Parameterized Strategy Pattern

OGNJEN SOBAJIC, MAHMOOD MOUSSAVI AND BEHROUZ FAR, University of Calgary

The Strategy pattern decouples algorithms from the class that uses them allowing the algorithms to vary independently. It does not, however, allow the algorithms to have different parameters. The parameterized strategy pattern 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 algorithms parameters. The abstract algorithm class is completely decoupled from parameters letting each concrete algorithm class to create its own list of parameter instances which mirrors its parameters.

General Terms: Design Patterns

Additional Key Words and Phrases: Parameters, Strategy Pattern, Algorithms

ACM Reference Format:

Pineo, D. and Ware, C. 2010. Neural Modeling of Flow Rendering Effectiveness. ACM Trans. Appl. Percept. 2, 3, Article 1 (May 2010), 10 pages.

1. INTRODUCTION

The strategy pattern is one of the OO design patterns (Gamma, Helm, Johnson, & Vlissides, 1994). It is useful when there is a family of algorithms which solve the same problem differently and which are used interchangeably. 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 instantiated attached at the time. Various algorithms are implemented differently, but the strategy pattern allows them to be used interchangeably.

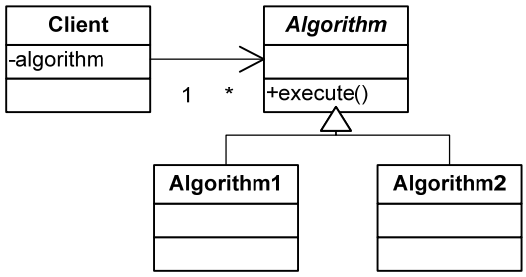

Fig. 1. Strategy Pattern

This paper addresses a scenario where the strategy pattern cannot be applied directly, but is to be adapted. The original motivation for this problem comes from a software architecture which is able to host different algorithms of a kind. 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 of the 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 applies the solution presented in the paper is presented in (Sobajic, Ilich, Moussavi, & Far, 2010). The applicability of this extension of the strategy pattern is not limited to time series

generation algorithms. Whenever 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 limitations of the strategy pattern and section 3 presents the solution to these limitations. Section 4 provides an example of implementation while section 5 presents dialog between the client and an algorithm as if they were humans. Section 6 discusses the possible variations and extensions of the solution. The related patterns are covered in section 7 which is followed by section 8 where the final conclusions are drawn.

2. SHORTCOMMINGS OF THE 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 the client and the algorithm. It means that adding a new algorithm, whose parameters are not in this union, would require some changes to be made on the client.

In the following section we present a solution which does not break the independence between the client and algorithms; is able to host algorithms with different parameters; and provides 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 6.1) not allowing 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, Boolean value, 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 actual list of the algorithm parameters as a list of parameter classes. This method is abstract and each algorithm defines it by returning its parameters.

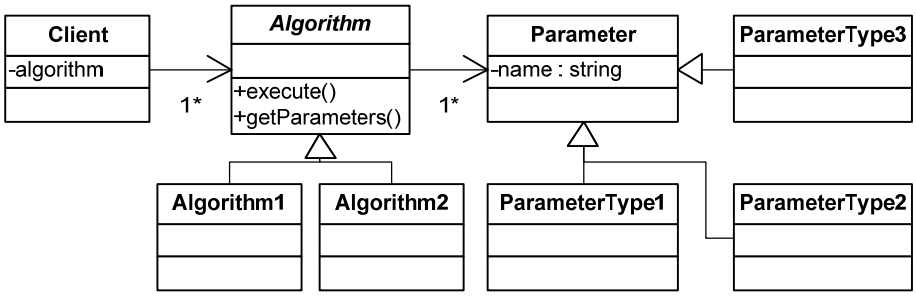

Fig. 2. The Strategy Pattern (Client, Algorithm, Algorithm 1 and Algorithm 2) on the left-hand side, and its extension (Parameter class with concrete Parameter classes) on the right-hand side

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 the Client

Role of the *Client* in the strategy pattern is to maintain a reference to an instance of a concrete algorithm class and to trigger the execution when it is required (Gamma, Helm, Johnson, & Vlissides,

1994). 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) so 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 and so on. In this way the user interface is fairly flexible, adapting itself based on the algorithm parameters. The interface is generated based on the algorithm parameters. However, the idea of automatic generating the user interface is not new, e.g. (Angel R. Puerta, 1994) and (Pawson, 2004).

Once the algorithm parameters are presented to the user, he is able to see the parameter values on appropriate controls and change them. As the user tweaks the controls assigned to the parameters, the client calls appropriate parameter objects setting new values. In other words the client's responsibility is to match the changes on controls made by the user to appropriate parameter objects. 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 several parameter classes which encapsulate basic data types such as Integer, Double, String and Boolean, 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 list of 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 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) has to be flexible in the sense that it changes 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 a concrete algorithm routine. The routine may have various parameters in its declaration and they can vary depending on the 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:

```
private void GeneticAlgorithm(int popul_size, 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 (Gamma, Helm, Johnson, & Vlissides, 1994) between a concrete algorithm interface and what the client expects (i.e. *execute()* method itself).

3.3 Parameter classes

The Parameter class is an abstract class which is inherited by concrete parameter classes. Each of the 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 mostly depends on the concrete application, as some applications may

need very specific parameter types. For example parameters of type date, time, money or temperature are not that common as integers and doubles which are basic data types and are used almost everywhere. However, when building finance algorithms, the type for money would be one of the inevitable, but algorithms handling climate data would probably need temperature parameters.

Although they may have different type of values, each parameter needs an unambiguous way to be addressed. In that manner the abstract parameter class, which is the base class of all the parameters, need not consist of anything more than just a parameter name. Parameter name serves as an identifier 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's 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 an observer pattern implemented for notifying other classes when the value changes.

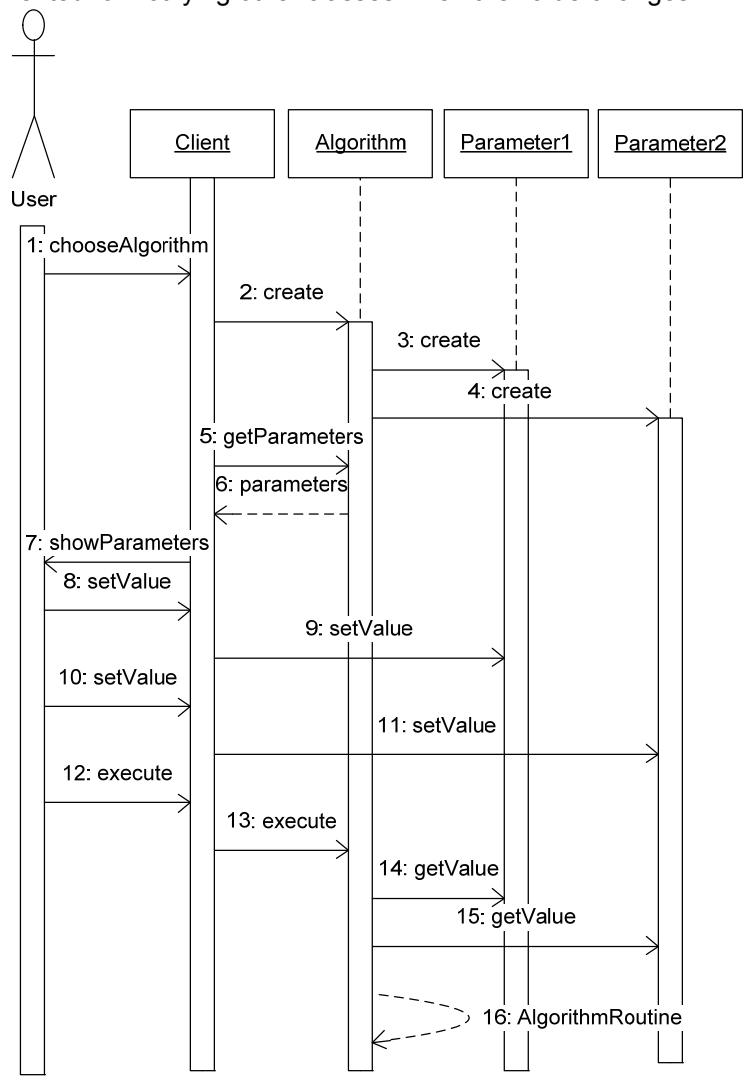

Fig. 3. The Sequence Diagram for happy-path

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 following:

– Once a concrete algorithm is instantiated (1 and 2), 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).

4. 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 Branch-And-Bound search. The abstract class *Algorithm*, and the *GeneticAlgorithm* (the class with the Genetic Algorithm) and *BranchAndBoundlgorithm* (the class with branch-and-bound 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 Geneticlgorithm()
    {
        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);
    }

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

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

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

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

    private void branch_and_bound(bool fastsearch, int depth)
    {
        //Branch-and-Bound algorithm 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 *BranchAndBoundSearchAlgorithm*.
- The *getParameters()* method in the *Algorithm* class does not return the actual 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 classes.
- The method *execute()* is abstract in the algorithm class (as it is in the strategy pattern).
- 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, but 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)
        : Parameter(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)
        : Parameter(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)
        : Parameter(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. Boundary constraints are not a mandatory component of the parameterized strategy pattern. However, to illustrate the pattern's extensibility, we have introduced the boundary constraints example in the code above. They are discussed in greater detail in 6.1.

5. DIALOG OF COLLABORATION

In order to illustrate collaboration between the client and the algorithm we provide their communication in a form of dialog. This dialog represents what they would say to each other if they were humans. Here is how the dialog between the Client and the Algorithm would look like (the text in bold refers to a dialog in the case of the strategy pattern only):

Client: I need to run you. *What should I do?*
Algorithm: You need to provide me with parameters' values. Here is the form, please fill it up.
(Algorithm is handing the Client the form to fill up. Client is filling the form up with parameters' values*)*
Client: Here are the parameters' values. ***Please do this job for me.***
(The Algorithm *first reads the form and then* ***starts working. He finishes)***
Algorithm: Here are the results.

The form the algorithm gives the client is actual list of parameters. Each algorithm may have different form to be filled out. The point is that the form is written in a way understandable for client so he can fill it up.

6. POSSIBLE VARIATIONS AND EXTENSIONS

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

6.1 Boundary Values

In addition to a specific value types, 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 4. These restrictions on parameter values can be easily incorporated in the parameter classes within the setter. If the parameter class is to accept a new value which is outside its boundaries (i.e. 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 closer 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 (i.e. in the setter) is 150, the parameter would set 100 instead, as it is maximum allowed value.

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 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. These two pieces of information are encapsulated as two parameters. These two parameters will have boundaries, but on the top of it, the begin time needs to be less than the end time:

$$begin\_time < end\_time \tag{1}$$

Or equivalently as:

$$end\_time - begin\_time > 0 \tag{2}$$

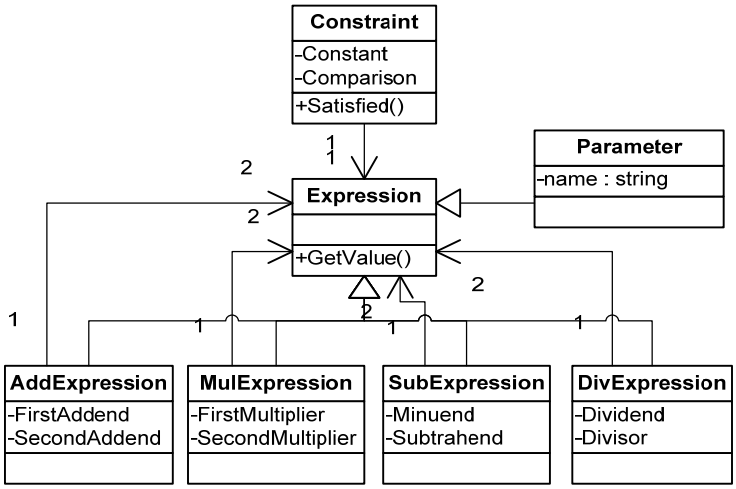

Fig. 4. Expression class hierarchy with the constraint class

Nevertheless, there might be expressions that are more complex involving other arithmetic operations and more than two parameters. In order to encapsulate all these constraints we need a hierarchy of expression classes which recursively can build on each other. Each of them will represent one of the arithmetic operations, on the top of the abstract Expression class which will define the interface to use as in Figure 4. The abstract Parameter class needs to inherit from abstract Expression class as parameter instances are leafs of any expression (e.g. Figure 5). 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 (Interpreter Design Pattern) (Gamma, Helm, Johnson, & Vlissides, 1994), or similarly on specification class hierarchy (Eric Evans), with the difference of that the number value is being calculated 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. 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. Therefore, an instance of the constraint class will have to keep track of changes of values for each Parameter involved in it. 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. Thus, the parameter class needs an observing mechanism to enable constraints and expressions to observe its values.

At the end, a concrete algorithm class might have an 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 (e.g. due to user action) 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.

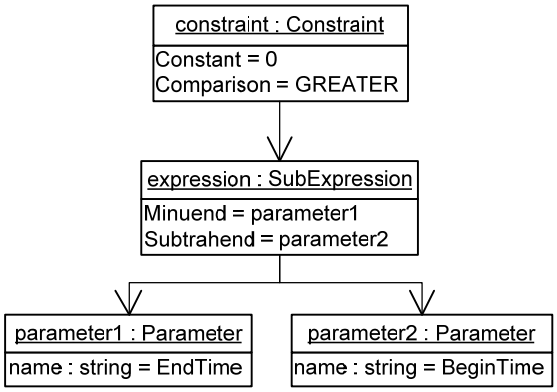

Fig. 5. The object diagram for the constraint: end_time – begin_time > 0

7. RELATED PATTERNS

In this section, we review a couple of related patterns.

7.1 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.

7.2 Adapter Pattern

The adapter pattern transforms one interface to another (Gamma, Helm, Johnson, & Vlissides, 1994). It is used when the client expects a different interface from the one being offered by the class. Although the implementation of this solution is quite different from the adapter pattern, the idea of the adapter pattern is similar. Each of the concrete algorithm classes acts as a single adapter pattern. A concrete algorithm class transforms a specific algorithm class interface (which consists of all its parameters) into the abstract algorithm interface. 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 a specific application of the adapter pattern.

7.3 Observer Pattern

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

7.4 Interpreter Pattern

The interpreter pattern is used in the extension with constraints as discussed in 6.2. While the *AddExpression*, *SubExpression*, *MulExpression* and *DivExpression* are non-terminal expressions (Interpreter Design Pattern), the parameter classes are terminal expressions as they contain concrete values.

7.5 Editor Pattern

The Editor Pattern (Beck & Johnson, 1994) as presented by Beck and Johnson exhibits very similar idea. Namely, Editor is an object which encapsulates the object interface. The client can retrieve the editor from the object and check what methods are supported. The editor is usually user interface object responsible for its interpretation on the interface. Unlike the Editor pattern which is responsible for the entire interface of the object, the solution here presents the exact signature of a method. Thus, while the Editor Pattern encapsulates the whole interface of the object and presents it on the user interface, the solution in this paper encapsulates parameters of a method (i.e. execute method in the strategy pattern) and appropriately presents it on the user interface. Additionally, the parameter classes are not coupled to the user interface as the Editor is. They are oblivious of the way they are presented and it is the responsibility of the client to present them appropriately to the user.

8. 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.

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.

REFERENCES

Angel R. Puerta, H. E. (1994). Model-Based Automated Generation of User Interfaces. *AAAI94.* Seattle.

Beck, K., & Johnson, R. (1994). Patterns Generate Architectures. *ECOOP.*

Eric Evans, M. F. (n.d.). *Specifications.* Retrieved 7 27, 2010, from Martin Fowler: http://www.martinfowler.com/apsupp/spec.pdf

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Interpreter Design Pattern. (n.d.). Retrieved 11 2010, from http://sourcemaking.com/design_patterns/interpreter

Pawson, R. (2004). *Naked objects.* Dublin.

Sobajic, O., Ilich, N., Moussavi, M., & Far, B. (2010). A Stochastic Time Series Generator with Adaptive Software Architecture.