaces "CrossoverOperator" and "MutationOperator" received new methods to visit this new kind of representation. Hence, all the concrete operators now implement methods to deal with clusters. These new methods are only called by the methods "acceptCrossover()" and "acceptMutation()" of the class "ClusteredPermutationSolution". Therefore, the previous functionality (optimization without clusters) remained the same and a new one was added without changing the main algorithm ("HITO" class).

D. Discussion

During the implementation, one significant easiness was the great reusability of the operators code. The existing methods were refactored and broken into smaller methods (omitted in the figure), which enabled reuse of common code. For instance, Two Points Crossover has procedures to cut an array in two points, swap subarrays between children and fill the remaining array positions, which all turned into new methods. Not only the addition of a new functionality was easier, but also the need of code reuse for the Visitor pattern drove us to better refactor our code.

When we executed HITO after the implementation, all we had to do were: i) instantiate the solution class with clusters; and ii) inform the clustering description to the solution. Because the operator classes are the same (the only change relies on which methods are invoked), even the low-level heuristic instantiation, performance update and selection were reused.

VI. Concluding Remarks

This work presented a solution based on the Visitor DP [8] to design genetic operators. This solution aims at decreasing the coupling between the problem representation and operators. The main benefits can be extended not only to other kind of evolutionary algorithms, but also to hyper-heuristics.

To demonstrate how the proposed solution can help to improve the design of genetic based algorithms, we presented a case study using HITO, a hyper-heuristic proposed in [13]. HITO was implemented with Visitor, and then extended to another variation of the same problem. Because the solution presented is scalable, the extension was easy to perform and demanded little effort. The main advantages observed were related to the easiness to add new operators and representations, while also providing a good code reuse.

As future work, we intend to study how other DPs can help to achieve a good design for meta- and hyper-heuristics. Even though HITO was proposed for the ITO problem, it does not mean that it cannot be applied to other problems. As future work, we intend to evaluate this possibility.

Acknowledgment

The authors would like to thank CAPES and CNPq.

References

[1] C. A. C. Coello, G. B. Lamont, and D. A. V. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*, 2nd ed., 2007.

[2] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*, 2003.

[3] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *J. of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[4] E. Alba, G. Luque, J. G. Nieto, G. Ordonez, and G. Leguizamon, "MALLBA: a software library to design efficient optimisation algorithms," *International Journal of Innovative Computing and Applications*, vol. 1, no. 1, p. 74, 2007.

[5] A. J. Nebro, J. J. Durillo, and M. Vergne, "Redesigning the jMetal Multi-Objective Optimization Framework," in *24th Genetic and Evolutionary Computation Conference*, 2015, pp. 1093–1100.

[6] A. V. Tsyganov and O. I. Bulychov, "Implementing Parallel Metaheuristic Optimization Framework Using Metaprogramming and Design Patterns," *Applied Mechanics and Materials*, vol. 263-266, pp. 1864–1873, 2012.

[7] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a Java framework for evolutionary computation," *Soft Computing*, vol. 12, no. 4, pp. 381–392, 2007.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, 1995.

[9] A. Patelli, N. Bencomo, A. Ekárt, H. Goldingay, and P. Lewis, "Two-B or not Two-B? Design Patterns for Hybrid Metaheuristics," in *24th Genetic and Evolutionary Computation Conference*, 2015, pp. 1269–1274.

[10] M. R. Wick and A. T. Phillips, "Comparing the template method and strategy design patterns in a genetic algorithm application," *ACM SIGCSE Bulletin*, vol. 34, no. 4, p. 76, 2002.

[11] J. Woodward, J. Swan, and S. Martin, "The 'composite' design pattern in metaheuristics," in *23rd Genetic and Evolutionary Computation Conference*, 2014, pp. 1439–1444.

[12] J. R. Woodward and J. Swan, "Template method hyper-heuristics," in *23rd Genetic and Evolutionary Computation Conference*, 2014, pp. 1437–1438.

[13] G. Guizzo, S. R. Vergilio, and A. T. R. Pozo, "Evaluating a Multi-Objective Hyper-Heuristic for the Integration and Test Order Problem," in *4th Brazilian Conference on Intelligent Systems*, 2015.

[14] M. A. Lones, "Metaheuristics in nature-inspired algorithms," in *23rd Genetic and Evolutionary Computation Conference*, 2014, pp. 1419–1422.

[15] V. Mannava and T. Ramesh, "Load Distribution Design Pattern for Genetic Algorithm Based Autonomic Systems," *Procedia Engineering*, vol. 38, pp. 1905–1915, 2012.

[16] G. R. Raidl, "Decomposition based hybrid metaheuristics," *European Journal of Operational Research*, vol. 244, pp. 66–76, 2015.

[17] J. L. Fernandez-Marquez, G. Di Marzo Serugendo, S. Montagna, M. Viroli, and J. L. Arcos, "Description and composition of bio-inspired design patterns: A complete overview," *Natural Computing*, vol. 12, no. 1, pp. 43–67, 2013.

[18] W. K. G. Assunção, T. E. Colanzi, S. R. Vergilio, and A. Pozo, "A multi-objective optimization approach for the integration and test order problem," *Information Sciences*, vol. 267, no. 0, pp. 119–139, 2014.

[19] Z. Wang, B. Li, L. Wang, and Q. Li, "A Brief Survey on Automatic Integration Test Order Generation," in *23rd Conference on Software Engineering and Knowledge Engineering*, 2011, pp. 254–257.

[20] W. Klewerton Guez Assunção, T. Colanzi, S. Vergilio, and A. Pozo, "Determining Integration and Test Orders in the Presence of Modularization Restrictions," in *27th Brazilian Symposium on Software Engineering*, Oct. 2013, pp. 31–40.

[21] M. Maashi, E. Özcan, and G. Kendall, "A multi-objective hyper-heuristic based on choice function," *Expert Systems with Applications*, vol. 41, no. 9, pp. 4475–4493, 2014.