Towards an Object-Oriented Pattern Proposal for Heuristic Structures of Diverse Abstraction Levels

Enrique Urra  
Escuela de Ingeniería en Informática  
Pontificia Universidad Católica de Valparaíso  
Av. Brasil 2950, Valparaíso, Chile  
Email: enrique.urra.c@mail.pucv.cl

Daniel Cabrera-Paniagua  
Escuela de Ingeniería Comercial  
Universidad de Valparaíso  
Pasaje La Paz 1301, Viña del Mar, Chile  
Email: daniel.cabrera@uv.cl

Claudio Cubillos  
Escuela de Ingeniería en Informática  
Pontificia Universidad Católica de Valparaíso  
Av. Brasil 2950, Valparaíso, Chile  
Email: claudio.cubillos@ucv.cl

Abstract—In the optimisation field, the term heuristic is associated with mechanisms for problem solving, ranging from simple algorithms to complex learning techniques. Recent research has focused on developing more appropriate environments for the design and implementation of heuristics. Particularly, the software design consideration is restricted only to the application of design patterns; a detailed discussion regarding methodological background and architectural design approaches is not adequately considered. In this work, we want to discuss software design issues, from an object-oriented perspective, which can be useful to develop heuristics methods considering different abstraction levels, ranging from specialized components to more general-purpose architectures. A theoretical algorithmic model is presented, which forms the basis for a design pattern proposal named Flowchart pattern. We provide a case study of a new heuristic construction framework that uses the pattern at its core, and we discuss how such tool has been used in the implementation of a comprehensive hyperheuristic architecture. The framework usage and the modular structure provided by the hyperheuristic architecture demonstrates how the pattern allows to construct objectual representations of algorithms, and the main consequence is the direct decoupling of an algorithm’s structure, its logic behaviour and the data that it treats, which allows for the development of highly dynamic structures that can be modified even at runtime. This approach may open new alternatives in which applied optimisation and software design meet.

Keywords—design pattern, heuristic methods, optimisation, framework, software design.

I. INTRODUCTION

The term heuristic has broad significance in different areas of science. In the field of optimisation, it is associated with behavioural mechanisms for problem solving, which range from simple algorithms to complex and intelligent learning techniques. As the number of heuristic approaches increases, the need for classifying them according to their abstraction level becomes stronger. There is a basic level at which heuristic methods are designed and implemented in a straightforward manner using extensive problem domain knowledge. The typical results are simple and less-sophisticated algorithms that can solve the problem with a solution-quality cost. Regarding more abstract approaches, there are two types of heuristic methods: metaheuristics [1], [2] and hyperheuristics [3], [4], the latter of which was developed more recently than the former. The prefix of each term adequately describes its operation and design. We can understand “meta” as abstraction, whereas “hyper” can be understood as highness. From a design perspective, metaheuristic have evolved into the construction of abstract models that must be adapted using extensive problem knowledge before the related algorithms can operate efficiently. This approach provides opportunities for incorporating sophisticated design elements into the algorithms, at the cost of increasing the coupling between the solver and the problem domain. Such blending makes the implementation highly efficient, but hard to reuse, even for the related solver components. This lack of reusability makes the applicability of metaheuristics a complex task in the industry context because the resources for the deployment effort are often highly restricted. In contrast, the design of hyperheuristics does not directly rely on abstraction but rather on modularity and extensibility. The solver mechanisms of such methods do not require modification, but highly specific problem domain layers must be designed and “plugged” into the solver.

The issues described above have been discussed previously in the optimisation research literature. Consequently, diverse technologies and solution paradigms have been developed to make heuristic methods more accessible and easy to deploy in industry contexts. For example, hyperheuristics research focuses on the modularity of problem domains regarding the solvers, autonomous search [5], [6] aims for self-adaptation of solving systems through their internal components and simple implicit configuration requirements for users, automated algorithm configuration [7], [8], [9] is a concrete approach for determine the solver parameters in a user-friendly manner, algorithm engineering [10], [11] defines a practical methodology context for algorithmic research, and algorithm synthesis [12], [13], [14] provides automated methods to generate solver implementations based on high-level semantics and languages. All these approaches make contributions from the software engineering perspective of heuristic methods, and they should be considered not only as solver programs but also as concrete software that constantly develops in dynamic environments in which different types of users, not only optimization-expert, are involved.

In this paper we address the software design perspective of solver implementations, which defines how their components are designed and used in different problem contexts. Particularly, we consider scenarios for implementation effort prioritisation, in which resources expended in the construction and deployment processes are highly valued and conserved by accepting a reduced, but adequate, quality solution. Such applications are referred to as good enough - soon enough - cheap enough scenarios [3], and their requirements usually are highly dynamic because the environments are dynamic at
the same time; therefore, the implementation of a particular algorithm can change or may be reused, even for simple algorithms. How should heuristic components be designed to perform well in such scenarios? What type of design problems and decisions must be faced when a heuristic method is implemented? To our knowledge, there are no proposals in the literature that directly discuss these issues from the software design perspective. In this work, we attempt to address these issues. We make two primary contributions: i) we present a discussion from the object-oriented (OO) design perspective, of common issues that are faced when heuristic methods that support diverse abstraction levels must be constructed. In response, an initial version of a theoretical algorithmic model is presented. Then, ii) we formalise the proposed guidelines through a concrete OO-design pattern, named the flowchart pattern, through which an objectual representation of an algorithm can be constructed. The flowchart representation enables the development of highly dynamic structures that can be modified even at runtime. In addition to automated building tools, this pattern may aid the development of useful heuristic design and implementation environments based on software design principles and leverage the principles’ advantages at many levels.

The structure of this paper is as follows: in Section II, discussion regarding heuristic design is presented, alongside the theoretical algorithmic model mentioned above; the latter form a basis for the pattern proposal presented in Section III. A case study, which corresponds to the implementation of a hyperheuristic model through a new framework based on the pattern is described in Section IV. Finally in Section V the conclusions of our work are presented.

II. BUILDING MODULAR HEURISTIC STRUCTURES

The design and implementation of heuristics methods represent an interesting context in which software design topics can be studied. Particularly in this context, the CPU time and solution quality performance balance represents a critical consideration when algorithms are constructed. Consider a situation in which a particular heuristic method must be implemented to solve a highly complex problem: one can construct the design in a straightforward manner, without necessarily considering non-functional requirements related to the design itself, and ensure that the implementation will perform faster and better from the results perspective. Certainly, the related requirements will be primarily static, and the implementation will be made-to-measure. Therefore, this can be an adequate approach if sufficient resources are available. However, the situation is different when performance is not as relevant as non-functional requirements related to the design. Sometimes, we must obtain adequate results in reasonable execution times, but the design of our solutions, or their components, must consider modularity, reusability, and extensibility, for example. Because we constantly address different problems, or face changing functional requirements, it is desirable that our solving expertise can be properly adapted to all contexts, and it is necessary to make a “sacrifice” in terms of the result’s performance in favor of software design principles. Such trade-offs are usually inevitable and represent a balancing process that we must consider in the aforementioned situations, which we will use as the basis for our further discussion.

The modularity of heuristic methods is highly relevant to their ability to be reused. A first approach to modular design is directly related to the software components through which the methods are designed. However, the conceptual background in which those components are based is another important issue that must be considered. Sometimes, heuristic methods can be difficult to understand because it is common to include extensive technical knowledge, which is primarily related to optimisation issues and the designer’s expertise, within implementations. Such cases demand extra effort to learn and understand the related tools and thus clearly reduces the heuristic methods’ applicability, which may not be a deliberate consequence. There are a number of libraries that offer reusable components to build heuristic methods; the libraries primarily focus on the metaheuristic context. In [15], a comprehensive review and benchmark regarding them is presented. Additionally, there are a few implementations in the hyperheuristic context [16], [17]. Commonly, conceptual elements related to optimisation are embedded into the application programming interface (API) offered by them. For example, ECJ [18] provide diverse components related to evolutionary computation (EC), such as “Population”, “Species”, and “Selection Method”. Because of this structure, it can be difficult to reuse some of the logic implemented through those classes in another context in which an approach different from EC must be used. This issue is usually related to the heuristic abstraction level at which a particular library works. Because the common target level is the metaheuristic level, it is difficult to visualise a direct adaptation of the related components to more complex architectures, such as the hyperheuristic level. Perhaps particular frameworks with an API less coupled to metaheuristic concepts, such as EasyLocal++ [19] and HeuristicLab [20], are more suitable for such extensions.

Beyond the coupling between optimisation-related concepts and the library interfaces, the software design perspective in most of the heuristic method frameworks is restricted only to the application of design patterns; a broader discussion regarding methodological background and architectural design approaches is not considered. Only the work related to HeuristicLab addresses some topics regarding the design process [21], and although the framework provides interesting tools from the usability perspective, an OO-centred review of its architecture design has not been presented. This issue is more clear in the software engineering best practices dimension (C5.4) discussed in [15], particularly regarding the dependency injection topic [22]. Dependency injection techniques are only possible through the use of highly abstract layers, on which both the low-level and high-level layers depend, as the dependency inversion principle states [23]. Of existing solutions, only Opt4j [24] provides support for those features. Thus, most frameworks do not properly abstract their core layers, nor do they leverage such an abstraction level. This is actually an architectural design issue whose relevance is directly related to the possibility of further enhancing the provided libraries with more specialised tools, not only in the plug-in context but even in the architectural integration context. Some of the ideas developed through the course of this work have been formalised into a theoretical algorithmic model that is directly related to the modularity concepts that an OO-based architecture for constructing heuristics should provide. It is important to remark that the model considers the elements...
discussed above, i.e., we focus on designing heuristics through a simple conceptual approach that can be understood outside the optimisation context, and we want to address different heuristic abstraction levels such that one could implement concrete components and they could be reused on more general-purpose architectures. The model, which forms the basis of our proposal of a related design pattern, is shown in Figure 1. There are three main algorithm aspects considered in the model:

- **Algorithm logic**, represented by Operator components. They represent highly specialized logic “containers”, which implement any kind of procedure through the `doOperation()` method, ranging from generic purpose to domain-related logic. The whole set of Operator implementations used by a particular algorithm can form its complete logical basis. Many of such implementations may require some type of data to perform (which is explained in the following). Operators should particularly promote reuse: while more context-independent their logic is, the more reusable is through different domains, for example, a simple iteration handler operator could be used on any algorithm that require process repetition, but a vehicle distance calculation may be used only in algorithms related to transportation problems.

- **Algorithm data**, represented by three elements in the model: data interfaces, data objects and data setters. Data interface components (DataInterface) allow to store and retrieve any kind of data required by different Operator implementations in the algorithm, or moreover, to perform any kind of logic directly related to the managed data. The idea behind defining these components as interfaces, is their effective decoupling from the operators, that is, decouple the data from the logic in the algorithm, making both independently maintainable. Therefore, the data setter components (DataSetter) help to assign data interface implementations, the data object components (DataObject), to operator implementations through an interface injection-like structure [22]. Any operator that require to use a data interface to perform its logic should implement a data setter related to such data interface.

- **Algorithm structure**, which represents how the logical components of the algorithm are structured within the execution flow. This element is commonly defined by different control structures that we use in any programming language, which include the decision-based ones (if, switch, etc.) and the iterative ones (while, for, etc.). Such constructs allow to change the execution flow in different ways, according to the current algorithm state (its data). In the figure, the algorithm structure is defined by the Step interface, which represents a single “execution step” within the algorithm. It must return the next step in the execution, whenever it can be determined. The effective step sequence, which represent the algorithm structure itself, depends on the step implementations. For example, a DirectStep calls an operator and return the next step directly, while a BooleanStep calls an operator and uses its execution return value as the criterion to return one of two possible steps. The BooleanStep is an example of conditional operations within an algorithm structure. The two step types described above are composed using an Operation reference, without coupling them with the particular operator implementation used. Additionally, there is an Algorithm class that holds a reference to the first step in the related algorithm. Through the `run()` method, such a class executes the initial step and then retrieves the next step to be returned. Consequently, it executes the
Fig. 2. An object diagram based on Figure 1. Two different heuristic structures (h1 and h2) can be defined using the same operator components (opA to opD).

It is important to note the modeling approach that this design promotes. With the \textit{Step} and \textit{Operation} interfaces, a direct decoupling of the algorithmic structure of the heuristic and the heuristic logic components is generated. In this manner, we could model different operators components that can implement specific or general purpose logic, without considering the manner in which they would be combined in an algorithmic structure. Such relationships could be further modeled through steps in many different manners simply by reusing the defined operation components. For example, the class model proposed in Figure 1 does not require a change if a new algorithmic structure is needed; rather, only the object model must be changed, and this could be performed at runtime because of the low coupling among the components. An object diagram based on the same design is presented in Figure 2, which illustrates the above idea. The heuristic h1 is the complete inverse of h2 in terms of the operation execution order; however, both approaches use the same set of \textit{Operation} objects.

\section{III. The Flowchart Pattern}

The model presented in Section II represent an implementation approach for our pattern proposal, the \textit{flowchart pattern}. The name is self-explanatory: through it, an objectual representation of an algorithm can be constructed, and such a representation resembles a flowchart. Figure 3 demonstrates how the objects of the algorithm h1 from Figure 2 can be illustrated using such a diagram.

In the following, we formalise the pattern proposal by using common definition sections in the same manner as the well-known Gamma proposals [25].

\subsection*{A. Intent}

The flowchart pattern offers improved flexibility in the construction of algorithms at different abstraction levels, by providing a proper interface to define the execution structure independently from the internal logic of its components and to allow dynamic manipulation of both the execution structure and the logic.

\subsection*{B. Structure}

The pattern structure is illustrated in Figure 4. It can be considered a generalisation of the example presented in Figure 1.

The participants of the pattern are described below:

\begin{itemize}
  \item \textbf{AbstractStep (Step)}
    \begin{itemize}
      \item This class declares an interface for different step types that can be used within an algorithmic structure.
      \item Any step must determine the next step to be executed within the execution flow through the \textit{doStep()} method. Such a method must return the reference of the following step.
      \item A chain of steps that sequentially returns references to other steps defines the execution structure.
    \end{itemize}
  \item \textbf{ConcreteStep (DirectStep, BooleanStep, \ldots)}
    \begin{itemize}
      \item This class implements a concrete step type that can be used in an algorithm. From a flowchart perspective, a concrete step can take the form of a process, a decision, or a subprocess, for example.
      \item A concrete step may reference an \textit{AbstractOperation} object and execute its main logic in the \textit{doOperation()} method before returning the next step in the algorithm.
    \end{itemize}
  \item \textbf{AbstractOperation (Operation)}
    \begin{itemize}
      \item This class declares an interface for a single operation related to the algorithm.
      \item All the logic related to the algorithm is commonly distributed among many objects that implement this interface.
    \end{itemize}
  \item \textbf{ConcreteOperation (OperationA, OperationB, \ldots)}
    \begin{itemize}
      \item This class implements a concrete operation that encapsulates the behaviour of some logical component of the algorithm in the \textit{doOperation()} method.
    \end{itemize}
  \item \textbf{SharedDataInterface (DataInterface)}
    \begin{itemize}
      \item This class declares a data interface component that can be used by the operators to handle heuristic-related data.
      \item Setters and getters are provided by this interface to access the data stored in the interface’s implementation. Such data often must be shared among multiple operators.
    \end{itemize}
\end{itemize}
The implementation of this interface is irrelevant to the operators because the operators handle the data only through the provided methods.

- **Algorithm**
  - This class implements the main algorithm front-end class through which the algorithm can be started.
  - Through the `execute()` method, this class sequentially executes the different steps defined in the structure.

**C. Consequences**

- The pattern enables a decoupling of the execution flow structure, the code logic of the implemented algorithm and the related data. Different logical components can be implemented within `ConcreteOperation` objects, and they can be deployed in many manners within an algorithm. Such components may reference different `SharedDataInterface` implementation instances according to their data requirements. The linking between the `ConcreteStep` objects is what defines the operator ordering, relationships and their functional meaning.

- The pattern allows dynamic algorithm construction and assembly. Because the execution flow is defined by object references, the flow can be configured and modified at runtime. Therefore, an algorithm structure can be manipulated in many ways without the need for code modification.

- Use of the pattern can make the algorithm design more complex and slower, particularly at earlier stages of the operator component construction. Without proper tools or automated methods, defining a complex algorithmic structure can be time-consuming.

**D. Related patterns**

The following well-known patterns [25] are related to the Flowchart pattern proposal:

- **Interpreter**: The relationships between the `ConcreteStep` implementations and their abstraction `AbstractStep` can be considered an adaptation of the Interpreter pattern, for a non-grammar context. In the flowchart case, the `AbstractStep` instances are designed within their implementations to be returned as the next step in the chain.

- **Strategy**: The `ConcreteStep` implementations commonly use the strategy pattern to compose operations within them. However, different types can be used as other steps.

**IV. A CASE STUDY: HYPERHEURISTICS IN THE hMod FRAMEWORK**

Based on the flowchart pattern proposed in this paper, a prototype development of a new Java-based heuristic library, the `hMod` framework, was started. This framework focuses on the design and construction of heuristics from highly specialised, smaller components that can be highly reusable among multiple implementations instances and types. From a broad perspective, `hMod` allows one to generate an objectual representation of a particular heuristic flowchart by assembling different algorithmic components through an automated building process. Such a diagram can be partitioned to generate different subprocesses than can reduce the heuristic design and implementation complexity. Therefore, one can focus on the modeling of a particular subprocess of a more complex, bigger heuristic and later, the subprocess can be easily integrated into the main construct. The building process in `hMod` is supported by dependency injection techniques [22]. The framework has been used in the design and implementation of hyperheuristics, which have been generally described in Section I. They are...
cross-domain solving techniques that are alternative to meta-heuristics and that provide a modular approach for reusing heuristic methods among multiple problem domains. A common hyperheuristic architecture is presented in Figure 5. It is based on a high-level solver, the hyperheuristic itself, which operates without knowing the addressed problem and only manages abstract data such as fitness or execution time values. There is a domain barrier layer that prevents the passing of domain-related data to the highest-level. Additionally, the hyperheuristic knows that there exists a set of specialised and simpler low-level heuristics that operate at the problem domain by using extensive domain-knowledge. These are managed as black-boxes, i.e., the hyperheuristic is only focused in the low-level heuristic’s execution and results. According to the information gathered through the search, the high-level solver can learn about the effects of combining low-level heuristics and perform different execution strategies, which can vary at different stages, with them. Because different sets of low-level heuristics (i.e., different problem domain heuristics) can be used by the hyperheuristic, a modular approach is required to effectively “plug” different problem domains into the high-level solver.

hMod allows for the modularisation of heuristics through XML definition files. A set of files configure different layers of the solving architecture. There are files that implement the core of the hyperheuristic solver, and its main skeleton is defined in the file selHyp-main.xml. On it, two subprocedures that correspond to the main hyperheuristic operators are specified: the heuristic selection and the move acceptance. Because different versions of those operators can be implemented, each version is defined in a separate file and each one specifies a common entry point through which the main file calls them. Additionally, there exists a domain barrier component that functions as a communication channel with the problem domain. This layer is defined in the selHyp-db.xml file (which is called by the main file) and provides a common entry point to call a particular low-level heuristic to be selected. To adequately connect the problem domain with the domain barrier, an adapter layer must be defined; in this layer, a configuration file is implemented for each possible domain supported by the hyperheuristic. In such files, information regarding the problem domain, such as the entry point of the low-level heuristics and the solution representation used at the problem domain, is specified.

Each XML file provides a particular flowchart representation, which is defined by special tag semantics defined by the framework. For example, the selHyp-main.xml file models the flowchart illustrated in Figure 6. Several components are defined in it:

- Each box shape defines a particular step-type used. There are common procedures (the empty boxes), subprocedures (the boxes with vertical lines on the sides) and decision steps (the diamond boxes). As mentioned in Section III, each step is connected with other steps to define the algorithm structure.
- Within each box, a particular operation is specified, and each operation implements a specific behaviour element of the main skeleton. Such operations are commonly defined by classes that follow the framework standards. For example, the ConfigInitializer class implements an operation for preparing an initialiser low-level heuristic to be executed.
- Some operations are related to the data interfaces defined in the rightmost elements of Figure 6. Such relationships are described by the numbers on each step. The data interfaces allow for communication between different operations by managing proper references to the interfaces. For example, in the TimeElapsedStart operation, a TimeElapsedHandler reference is used to set the start time of the hyperheuristic execution, whereas in the TimeElapsedCheck operation, the current execution time is checked against the start time.

An example of the XML code is presented in Listing 1. This code corresponds to the execution flow defined between the AcceptState and the IterationMaxCheck operators. Different steps are defined in this code. For example, a directStep tag represents a common procedure step that encloses an operator tag to define the related operator class, whereas the subStep and the booleanStep tags define subprocedures and decision steps, respectively. Names such as selHyp/acceptState are aliased class names, which are commonly defined in context files, such as the selHyp-context.xml file. Because each step has an id attribute to reference it, the linking between steps is configured by the nextRef attribute for the directStep and subStep cases and by nextTrueRef and nextFalseRef for the booleanStep case. The data interfaces are configured within the operators through the diSetter tags, in which the dest attribute points to a particular object id configured in some XML definition (not shown in the example) that implements the related interface. Additionally, the step that starts a subprocedure is configured through the subRef attribute for the subStep case. In the example, the selHyp/returnSelectorRef id is used to point to the first step of a complete heuristic selection operator subprocess.

```
<directStep id="selHyp_main_05" nextRef="selHyp_main_06">
  <operator def="selHyp/acceptState">
    <diSetter def="selHyp/shSolutionSetter" dest="selHyp_solutionData"/>
    <diSetter def="selHyp/stepSetter" dest="selHyp_mainData"/>
  </operator>
</directStep>
```
Fig. 6. The flowchart representation of the main hyperheuristic skeleton.

```xml
<directStep>
  <subStep id="selHyp-main_06" subRef="selHyp-heuristicSelectionRef" nextRef="selHyp-main_07" />
  <booleanStep id="selHyp-main_07" nextFalseRef="selHyp-main_08" />
  <operator def="hMod/iterationMaxCheck">false
    <setter def="hMod/iterationSetter" dest="selHyp-mainData" />
  </operator>
  <booleanStep>
    <BooleanStep
    <operator id="MoveAcceptanceCall">true
      <Call id="hMod/MoveAcceptanceCall" />
    </operator>
    ...
  </booleanStep>
</directStep>
```

Listing 1. An hMod XML definition example.

To load and execute this algorithm, we can instantiate the default container (a component that handle the XML parsing) within an executable class, by using the AlgorithmContainerFactory class provided by the framework. Listing 2 shows an example. If a different heuristic selection or move acceptance operator is needed, then the related XML file must be changed in the code. The same statement is valid for the problem domain solved and the XML configuration used.

```java
package tests.demo;

import hMod.container.AlgorithmContainer;
import hMod.container.AlgorithmContainerFactory;
import hMod.core.Algorithm;
import hMod.exception.AlgorithmBuildException;
import hMod.exception.AlgorithmException;
import hMod.exception.UnsolvedReferencesException;

public class Main {
    public static void main(String[] args)
        throws AlgorithmBuildException, AlgorithmException {
        AlgorithmContainerFactory factory = AlgorithmContainerFactory.newInstance();
        AlgorithmContainer container = factory.newContainer();
        // Framework core
        container.loadFileDefinition("hMod-context.xml");
        // HH core
        container.loadFileDefinition("selHyp-context.xml");
        container.loadFileDefinition("selHyp-data.xml");
        container.loadFileDefinition("selHyp-db.xml");
        container.loadFileDefinition("selHyp-main.xml");
        // Heuristic selection load
        container.loadFileDefinition("selHyp-hs-sr.xml");
        // Move acceptance load
        container.loadFileDefinition("selHyp-ma-am.xml");
        // Some problem config load
        container.loadFileDefinition("problem1-config.xml");
        // Execution
        Algorithm a = container.getAlgorithm();
        a.execute();
    }
}
```

Listing 2. An example of how to build and run the hyperheuristic implementation in hMod.

V. CONCLUSIONS

In this paper, we have proposed the flowchart pattern, which allows one to build an objectual representation of an algorithm flowchart, even at runtime, for highly dynamic heuristic environments. The pattern is based on diverse observations regarding design practices for heuristic implementation at different abstraction levels. The pattern has been initially evaluated within a new library proposal, the hMod framework, which offers specialised semantics through different tools and technologies for easing the algorithm building capabilities offered by the pattern. High-level heuristic methods such as hyperheuristics have been successfully implemented using the framework. These applications has provided us introductory approaches of benefits (and disadvantages) of the pattern in the software design context of heuristic methods. Further work can be centered in completion and improvements for the library from its actual prototype state, in which the observations...
presented in [15] will give us interesting insights to consider for this purposes.

Moreover the technical proposal regarding hMod, the main focus in this work is about conceptual contribution. As mentioned in Section II, most of current research and tools are defined at a higher-level, commonly coupled to specific concepts and scenarios. In this work, through a pattern proposal, we want to define a useful basis for heuristic design and implementation at the lower-level. This abstraction allows its applicability to a wide range of heuristic models. Consequently, the application of the pattern in different applications and libraries related to heuristic implementation should offer the same approach, allowing to construct features that can be useful for heuristics of diverse complexity, ranging from simple and highly specialized heuristics to complex hyperheuristic architectures.

As we mentioned before, building automation tools are relevant for the pattern’s success. However, design tools that can help one understand the pattern semantics can also be important for the pattern’s usage in concrete scenarios. For example, IDE tools that can help one to build the workflows described by the pattern’s objectual relationships can represent an interesting design approach that not only can ease the heuristic designer’s tasks but also can offer them a completely new methodology or paradigm through which such tasks could be conceived. Such ideas will help to extend the conceptual background developed in the context of other frameworks, such as HeuristicLab [21], to many other platforms in a more architecturally oriented manner and always based on the benefits provided by software design principles.

We foresee many opportunities for future work regarding the pattern concept and its consequences in practical contexts. We will continue exploring how different tools could be integrated to provide useful heuristic development environments. This approach may open new alternatives in which applied optimisation and software design research meet.

REFERENCES


