Extending the Strategy Pattern for parameterized Algorithms

Ognjen Sobajic, Mahmood Moussavi, Behrouz Far
University of Calgary, Calgary, AB, Canada

Abstract

The strategy pattern decouples algorithms from the class that uses them allowing the algorithms to vary independently. It does not, however, allow algorithms to have different parameters. The solution presented in this paper addresses the case when the algorithms have different sets of parameters, and when the user is allowed to see and modify these parameters for each concrete algorithm before its execution. This is accomplished by introducing special parameter classes which encapsulate algorithm parameters and have certain responsibilities (e.g. boundary values checking). The abstract algorithm class is completely decoupled from parameters letting each concrete algorithm class create its own list of parameter instances which mirrors its parameters.

1. Introduction

The strategy pattern [1] is one of the software design patterns. It is useful when there is a family of algorithms which are used interchangeably and which solve the same problem differently. The consumer (i.e. the Client on Figure 1) of these algorithms is completely decoupled of any particular implementation. It only has to maintain a reference to the abstract algorithm which defines common interface for all the implemented algorithms. The abstract algorithm defines the *execute()* method which when called executes the routine for a particular algorithm attached at the time. Various algorithms are implemented differently, but strategy pattern allows them to be used interchangeable.

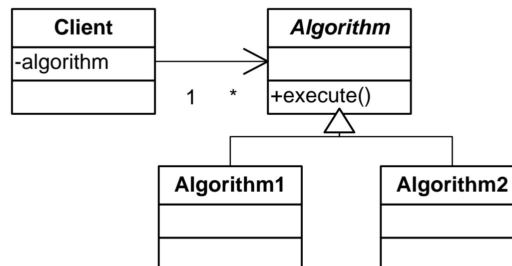

Figure 1 - Strategy Pattern

This paper addresses a scenario where the strategy pattern cannot be directly used, but is to be adapted. The original motivation for this problem comes from time series generation algorithms. These algorithms produce artificial time series which are longer than historic time series given at the input to the algorithms. The recognized difficulty in these algorithms is to produce artificial time series which preserve all relevant statistics comparing to historic data. There are various time series generation algorithms; however, none of them is universally accepted. Each algorithm can be used in a specific scenario for a specific branch of engineering. This has motivated the need for an adaptable software architecture which would be able to host various time series generation algorithms. One of the problems within this architecture is that each of the algorithms may have some tuning mechanisms in the form of parameters that have to be submitted to the algorithms before its execution. These parameters are by no means the same set of parameters for each of the algorithms. One of the time series generation algorithms with its adaptable software architecture which can host multiple time series generation algorithms and which utilizes solution presented in the paper is presented in [2]. The applicability of this extension of the strategy pattern is by no means limited to time series generation algorithms. Wherever there is a choice between various algorithms which use different set of parameters, this solution should be applicable.

In this paper section 2 briefly discusses the limitations of the strategy pattern and section 3 presents a solution to these limitations. Section 4 reviews the related patterns. Section 5 provides an example of implementation, and section 6 discusses the possible variations and extensions of the proposed solution. This is followed by section 7 which draws the final conclusions.

2. Shortcomings of Strategy pattern

The strategy pattern can be used to host different algorithms which either have no parameters or the set of parameters for each algorithm is the same. The problem arises if various algorithms with different sets of parameters are to be used. Obviously, these parameters cannot be declared in the *execute()* method in the abstract class *Algorithm*, as they vary from algorithm to algorithm. Existence of various sets of parameters results in different interfaces which would not be acceptable for the strategy pattern. Hence, the strategy pattern needs to be adapted in order to handle the problem.

The traditional solution is to develop a context interface that would have the union of all parameters needed by the different algorithms. However, this solution breaks the independence between client and the algorithm. It means that adding a new algorithm, whose parameters are not in the union of parameters, requires some changes to be made on the client.

In the following section we present a solution which does not break the independence between client and algorithms; is able to host algorithms with different parameters; and provide the user with parameters specific to a chosen algorithm.

3. Solution

Based on the strategy pattern this solution is extended with a mechanism which allows each concrete algorithm class to define its own set of parameters. Once they are defined, they need to be presented to the user, so the user can change their values. Additionally, the parameters might exhibit some constraints (e.g. boundary values as discussed in section 6) not allowing the user to set the values which would violate these constraints. These requirements set some responsibilities over parameters which is why the parameters are encapsulated in special classes. All parameter classes inherit from an abstract class *Parameter* as in Figure 2. The classes inheriting the class Parameter are concrete classes, each of which represents concrete type of parameter (e.g. integer, double precision number, Boolean, etc). Each of the concrete algorithm classes may contain one or more instances of the *Parameter* class which mirror the actual parameters of the algorithm. The method *getParameters()* returns the list of the algorithm parameters as a list of parameter classes. This method is abstract and each algorithm defines it by returning its parameters.

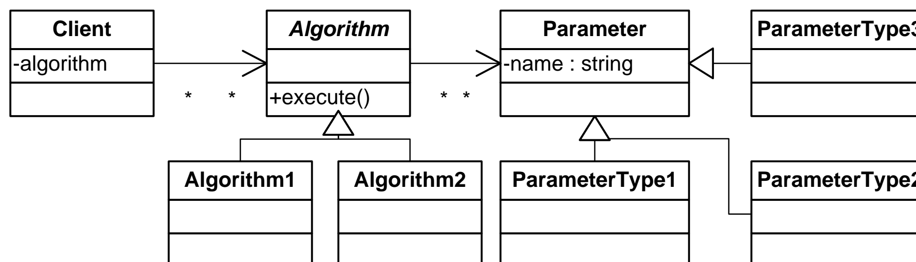

Figure 2 - The strategy pattern (Client, Algorithm, Algorithm 1 and Algorithm 2) on the left-hand side, and its extension (Parameter class with concrete Parameter classes) on the right-hand side

In the following sub-sections, each of the roles (Client, Algorithm and Parameter) is discussed in greater detail, as well as the typical scenario of usage.

3.1 Role of Client

In the strategy pattern the role of *Client* is to maintain a reference to an instance of a concrete algorithm class and to trigger the execution when it is required [1]. When it comes to this scenario, the *Client* has more responsibility. The *Client* needs to be aware of all parameter types (i.e. all the classes inherited from *Parameter* class) and it can present them to the user (i.e. with appropriate controls on the user interface). For example text box will be created for each string parameter, while sliders will represent integer parameters and check box will be displayed for Boolean parameters etc. Once the algorithm parameters are presented to the user, he is able to see the parameter values on appropriate controls and to change them. As the user tweaks the controls assigned to the parameters, the client calls appropriate parameter classes setting new values. In other words the client's responsibility is to match the changes on controls made by user to appropriate parameter classes. Basically, the client pulls the parameters from the algorithm, presents them to the user, and after having them possibly modified by the user, sends back their values to parameter classes.

While the client does not depend on concrete *Algorithm* classes, it does depend on all concrete *Parameter* classes. This is because the client has to recognize each of the parameter classes and present them to the user accordingly. Consequently, from one hand new algorithm classes can be easily added, as nothing depends on them. From another hand, if a new parameter class is added, the client has to be updated accordingly in order to be able to recognize and present the parameter type to the user. However, a stable set of Parameter classes can be achieved as soon as all the primitive types are implemented in appropriate classes. In other words, after having implemented a couple of parameter classes which encapsulate basic data types such as integer, double, string, Boolean and char, the need for new parameter classes is very unlikely to occur (i.e. a new algorithm which would require a new kind of parameter).

Furthermore, the client need not necessarily be the user interface. The Client can be just an intermediate level between the user interface and the algorithm. If it is the case, then the Client passes the parameters further to the user interface. In this case, the Client does not depend on concrete Parameter classes, but the user interface does. Basically, everything being said in the sub-section 3.1, then, holds for user interface, and the client becomes a middle man between the algorithm and the user interface.

However, in either case the user interface (being the client or not) is flexible in the sense that it updates depending on the list it gets from the concrete algorithm class. It creates user controls, for each of the parameters being received.

3.2 Role of concrete algorithm

A concrete algorithm class contains concrete algorithm routine. The routine may have various parameters in its declaration and they can vary depending on concrete algorithm class. These parameters in the declaration should be mirrored by the list of appropriate *Parameter* class instances. For example, consider the following algorithm declaration for a possible genetic algorithm:

```
void GeneticAlgorithm(int populationsize, int maxiterations, double mutationprob);
```

This should be accompanied with the following definition of parameter class instances:

```
parameters = new Parameter[3];
parameters[0] = new IntParameter("Population size");
parameters[1] = new IntParameter("Max iterations");
parameters[2] = new DoubleParameter("Mutation Probability");
```

In this case classes `DoubleParameter` and `IntParameter` are inherited from parameter class and they encapsulate a double precision number and an integer respectively.

As the algorithm routine is not accessed directly by the Client, the routine should be private. The Client invokes *execute()* method instead. In this sense, the concrete algorithm class *execute()* hides the algorithm interface and acts as adapter (i.e. wrapper). Hence, each of the concrete algorithm classes can be regarded as an application of adapter pattern [1] between a concrete algorithm interface and what the client expects (i.e. *execute()* method itself).

3.3 Parameter classes

Parameter class is an abstract class which is inherited by concrete parameter classes. Each of concrete parameter classes should represent a concrete parameter type applicable for the algorithms being used. Some of the examples are integer, floating-point number, Boolean value, string type etc. The exact set of parameter classes highly depends on concrete application, as some applications may need very specific parameter types.

The abstract parameter class need not consist anything more than just a parameter name which is to be presented to the user in addition to the actual parameter value. As for concrete parameter classes it is desirable to have default values which are assigned to the parameter value on initialization (i.e. in the constructor). Also, for numeric values, there should be minimum and maximum values. It is the parameter class responsibility to reject values out of its boundary values.

In some more complex scenarios, one of which is discussed in 6.2, the parameter class might need observer pattern implemented for notifying other classes when the value changes.

3.4 Typical scenario

The typical scenario is more complex than the one with the strategy pattern. Also the user is involved in this scenario. The sequence diagram is presented in Figure 3.

The *happy path* goes as the following:

- Once a concrete algorithm is instantiated, it creates parameter objects to match its real parameters (3 and 4). In this example the algorithm has only two parameters.
- The client requests the list of parameters for the algorithm (5)
- The concrete algorithm sends its list of parameters to the client (6)
- The client presents the parameters on the user interface (7). For example:
    - The numeric parameters are presented by sliders, which the user can move and change their values
    - the Boolean parameters are presented as checkboxes and the user can check or uncheck them
    - the string parameters are presented as textbox where the user can change their string values
- As the user is interacting with user interface controls, the client is in charge to notify parameter classes of their new values accordingly (8,9) and (10,11)
- Once the algorithm is to be executed (12), the parameter instances have their updated values
- On the execution, the concrete algorithm class pulls the updated parameter values from the parameter classes first (14, 15) and finally invokes the algorithm routine itself passing it the parameter values (16)

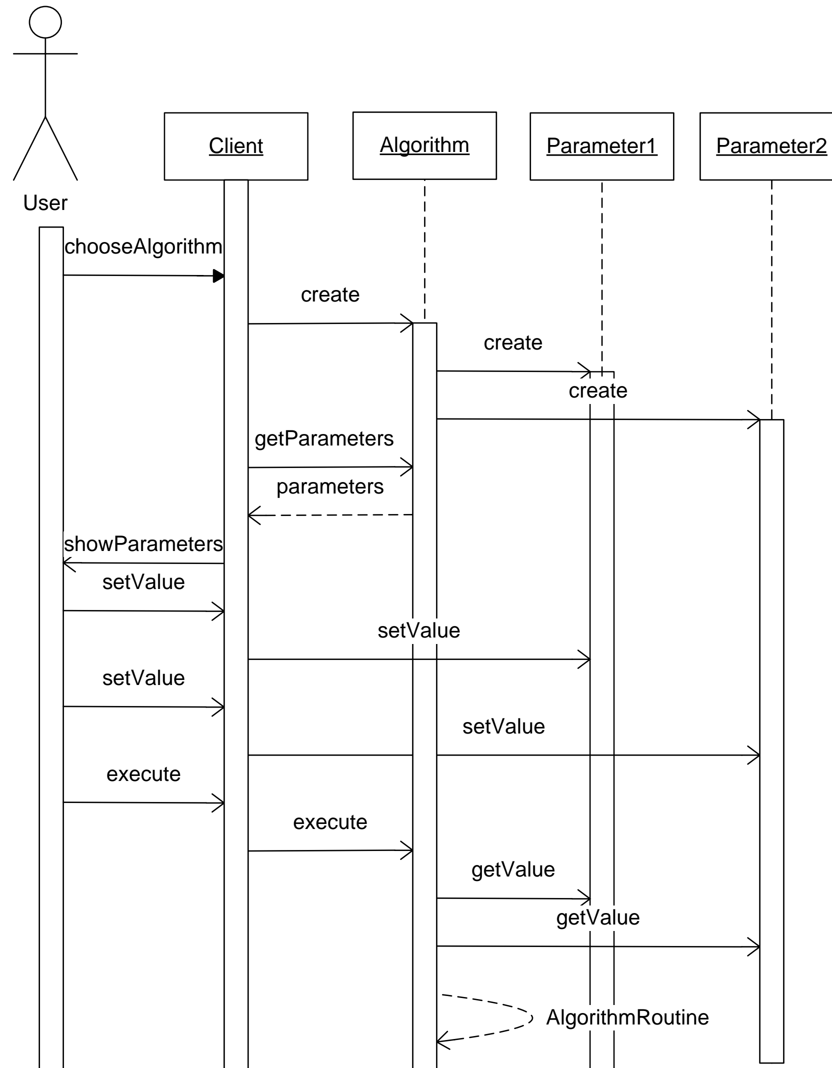

Figure 3 - Sequence Diagram for happy-path of Usage

4. Related patterns

In this section, there are two related patterns.

4.1 Strategy pattern

As discussed earlier, this solution overcomes some of the shortcomings of the strategy pattern. The crucial difference between the strategy pattern and the solution presented is in using the parameter classes.

4.2 Adapter patter

Adapter pattern transforms one interface to another [1]. It is used when the client expects different interface from what it is offered by the class whose service is used. In this solution, each of the concrete algorithm classes acts as a single adapter pattern. A concrete algorithm class transforms the abstract algorithm class interface into specific interface for a concrete algorithm routine. It is done by using the parameter classes as encapsulation of parameters

required for the concrete algorithm. Thus, we could say that each of the concrete algorithm classes is an application of the adapter pattern.

4.3 Observer pattern

The observer pattern is directly incorporated in the case discussed in 6.2. If there are classes interested in parameter values (e.g. Expression and Constraint classes as in 6.3), the parameter class needs an observer mechanism for its values. The observers (e.g. Expression classes) sign to a parameter class to be informed on value change. The trigger mechanism is implemented in the setter of each parameter class separately.

5. Example of Implementation

In this example, there are two algorithms which solve the same algorithm problem. The nature of the very algorithm problem is not quite important as there is no algorithm code presented here. However, we can assume that we are solving, for example, Travelling Salesman Problem (a problem to find a shortest route which would span over all of the nodes in the graph).

One of the algorithms is based on Genetic Algorithms and another one is a binary search. The abstract class *Algorithm*, and the *AlgorithmA* (the class with the Genetic Algorithm) and *AlgorithmB* (the class with binary search algorithm) classes are listed below.

```
abstract class Algorithm
{
    public Algorithm()
    { }

    protected Parameter[] parameters;

    public Parameter[] getParameters()
    { return parameters.copy(); }

    public abstract void execute();
}

class GeneticAlgorithm : Algorithm
{
    public AlgorithmA()
    {
        parameters = new Parameter[3];
        parameters[0] = new IntParameter("Popul size", 50, 1000, 100);
        parameters[1] = new IntParameter("Max iterations", 2, 10000, 20);
        parameters[2] = new DoubleParameter("Mutation Probability", 0.0, 0.9, 0.05);
    }

    int populationsize;
    int maxiterations;
    double mutationprob;

    public override void execute()
    {
        populationsize  = ((IntParameter)parameters[0]).GetValue();
        maxiterations   = ((IntParameter)parameters[1]).GetValue();
        mutationprob = ((DoubleParameter)parameters[2]).GetValue();
        GeneticAlgorithm(populationsize, maxiterations, mutationprob);
    }
```

```
    private void GeneticAlgorithm(int populationsize, int maxiterations, double 
mutationprob)
    {
        //Genetic Algorithm itself
    }
}

class BinSearchAlgorithm : Algorithm
{
    public AlgorithmB()
    {
        parameters = new Parameter[2];
        parameters[0] = new BoolParameter("Fast Search", true);
        parameters[1] = new IntParameter("Depth", 1, 50, 10);
    }

    bool fastsearch;
    int depth;

    public override void execute()
    {
        fastsearch  = ((BoolParameter)parameters[0]).GetValue();
        depth       = ((IntParameter) parameters[1]).GetValue();
        BinarySearch(fastsearch, depth);
    }

    private void BinarySearch(bool fastsearch, int depth)
    {
        //Binary Search itself
    }
}
```

Note the following:

- The list of parameters is declared in the abstract Algorithm class, but the actual list of parameters is defined in the constructors of the concrete classes *GeneticAlgorithm* and *BinSearchAlgorithm*.
- The *getParameters()* method in the *Algorithm* class does not return the list of parameters, but its shallow copy. Hence it prevents the client from modifying the list, but still enabling it to access instances of the parameter class.
- The method *execute()* is abstract in the algorithm class (as it is in strategy pattern [1])
- The method *execute()* is implemented in the concrete classes. It first pulls out the parameters values from the parameter classes and stores them in the corresponding primitive types (e.g. integers and float-precision numbers). At the end, it invokes the algorithm routine, passing it the values taken from the parameter classes.
- In *execute()* method there are some castings required for extracting the parameters' values which might be somewhat expensive. In programming languages such as C/C++ which support pointer types and whose compiled code is not managed, rather directly executed, the castings can be avoided. Pointers can be used for each of the parameters' values by assigning each pointer to a certain parameter value. The pointers are assigned on parameter classes' initialization. Once the algorithm is to be executed, it is the pointers that are consulted and the values are retrieved for the execution.
- In order to implement a workaround with pointers (as discussed in the previous point), the programming language must not be managed in runtime. If the code is not directly executed, but managed, the virtual machine might move the objects without updating the pointers referring to them. If it happens to a parameter class object, the pointer to its value becomes invalid. An example of managed programming language with pointer support is C#.

The following is an implementation of abstract Parameter class and three concrete parameter classes inheriting it, i.e. *BoolParameter*, *IntParameter* and *DoubleParameter*.

```
abstract class Parameter
{
    private string name;

    public string GetName()
    { return name; }

    public Parameter(string name)
    { this.name = name; }
}

class BoolParameter : Parameter
{
    private bool Value;

    public bool GetValue()
    {   return Value;   }

    public void SetValue(bool value)
    {   Value = value;   }

    public BoolParameter(string name, bool defaultvalue)
        : base(name)
    {
        Value = defaultvalue;
    }
}

class IntParameter : Parameter
{
    private int min;

    private int max;

    private int Value;

    public int GetValue()
    { return Value; }


    public void SetValue(int value)
    {
        if (value < min)
            throw new ArgumentOutOfRange(GetName() + " can't be less than " + min);
        if (value > max)
            throw new ArgumentOutOfRange(GetName() + " can't be greater than " + max);
        Value = value;
    }

    public IntParameter(string name, int min, int max, int defaultvalue) : base(name)
    {
        this.min = min;
        this.max = max;
        Value = defaultvalue;
    }
}
```

```
class DoubleParameter : Parameter
{
    private double min;

    private double max;

    private double Value;

    public double GetValue()
    { return Value; }

    public void SetValue(double value)
    {
        if (value < min)
            throw new ArgumentOutOfRange(GetName() + " can't be less than " + min);
        if (value > max)
            throw new ArgumentOutOfRange(GetName() + " can't be greater than " + max);
        Value = value;
    }

    public DoubleParameter(string name, double min, double max, double defaultvalue)
        : base(name)
    {
        this.min = min;
        this.max = max;
        Value = defaultvalue;
    }
}
```

Note the following:

- The parameter class is abstract and it only encapsulates the name which is common for all parameter classes
- The BoolParameter simply inherits the abstract Parameter class and encapsulates a Boolean value with getter and setter
- *IntParameter* and *DoubleParameter* classes also encapsulate appropriate types (i.e. integer and double-precision values correspondingly), but they also cast boundary constraints in the setter, not allowing values outside the minimum/maximum boundaries. Anyways, this can be implemented differently as discussed in 6.1.

6. Possible variations and extensions

The solution presented can have some variations and extensions in various ways. However, they are not mutually exclusive and can be applied in conjunction. Three of them are presented in the following sub-sections.

6.1 Boundary Values

In addition to a specific type of values, Parameter classes may exhibit some other constraints regarding their values. They can have boundary values (i.e. minimum and maximum) as listed in the implementation of *IntParameter* and *DoubleParameter* in section 6. These restrictions on parameter values can be easily incorporated in the parameter classes. If the parameter class is to accept a new value which is outside its boundaries (e.g. in the setter), it needs to reject the value and notify the client about it (e.g. throwing an exception). Then, the client notifies the user that the new value has not been accepted due to boundary constraints.

However, there might be another solution to boundary values without throwing exceptions. The parameter class might set the value equal to the boundary value which is closest to the value it received without throwing any

exception. For example, if the minimum and maximum are 0 and 100 respectively, and the received value (in the setter) is 150, the parameter would set 100 as its maximum allowed value.

However, the choice between these two solutions affects the client (i.e. user interface) implementation, as the client needs to provide the user with updated parameters values. In the first one the client expects an exception, and if it catches one it restores the old value. In the second one, it has to check if the new value has been accepted (e.g. by the getter method) and show it to the user.

6.2 Observing values

The parameter class might have implemented observer mechanism for observing their values changes. Not only is the user interface signed to observe the value change, but also constraint classes, which are discussed in the following subsections. They need to be notified when the values are changed as the constraint class has to revaluate its condition and reject new value if the condition is not met.

6.3 Constraints over parameters

Sometimes constraints over a parameter value cannot be expressed simply as boundary values (i.e. minimum and maximum). Constraints, in general, may involve more parameters in one logical condition which is to be satisfied. For example let's assume there is an algorithm which must process the data within a given time frame. The time frame is given as the start and the end of the time interval and these two pieces of data are encapsulated as two parameters. Obviously these two parameters will have boundaries, but at the top of it, the begin time needs to be less than the end time:

Or equivalently as:

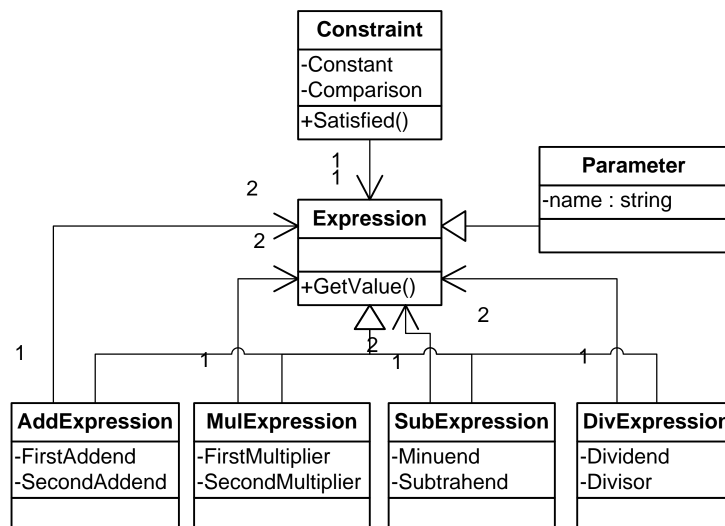

Figure 4 - Expression class hierarchy with the constraint class

Nevertheless, there might be expressions that are way more complex than this one involving all four arithmetic operations and involving more parameters. In this case we need a hierarchy of expression classes. Each of them will

represent one of the arithmetic operations, on the top of abstract Expression class which will define the interface to use. The leaf class will be the parameter class itself. In this way an arbitrary complex expression may be constructed by composing expression classes and having parameter classes as leafs of this composition. The idea is based on interpreter design pattern [1], or similarly on specification class hierarchy [3], with the difference of that the number value is being calculated instead of Boolean one, and correspondingly, having arithmetic classes instead of Boolean one, and correspondingly, having arithmetic classes instead of Boolean operations as in Figure 4.

On the top of the Expression class hierarchy, there is a class encapsulating constraints themselves (Figure 4). It simply consists of an object of the Expression class, a constant value and comparison sign. The constraint is satisfied when value of the expression and constant are in accordance with the comparison sign which can be *equal*, *less*, *less or equal*, *greater* or *greater or equal*. Changing any of the parameter values involved in the constraint requires revaluation of the expression value and thus revaluation of the constraint. This is why an observer pattern might be appropriate inside the parameter classes. All the expressions and constraints involving a parameter need to observe the parameter value as they need to revaluate their states after parameter value changes.

At the end, a concrete algorithm class might have arbitrary number of constraints objects which represent real constraints over their parameter values. These constraints are defined (i.e. initialized) as soon as parameter classes are initialized. If changing a parameter value results in violating one of the constraints, the parameter class needs to restore previous value which did not violate this constraint. In this way, user is not allowed to set a combination of parameters' values which are not satisfactory for the algorithm execution.

7. Conclusion

This paper presents an easy way to extend the strategy pattern for algorithms with incompatible interfaces. Since the approach presented in the paper is built on the strategy pattern, it is important to emphasize the main difference between the two. While the strategy pattern can host only algorithms with the same set of parameters (or usually without any parameters), this extension goes one step forward hosting algorithms with different set of parameters, and enabling user to see and modify these parameters before the algorithm execution. Also, each concrete algorithm can be considered as an application of adapter pattern as it transforms required interface to the algorithm specific interface using specific parameter classes.

Acknowledgments

We would like to thank Pedro Miguel Ferreira Costa Monteiro for a great deal of help provided during the shepherding process. He had a lot of useful comments and suggestions, and this article has undergone significant changes since Pedro's first review.

8. References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley, 1994.

2. *A Stochastic Time Series Generator with Adaptive Software Architecture.* Sobajic, Ognjen, et al. 2010. (accepted in the conference proceedings, but not published yet).

3. Eric Evans, Martin Fowler. Specifications. *Martin Fowler.* [Online] [Cited: 7 27, 2010.] http://www.martinfowler.com/apsupp/spec.pdf.