About Designing an Observer Pattern-Based Architecture for a Multi-objective Metaheuristic Optimization Framework

Antonio Benítez-Hidalgo[1], Antonio J. Nebro[1(✉)], Juan J. Durillo[2], José García-Nieto[1], Esteban López-Camacho[1], Cristóbal Barba-González[1], and José F. Aldana-Montes[1]

[1] Dept. de Lenguajes y Ciencias de la Computación, University of Malaga, Campus de Teatinos, 29071 Malaga, Spain
antonio.b@uma.es, {antonio,jnieto,esteban,cbarba,jfam}@lcc.uma.es
[2] Leibniz Supercomputing Centre, Munich, Germany
juanjod@gmail.com

Abstract. Multi-objective optimization with metaheuristics is an active and popular research field which is supported by the availability of software frameworks providing algorithms, benchmark problems, quality indicators and other related components. Most of these tools follow a monolithic architecture that frequently leads to a lack of flexibility when a user intends to add new features to the included algorithms. In this paper, we explore a different approach by designing a component-based architecture for a multi-objective optimization framework based on the observer pattern. In this architecture, most of the algorithmic components are observable entities that naturally allows to register a number of observers. This way, a metaheuristic is composed of a set of observable and observer elements, which can be easily extended without requiring to modify the algorithm. We have developed a prototype of this architecture and implemented the NSGA-II evolutionary algorithm on top of it as a case study. Our analysis confirms the improvement of flexibility using this architecture, pointing out the requirements it imposes and how performance is affected when adopting it.

Keywords: Multi-objective optimization · Metaheuristics
Software framework · Software architecture · Observer pattern

1 Introduction

Most of real-world optimization problems can be formulated as minimizing or maximizing two or more conflicting functions simultaneously, so the result of optimizing them is not a unique solution but a set of trade-off solutions known as Pareto optimal set. In practice, obtaining this set is frequently unfeasible, so

non-exact techniques providing an approximation of it are commonly used, being metaheuristics the most popular ones [3,4]. Metaheuristics comprise a family of approximate optimization algorithms including evolutionary algorithms, particle swam optimization, ant colony optimization, and many others [2].

Multi-objective optimization with metaheuristics is a very active research field since year 2000, which has been supported by the development and availability of software frameworks that not only provide implementations of state-of-the-art algorithms, but also benchmark problems, quality indicators, support for performance assessment, visualization tools, etc. Examples of these frameworks are PISA [1] (implemented in the C language), Paradiseo-MOEO [12], jMetalCpp [13]) (both implemented in C++), jMetal [6], MOEAFramework [10] (both written in Java), and Inspyred [9] and Platypus [11] (both written in Python).

In general, these frameworks have a monolithic architecture around which metaheuristics are implemented. Depending on their designs (all the aforementioned frameworks but PISA follow an object-oriented approach), they offer a certain degree of flexibility. Frequently, however, significant code changes are required to implement variants of existing algorithms or to add new components to, for example, inspect or analyze the internal behavior of an algorithm during the search.

Our motivation comes from our experience with the jMetal framework. The jMetal project started in 2006, although the framework was redesigned from scratch in 2015 [14] to improve its architecture and to make it more flexible and extensible. Remarkable included features are the provision of metaheuristic templates and the possibility of getting measurements of algorithm-specific information during its execution. Measures are based on the application of the observer pattern [8] and offers pull and push requests to get algorithm data.

Here we explore the design of a multi-objective metaheuristics framework whose components are all based on the observer pattern. In particular, there will be three entities, namely, observer, observable, and observer/observable which can be combined for implementing multi-objective optimization algorithms. We analyze the advantages and drawbacks of this approach by using jMetal as base platform.

The rest of this paper is organized as follows. Section 2 describes the observer pattern, which is the basis of the proposed architecture that is explained in Sect. 3. Section 4 contains some implementation details. We discuss the implications of using the observer-based pattern architecture in Sect. 5. Finally, we present the conclusions and lines of future work in Sect. 6.

2 The Observer Pattern

The book of Gamma *et al.* [8] popularized the concept of design patterns, which can be defined as general and reusable solutions to recurrent design situations happening in software design. A design pattern describes a problem to be solved, a solution to that problem, when to apply it, and its consequences. According to [8], design patterns can be classified in three categories: creational, structural and behavioral. The observer pattern belongs to the latter family.

The observer pattern allows modelling one-to-many relationships among objects, in such a way that when an object (observable) changes its state, the objects depending on it (observers) get notified and updated automatically. This requires to register the observers on the observables beforehand. The pattern is useful in many contexts, and it is included in many libraries and programming languages such as Java.

A simplified UML class diagram representing the pattern is included in Fig. 1. We can see that `Observer` is an interface containing an `update()` method while `Observable` is a class containing two public methods: one to register/unregister observers and a private method to notify registered observers that the state has changed. This method merely invokes the `update()` method of the observers.

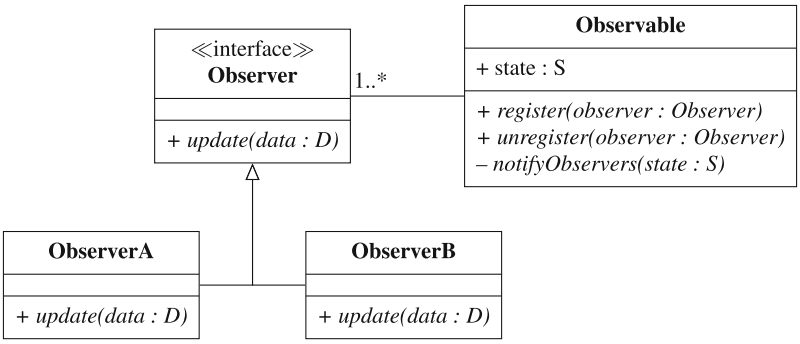

Fig. 1. Simplified UML class diagram representing the observer pattern.

3 Proposed Architecture

In this section, we propose an observer-based architecture for multi-objective metaheuristics. Without loss of generality, we focus on evolutionary algorithms, which are the most widely known and used multi-objective metaheuristics; in particular, we use the multi-objective NSGA-II [5] algorithm as a case of study.

An evolutionary algorithm follows the pseudo-code shown in Algorithm 1. The first step creates an initial population $P(0)$ which is evaluated (line 3) before starting the main loop. An iteration counter t is also initialized (line 2). The evolution process consists in iteratively selecting a mating pool of solutions M from $P(t)$ (line 5), which is used for reproduction, leading to an offspring population $Q(t)$ (line 6). This population is evaluated (line 7) and a replacement strategy is applied to get the population at the next generation $P(t+1)$ by combining $P(t)$ and $Q(t)$ (line 8). The iteration counter t is updated at the end of the loop (line 9). The algorithm stops when a finishing condition is met (line 4).

Algorithm 1. Pseudo-code of an evolutionary algorithm

```
1: P(0) ← InitialPopulationGeneration()
2: t ← 0
3: Evaluation(P(0))
4: while not StoppingCriterion() do
5:    M(t) ← Selection(P(t))
6:    Q(t) ← Reproduction(M(t))
7:    Evaluation(Q(t))
8:    P(t + 1) ← Replacement(P(t), Q(t))
9:    t ← t + 1
10: end while
```

3.1 Algorithm Templates in jMetal

The issue we explore in this section is how to implement the pseudo-code of Algorithm 1 to develop evolutionary algorithms. The solution adopted by jMetal is to provide a template in the form of an `AbstractEvolutionaryAlgorithm` class that closely mimics the pseudo-code:

```
@Override public void run() {
    List<S> offspringPopulation;
    List<S> matingPopulation;

    population = createInitialPopulation();
    population = evaluatePopulation(population);
    initProgress();
    while (!isStoppingConditionReached()) {
        matingPopulation = selection(population);
        offspringPopulation = reproduction(matingPopulation);
        offspringPopulation = evaluatePopulation(offspringPopulation);
        population = replacement(population, offspringPopulation);
        updateProgress();
    }
}
```

This way, an evolutionary algorithm can be implemented by extending the abstract class, i.e., by implementing all its methods, as is the case of NSGA-II.

We analyze next three different situations that can arise in practice: modify the default behavior of an algorithm, extend it to provide information during its execution (e.g., to display the current front), and adding new features such as an external archive to store all the non-dominated solutions found during search.

By using the abstract class scheme, the natural way to modify the behavior of a base algorithm is by defining subclasses of it by applying inheritance. An example is to change the stopping condition. In the default implementation of NSGA-II in jMetal, the algorithm stops when a maximum number of evaluations have been performed. Changing this condition to, for example, stop after a given time limit instead, requires to write a new subclass which simply overrides the

`isStopingConditionReached()` method. This solution is simple but introduces a potential hazard: confusing users of that algorithm with many subclasses that provide slightly different behaviours. Furthermore, this new stopping condition cannot be shared by other algorithms.

By default, when an algorithm is configured and executed in jMetal, it does not show any information until it finishes. At that point it generates two files containing the solutions and the Pareto front approximation found. We might be interested, however, in getting run-time information (the iteration number, computing time, population at each iteration, etc.) for writing it to files for further analysis or plotting graphs. This can be achieved in jMetal by defining measures, which require to create a new subclass redefining the `updateProgress()` method, as measures are updated in general at the end of each algorithm iteration. As before, if different measures are needed, new subclasses need to be defined.

There may be situations where we can be interested in adding all the evaluated solutions in NSGA-II to an external archive. A reason is that NSGA-II deletes some non-dominated solutions that later could be useful, so adding them to an external unbounded archive would keep all of them. Incorporating external archives to NSGA-II can be achieved by redefining the `evaluatePopulation()` method.

Summarizing, adopting an abstract class scheme for designing evolutionary algorithms provides enough flexibility at the cost of populating the framework with a large set of variants as minor changes of the default algorithms' behaviors imply to redefine some of the methods of the template.

3.2 Observer Pattern-Based Architecture

Our alternative to the monolithic template-based approach is to decouple all the algorithm components in separate entities featuring one of three possible behaviors: observable, observer, and observer/observable. This way, a metaheuristic within this new approach is composed of components that follow the observer pattern.

The first column of Table 1 shows the components we have considered for a first prototype implementation of NSGA-II. These components can be considered as interfaces that can be implemented in many ways; the second column of the table shows the implementations currently provided. The only pure observable entity is the *CreateInitialPopulation* interface, while the rest of components of an evolutionary algorithm are both observer and observable entities. The observer category includes a *PopulationObserver* interface, representing entities that process populations somehow; in particular, we include observers for writing a population in files, storing a population in an external archive, and plotting a front.

Figure 2 shows how the aforementioned components are linked to provide an implementation of NSGA-II. All these elements, except the one that creates the initial population, must register the previous component from which it will receive data as soon as it is available. It is noteworthy that two instances of

Table 1. Entities following the observer pattern for implementing an evolutionary algorithm (NSGA-II) in jMetal.

Observable	Implementations
CreateInitialPopulation	RandomPopulationCreation
Observer/observable	Implementations
Selection	BinaryTournamentSelection
Evaluation	SequentialEvaluation
	MultithreadedEvaluation
Replacement	RankingAndCrowdingReplacement
	RankingAndHypervolumeContributionReplacement
Termination	TerminationByEvaluations
	TerminationByTime
Variation	CrossoverAndMutationVariation
Observer	Implementations
PopulationObserver	PopulationToFilesWriter
	ExternalArchiveObserver
	ParetoFrontPlot

the *Evaluation* interface are needed, as the evaluation of the initial population is observed by the *Termination* entity, while the evaluation after the variation step is observed by the replacement component. The result is a workflow where a population of solutions, which is created by the pure observable element, is processed by each component and the resulting population is notified to the registered observers.

The behavior of the components of this observer-based implementation is as follows:

1. The *CreateInitialPopulation* produces the initial population, which is notified to the *Evaluation* component. This population has a number of attributes, including an evaluation counter (initialized to 0), a flag indicating whether the computation has finished or not (initialized to False), and the start time of the computation. The population and its attributes constitute the observable data that are notified to the registered observers.
2. The first *Evaluation* component carries out the evaluation of all the individuals of the population and updates the evaluation counter attribute.
3. The *Termination* component receives the population and checks for the stopping condition by examining the corresponding attributes. If the condition is fulfilled, the attribute in the population indicating whether the computation has finished is set to True. The population and its attributes are notified to the observers.

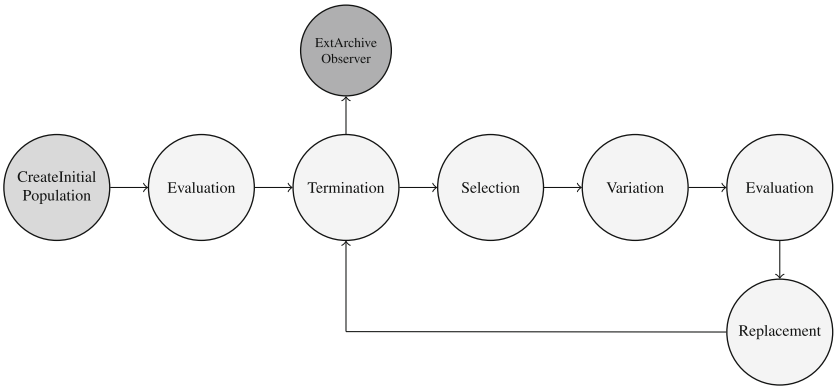

Fig. 2. Architecture of the NSGA-II evolutionary algorithm using the observer pattern. Components filled with light grey () are both observer and observable; components filled with grey () are only observable, and components filled with dark grey () are only observers.

4. The *Selection* component takes the population and generates an offspring population by applying a given selection operator (i.e., binary tournament). This offspring population is attached to the population as another attribute.
5. The *Variation* component receives the population and applies some variation operators (typically, crossover and mutation) to the offspring population. This leads to a new offspring population that replaces the original one.
6. The second *Evaluation* component checks whether the population has an offspring population attribute. If so, this last one is evaluated and the evaluation counter is updated; else, it behaves like indicated in step 2.
7. The *Replacement* component takes the population and the offspring population and creates a new population by applying some replacement strategy (e.g., ranking and crowding in the case of NSGA-II), and the offspring population attribute is eliminated.

All the observer/observable components check their termination attribute every time they are notified a population. In case of being `true`, a component notifies the population to its registered observers and finishes. In the case of observers, they also make the same check. This way, all the components finish in an ordered way.

4 Implementation Details

We have implemented a prototype of the observer-pattern architecture in jMetal. The main issue to deal with is how to implement the `Observable` class, as the *Observer* interface only requires to include the `update()` method in the classes implementing it.

A direct implementation of the observer pattern, where the observable objects directly invoke the update method of the observers, can lead to a stack overflow

problem. Our approach has been to associate a thread to each component, so all of them are concurrent entities, and use a bounded buffer (with a capacity for only one element) to apply the producer/consumer concurrent scheme. This way, observers are waiting on their buffers by invoking a get operation and, when observables make a notification and invoke the `update()` method of its registered observers, they put the observable data (i.e, populations and their attributes) into the buffer, thus waking up the observers. To avoid concurrent accesses, the observed data are copied before being inserted into the buffers.

It is worth mentioning that other implementations are possible. Another possibility is to use a message passing library, what would allow to deploy the algorithm components in different nodes of a distributed systems (i.e., a cluster). This way, the most compute intensive components (typically, the evaluators) could be deployed in nodes having highest performance processors.

5 Discussion

We discuss in this section the requirements the observer-based approach imposes and how they are dealt with. We focus on the observable data, the manipulation of global data, the flexibility of the architecture, and the performance implications.

Any observable entity must offer a public interface to its potential observers defining the data that will be provided. The data always include a population and a number of attributes, which are used and updated somehow by each component. As a consequence, all the components must state clearly what they are waiting for and what they produce; otherwise, the resulting algorithm will fail.

All the components are independent entities that do not share a common memory space where global data, like the evaluation counter, can be stored. As a consequence, there is a need of using attributes which are attached to the population in each step, which can be a major issue in case of algorithms using global coarse grained shared data structures. This matter needs further development and that will determine the viability of the observer-based approach.

Our main motivation in this study is to propose a modular and flexible architecture that allows to extend an existing evolutionary algorithm without the need of modifying the current implementations nor applying inheritance to create specialized subclasses. The NSGA-II version based on this scheme (shown in Fig. 2) can be extended in many ways:

- Adding an external unbounded archive to store the non-dominated solutions found, which can be achieved by including an external archive observer and registering it to the *Evaluation* components. An external bounded archive could be also incorporated the same way.
- Changing the stopping condition (e.g., by time, by the number of evaluations, interactively by the user, etc.) only requires to change the *Termination* component.
- A graphical observer can be registered into the *Replacement* component for visualizing the current population in real-time.

- A number of population observers can be attached to the *Termination* component to write the final population in files, to generate plots with the obtained Pareto front approximations, or just to assess the quality of the front by applying a quality indicator.
- Implementing the SMS-EMOA [7] evolutionary algorithm, which is based on NSGA-II, only requires to change the *Replacement* component by one computing the ranking and hypervolume contribution of the solutions of the population.

It is worth highlighting that these changes only require to add more observers, add more observers/observables, or replace some of the components by others. The newly created components can also be used by other algorithms, thus promoting code reuse.

Table 2. Running times of NSGA-II implementation in jMetal and the one using the observer pattern based architecture (the column names represent pairs population size/number of evaluations). The target problem is ZDT1.

	100/25000	200/50000
NSGA-II (monolythic)	782 ms	1422 ms
NSGA-II (observer)	1046 ms	1768 ms
SMS-EMOA (monolythic)	32586 ms	478570 ms
SMS-EMOA (observer)	34685 ms	479237 ms

A consequence of using the proposed architecture instead of a monolithic one is a performance overhead as observed data must be copied when observers are notified by observables. To quantify that overhead, we have run the NSGA-II version included in jMetal and the one developed using the observer pattern with two configurations (population sizes of 100/200 and number of evaluations of 25000/50000) are shown in Table 2[1]. The obtained times reveal a time overhead of 120–350 ms, which is only a fraction of the overall wall time even for a problem such as ZDT1 which can be evaluated in a very short time. In case of solving a real-world problem, the differences in running time should be negligible. We also include in Table 2 the execution times of the SMS-EMOA algorithm with a population size of 100 and 25000 function evaluations. The times used for monolythic and observer versions are similar, since in this algorithm, most of the computing time is spent in the replacement step.

6 Conclusions

We have presented in this paper a study about the design of an observer pattern-based architecture for a framework for multi-objective metaheuristics. In this

[1] MacBook Pro, 2.2 GHz Intel Core i7, macOS 10.13.4, Java SE 1.8.0_101, jMetal 5.5.

architecture, all the algorithm components are classified into three categories of components: observable, observer/observable, and observer. By taking a multi-objective evolutionary algorithm, NSGA-II, as a case of study, we have shown how it can be implemented using the proposed scheme.

Our analysis indicates that the resulting implementation is very flexible, allowing to develop new variants of an algorithm by simply adding or replacing components. The computing times of two algorithms using the proposed architecture indicate a minimal time overhead regarding the original monolithic-based implementations.

The future research work is in the line of validating the observer-based architecture to cope with other multi-objective algorithms which are not extensions of NSGA-II, such as MOEA/D, and with non-evolutionary metaheuristics, such as particle swarm optimization algorithms.

Acknowledgements. This work was partially funded by Grants, TIN2017-86049-R, TIN2014-58304 (Spanish Ministry of Education and Science) and P12-TIC-1519 (Plan Andaluz de Investigación, Desarrollo e Innovación). Cristóbal Barba-González was supported by Grant BES-2015-072209 (Spanish Ministry of Economy and Competitiveness). José García-Nieto is the recipient of a Post-Doctoral fellowship of "Captación de Talento para la Investigación" Plan Propio at Universidad de Málaga.

References

1. Bleuler, S., Laumanns, M., Thiele, L., Zitzler, E.: PISA – a platform and programming language independent interface for search algorithms. In: Fonseca, C.M., Fleming, P.J., Zitzler, E., Deb, K., Thiele, L. (eds.) Evolutionary Multi-Criterion Optimization (EMO 2003). Lecture Notes in Computer Science, pp. 494–508. Springer, Heidelberg (2003)
2. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput. Surv. **35**(3), 268–308 (2003)
3. Coello, C., Lamont, G.B., van Veldhuizen, D.A.: Multi-objective Optimization Using Evolutionary Algorithms, 2nd edn. Wiley, New York (2007)
4. Deb, K.: Multi-objective Optimization Using Evolutionary Algorithms. Wiley, New York (2001)
5. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
6. Durillo, J.J., Nebro, A.J.: jMetal: a Java framework for multi-objective optimization. Adv. Eng. Softw. **42**(10), 760–771 (2011)
7. Emmerich, M., Beume, N., Naujoks, B.: An EMO algorithm using the hypervolume measure as selection criterion. In: Coello, C.A., Hernández, A., Zitler, E. (eds.) Third International Conference on Evolutionary MultiCriterion Optimization, EMO 2005. LNCS, vol. 3410, pp. 62–76. Springer (2005)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Professional, Reading (1994)
9. Garrett, A.: Inspyred. http://aarongarrett.github.io/inspyred/. Accessed 02 May 2018
10. Hadka, D.: MOEAFramework. http://moeaframework.org/. Accessed 02 May 2018

11. Hadka, D.: Platypus. http://platypus.readthedocs.io/en/latest/. Accessed 02 May 2018
12. Liefooghe, A., Jourdan, L., Talbi, E.-G.: A software framework based on a conceptual unified model for evolutionary multiobjective optimization: ParadisEO-MOEO. Eur. J. Oper. Res. **209**(2), 104–112 (2011)
13. López-Camacho, E., García-Godoy, M.J., Nebro, A.J., Aldana-Montes, J.F.: jMetalCpp: optimizing molecular docking problems with a C++ metaheuristic framework. Bioinformatics **30**(3), 437–438 (2014)
14. Nebro, A.J., Durillo, J.J., Vergne, M.: Redesigning the jMetal multi-objective optimization framework. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO Companion 2015, pp. 1093–1100. ACM, New York (2015)