

Carleton University  
School of Computer Science  
95.540 Software Patterns

# **Parameter-View Generator**

**(A Proto-Pattern)**

## ***Abstract***

Nowadays, most if not all software applications are equipped with a graphical user interface (GUI) that allows users to interact with the application. Typically, such GUI's provide a means for users to trigger some operation(s) of the application. Such operations typically either require or offer parameters that the user can modify. From the point of view of a software developer, it is often time-consuming and also inefficient to have to write code to create the GUI components that fetch such input from the user. The problem is exacerbated if the application contains many operations each of which may require multiple parameters. This pattern proposes a solution that provides automatic generation of user interface components for acquiring the values of these parameters.

Jon Harris (jbharris@magma.ca)  
December 16, 2002

## **Problem**

*An interactive software application can contain several executable operations. Some or all of these operations may accept parameters whose values can be set by the user. Acquiring, validating and forwarding the parameter values provided by the user to the appropriate operation can be a tedious and or cumbersome task.*

As an example, consider a Photo Editing application where we wish to provide users with a variety of operations to apply to their images. Imagine that we wish to add to the application the functionality of being able to apply an image resize operation to the current image. Applying the resize operation requires parameter values from the user such as the new image width and height and also whether or not to constrain proportions of the original image.

In addition to writing the code that performs the operation, we are now faced with several onerous tasks:

- Adding a trigger to the operation in the GUI.
- Creating GUI components for acquiring the parameter values from the user.
- Retrieving the parameter values from the GUI components.
- Validating the parameter values.
- Forwarding the parameter values to the operation.

For operations with several parameters, these tasks may need to be performed multiple times.

## **Context**

You are developing a software system within which there are several self-contained operations that you wish to be able to apply to some data.

These operations require parameter values that must be editable by the user at run-time.

## **Forces**

- **Diversity of parameter types**  
Not all parameters will have the same type. For example, parameters may be numbers, strings, booleans, or other more complex objects. Furthermore, several parameters of differing types may all be required for one operation.
- **Validation of parameter values**  
The values of a parameter for an operation may only have certain acceptable values. The complexity of the operation will increase if it must validate the values of its parameters, yet not doing so makes it difficult control their validity.

- **Dependencies between parameters**  
Certain parameters of the same operation may be dependant upon one another. For example, an integer parameter may only be valid if its value is at most half the value of another integer parameter.
- **Maintainability**  
The complexity of the code for actually performing the execution of an operation will increase if cluttered with code for acquiring the values of its parameters from user interface components, making it difficult to understand and maintain.
- **Evolution flexibility**  
The parameters required by an operation may change over time. Also, the look and feel implementation of the user interface components may be modified at a later date.

### **Solution**

The solution is to introduce four new classes to the system to represent operations and their parameters, and user interface components for each.

- A *Parameter* class defines an interface for setting the name, type and value of parameters for an operation.
- An abstract *Operation* class stores general information about an operation such as its name and a list of *Parameter* objects. Each operation requiring parameter values from the user will be made to extend this base class.
- A *ParameterView* class stores a reference to a *Parameter* object, and defines the visual representation of this *Parameter* in the user interface.
- An *OperationView* class stores a reference to an *Operation* object, and defines the visual representation of this Operation in the user interface. The *OperationView* also acts as a container for *ParameterView* objects which correspond to the *Parameter* objects in the list of its *Operation* object.

This structure can be considered as an extension of the Model View Controller pattern [Buschmann et al., 2000]. In this context, the Model can store a list of Operations that can be performed on its data, and the View can store a collection of the corresponding OperationViews.

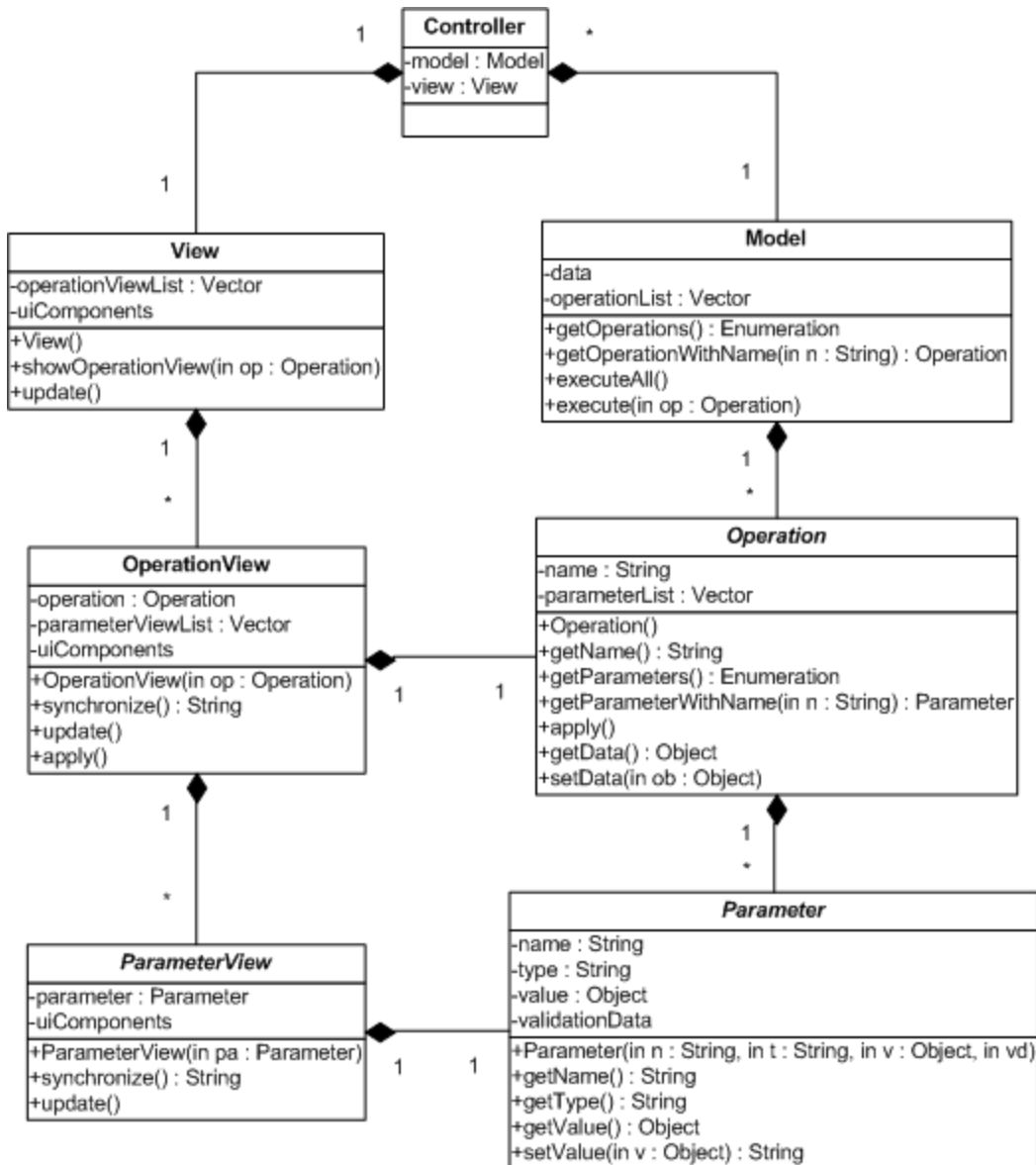


Figure 1 Class Diagram for the proposed system structure

### The *Parameter* class

Each *Parameter* is responsible for validating its own values. The constructor of the *Parameter* class accepts validation criteria as input along with its name, type and initial value. These criteria allow each *Parameter* to store its own set of rules for validating its values; this allows even *Parameters* of the same type to perform independent validation. The simplest way to enforce these validation rules is to check new values whenever they are passed to a *Parameter* via its *setValue* method, only valid values are made permanent. Validation errors can be returned to the caller of this method via a string describing its details, which can in turn be displayed to the user.

In some cases, the validation of a *Parameter*'s value must be performed in conjunction with the values of another *Parameter*. In such a scenario, it would be necessary to provide each *Parameter* with a reference to its containing *Operation* object, so that it can retrieve the necessary *Parameters* from it. Complex validation rules based on other *Parameter*'s values can be expressed through a simple use of an *Interpreter* [2].

### The *ParameterView* class

The *ParameterView* class is responsible for displaying its underlying *Parameter* in an appropriate manner for the user. The constructor of the *ParameterView* class takes a *Parameter* as input, at which point it can select the most appropriate user interface component to use given its type.

A *ParameterView* object and its underlying *Parameter* object are linked via synchronization and update methods of the *ParameterView* class.

- Whenever the synchronize method is called, the *ParameterView* attempts to set the values of its *Parameter* to those selected by the user in its user interface component. Any errors can either be displayed or forwarded to the caller.
- Whenever the update method is called, the *ParameterView* resets the values displayed in its user interface component to those of its *Parameter*.

### The *Operation* class

The *Operation* class represents the interface that all of its sub-classes must implement. It specifies that each sub-class must implement an apply method for performing its functionality, and also provides an interface for accessing the list of *Parameters* of each sub-class.

Sub-classes of the *Operation* class specify which of their parameters are editable by the user by placing corresponding instances of *Parameter* objects in the list of *Parameters*. The logical place for this to be done is in the Constructor of the sub-class. The apply method of each sub-class can access the values of each *Parameter* that it requires by retrieving the one with the appropriate name.

### The *OperationView* class

The *OperationView* class is responsible for displaying its underlying *Operation* in an appropriate manner for the user. The constructor of the *OperationView* class takes an *Operation* as input, at which point it can display general information such as its name, and create and display a *ParameterView* for each of its *Parameters*.

A means for executing the underlying *Operation* can be added to the *OperationView* via some form of active user interface component (i.e. a clickable button). When the execution is triggered, the *OperationView* can synchronize the values of each of its *ParameterViews* and then call the apply method of its *Operation* provided there are no errors.

## Implementation Details

### 1. Number of Parameter Types

When there number of types of parameters required by all of the operations in the system is limited, it is simplest to equip the *Parameter* and *ParameterView* classes with the ability to handle all of these types.

As the number of types of parameters increases, and the differences between these types becomes more apparent, a better design may be to handle each of these parameter types in sub-classes of the *Parameter* and *ParameterView* classes.

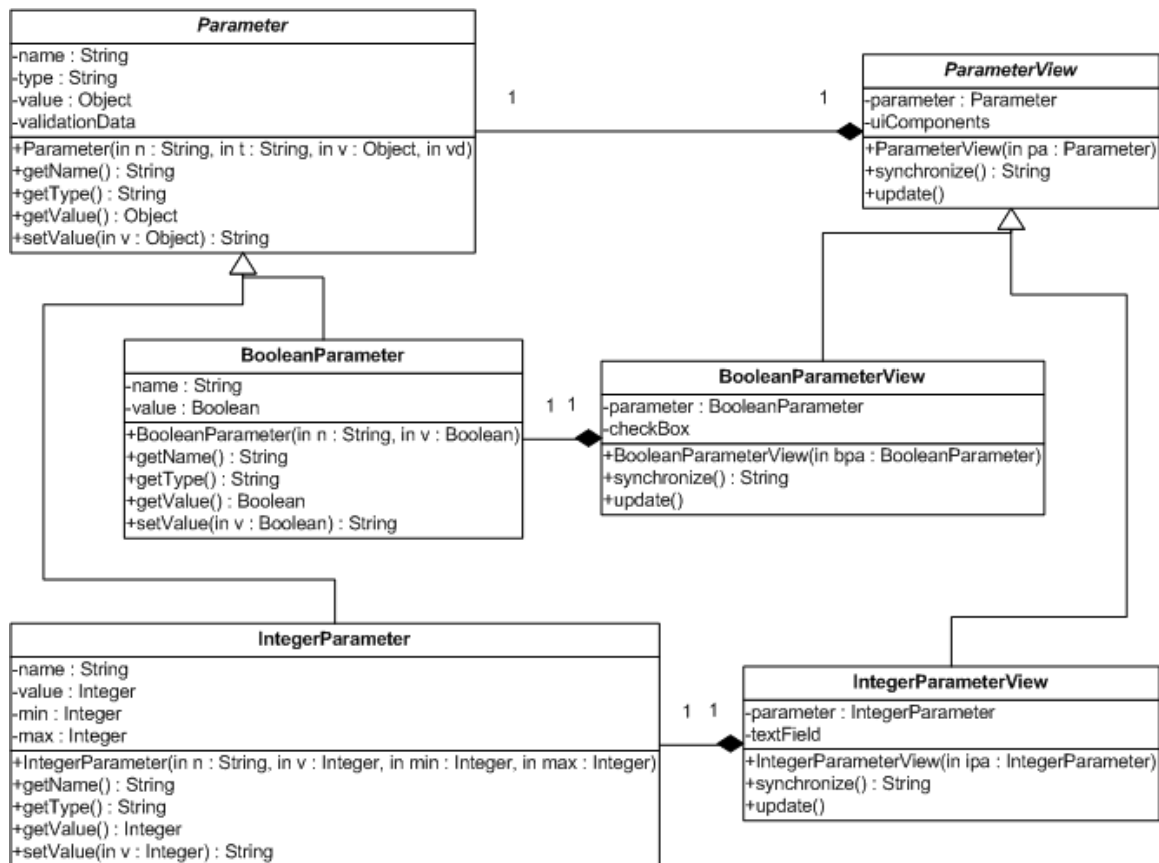


Figure 2 Class diagram of *Parameter* and *ParameterView* sub-classes

In this case, the addition of a *ParameterViewFactory* class may facilitate the creation of instances of *ParameterView* sub-classes from their corresponding *Parameter* sub-class. (See the *Abstract Factory Pattern* in [2]).

## 2. Retrieving the *Operations* from the *Model*

The *Model View Controller* pattern advocates that several *Controller-View* pairs may be connected to (observing) the same model. In this context, multiple users should be able to perform *Operations* independently and provide their own *Parameter* values, without these values being concurrently modified by other users. The *Prototype* pattern [2] can be used in this situation such that the *Model* returns a clone of an *Operation* that is requested by a *Controller-View* pair.

## 3. Providing the *Operations* with the data on which to operate

Prior to execution, each *Operation* needs to know what data to apply itself to. The simplest way to handle this is to have the *Model* pass the appropriate data to the *Operation* when a request is made for its retrieval by a *Controller-View* pair.

## 4. Allowing for changes to implementation of the user interface components

With the basic application of this pattern, there are three classes related to the user interface of the software application, the *View*, the *OperationView* and the *ParameterView* classes. A general scheme for anticipating future changes to the look and feel of the user interface would be to split these three classes into sub-classes for each look and feel implementation.

The trouble arrives if or when sub-classes of the *ParameterView* are created for the each type of *Parameter*. The number of classes that we need to split into sub-classes suddenly explodes.

The application of the *Bridge* pattern can help remedy this situation through the introduction of two separate class hierarchies for the *ParameterView* sub-classes that can be modified independently.

The application of the *Abstract Factory* pattern can also be of use to ensure that the correct look and feel sub-classes of the *OperationView* and *ParameterView* are instantiated for the *Operations* and *Parameters* of the system.

## 5. Triggering the execution of the *Operations*

The *Operation* objects of this pattern bear many similarities to the *Command* objects described by the *Command* pattern [2].

As suggested by the *Command* pattern, instances of *Operations* may be placed in the *View* so that they are triggered when the user selects them. A plausible example of this is placing some concrete instances of *Operation* sub-classes inside a menu, allowing the user to select them at their whim.

Other interesting features of the *Command* pattern that can equally be applied here are:

- The ability to support undo-functionality for *Operations* through simultaneous application of the *Memento* pattern [2]
- The ability to group *Operations* together to form composite *Operations* through simultaneous application of the *Composite* pattern [2]

## Examples

This section gives a very simple implementation of the image resize operation example mentioned previously.

For completeness, simplified source code for the four classes introduced by this pattern is provided below. Following this is sample source code for the image resize operation.

### The Parameter Class

```
public class Parameter
{
    private String name;
    private String type;
    private Object value;
    private boolean bounded;
    private String minValue;
    private String maxValue;

    public Parameter( String aName, String aType, Object aValue, boolean aBounded,
                    String aMinValue, String aMaxValue )
    {
        name = aName;
        type = aType;
        value = aValue;
        bounded = aBounded;
        minValue = aMinValue;
        maxValue = aMaxValue;
    }

    public String getName() { return name; }
    public String getType() { return type; }
    public Object getValue() { return value; }
    public boolean getBounded() { return bounded; }
    public String getMinValue() { return minValue; }
    public String getMaxValue() { return maxValue; }

    public String setValue( Object valueObject )
    {
        String errorString = "";
        if ( type.equals( "Integer" ) )
        {
            try
            {
                int newValue = Integer.valueOf( valueObject.toString() ).intValue();
                if ( bounded )
                {
                    int min, max;
                    min = Integer.valueOf( minValue ).intValue();
                    max = Integer.valueOf( maxValue ).intValue();
                    if ( min > newValue || newValue > max )
                    {
                        errorString += name + " must be between " + min + " and " + max;
                    }
                }
            }
            catch ( Exception e )
            {
                errorString += name + " is not an integer";
            }
        }
        else
        {
            value = valueObject;
        }
        return errorString;
    }
}
```

```

        }
        else
        {
            value = new Integer( newValue );
        }
    }
    else
    {
        value = new Integer( newValue );
    }
}
catch ( NumberFormatException nfe )
{
    errorString = name + " must be of type " + type;
}
}
else if ( type.equals( "Boolean" ) )
{
    value = valueObject;
}
return errorString;
}
}

```

## The Operation Class

```

import java.util.Vector;
import java.util.Enumeration;

public abstract class Operation
{
    protected String name;
    protected Vector parameters;
    protected Object data;

    public Operation()
    {
        parameters = new Vector();
        name = "Not Set";
    }

    public String getName() { return name; }
    public Enumeration getParameters() { return parameters.elements(); }
    public Object getData() { return data; }
    public void setData( Object theData ) { data = theData; }

    public abstract void apply();

    protected Parameter getParameterWithName( String aName )
    {
        Enumeration enumParams = getParameters();
        while ( enumParams.hasMoreElements() )
        {
            Parameter curParam = (Parameter)enumParams.nextElement();
            if ( aName.equals( curParam.getName() ) )
            {
                return curParam;
            }
        }
        return null; // Exception omitted for simplicity.
    }
}

```

## The ParameterView class

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ParameterView extends JPanel
{
    private Parameter parameter;
    private JLabel nameLabel;
    private JComponent valueComponent;
    private JLabel boundLabel;

    public ParameterView(Parameter aParameter)
    {
        parameter = aParameter;

        nameLabel = new JLabel( parameter.getName() );
        add(nameLabel);

        if ( parameter.getType().equals( "Boolean" ) )
        {
            JCheckBox cb = new JCheckBox();
            cb.setSelected( ((Boolean)parameter.getValue()).booleanValue() );
            valueComponent = cb;
            add( valueComponent );
        }
        else if ( parameter.getType().equals( "Integer" ) )
        {
            JTextField tf = new JTextField( parameter.getValue().toString() );
            valueComponent = tf;
            add( valueComponent );
        }

        if ( parameter.getBounded() )
        {
            if ( parameter.getMinValue() != null && parameter.getMaxValue() != null )
            {
                boundLabel = new JLabel( "(" + parameter.getMinValue() + " -> " +
                    parameter.getMaxValue() + ")" );
                add( boundLabel );
            }
        }
    }

    public void update()
    {
        if ( parameter.getType().equals( "Integer" ) )
        {
            ((JTextField)valueComponent).setText(parameter.getValue().toString());
        }
        else if ( parameter.getType().equals( "Boolean" ) )
        {
            ((JCheckBox)valueComponent).setSelected(
                ((Boolean)parameter.getValue()).booleanValue() );
        }
    }

    public String synchronize()
    {
        String errorString = "Unknown Parameter Type";
    }
}
```

```

if ( parameter.getType().equals( "Integer" ) )
{
    errorString = parameter.setValue( ((JTextField)valueComponent).getText() );
}
else if ( parameter.getType().equals( "Boolean" ) )
{
    errorString = parameter.setValue(
        new Boolean( ((JCheckBox)valueComponent).isSelected() ) );
}
return errorString;
}
}

```

### The OperationView class

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class OperationView extends JPanel implements ActionListener
{
    private Operation operation;
    private Vector parameterViews;
    private JLabel nameLabel;
    private JButton applyButton;

    public OperationView( Operation anOperation )
    {
        operation = anOperation;

        nameLabel = new JLabel( operation.getName() );
        add( nameLabel );

        ParameterView newParamView;
        Enumeration enumParams = operation.getParameters();
        while ( enumParams.hasMoreElements() )
        {
            newParamView = new ParameterView( (Parameter)enumParams.nextElement() );
            parameterViews.add(newParamView);
            add( newParamView );
        }

        applyButton = new JButton( "Apply" );
        applyButton.addActionListener( this );
        add( applyButton );
    }

    public void update()
    {
        Enumeration enumParamViews = operation.getParameters();
        while ( enumParamViews.hasMoreElements() )
        {
            ((ParameterView)enumParamViews.nextElement()).update();
        }
    }

    public String synchronize()
    {
        String tempString, errorString = "";
        Enumeration enumParamViews = operation.getParameters();
        while ( enumParamViews.hasMoreElements() )
    }

```

```

    {
        tempString = ((ParameterView)enumParamViews.nextElement()).synchronize();
        if ( tempString.length() != 0 )
        {
            errorString+= tempString + "\n";
        }
    }
    return errorString;
}

public void actionPerformed( ActionEvent e ) { apply(); }

public void apply()
{
    String errorString = synchronize();
    if ( errorString.length() == 0 )
    {
        operation.apply();
    }
    else
    {
        // display the errorString to the user.
    }
}
}

```

### The ImageResizeOperation class

```

import java.awt.Image;

public class ImageResizeOperation extends Operation
{
    public ImageResizeOperation()
    {
        super();
        name = "Resize an Image";
        parameters.add( new Parameter( "Width Ratio", "Integer", new Integer(100), true, "1", "500" ) );
        parameters.add( new Parameter( "Height Ratio", "Integer", new Integer(100), true, "1", "500" ) );
        parameters.add( new Parameter( "Constrain Proportions", "Boolean", new Boolean(true), false, "", "" ) );
    }

    public void apply()
    {
        Image theImage;
        try
        {
            theImage = (Image)data;
        }
        catch ( ClassCastException cce )
        {
            // Error handling code goes here.
            return;
        }
        int widthRatio = ((Integer)getParameterWithName( "Width Ratio" ).getValue()).intValue();
        int heightRatio = ((Integer)getParameterWithName( "Height Ratio" ).getValue()).intValue();
        boolean constrainProportions = ((Boolean)getParameterWithName(
            "Constrain Proportions" ).getValue()).booleanValue();
        if ( constrainProportions )
        {
            heightRatio = widthRatio;
        }
        int newWidth = theImage.getWidth( null ) * widthRatio;
        int newHeight = theImage.getHeight( null ) * heightRatio;
    }
}

```

```
data = theImage.getScaledInstance( newWidth, newHeight, Image.SCALE_DEFAULT );
}
}
```

Using this framework, very little extra coding is required in the image editing application to acquire the parameter values from the user that are necessary to resize an image.

If the user selects an option to resize the current image from within the application the following steps are required:

- The Controller obtains an instance of an ImageResizeOperation object from the Model.
- The Controller passes this ImageResizeOperation to the View.
- The View initializes a new OperationView with the given ImageResizeOperation and displays it.

**Resulting Context**

The result of applying this pattern is a software system that will easily allow the addition of extra modules of functionality (called *Operations*).

UI Components for the input of parameter values for each Operation will be generated automatically.

There are a number of benefits of applying this pattern:

- New *Operations* can be provided with an associated user interface for obtaining parameter values with little or no extra cost, provided they use existing parameter types.
- The number and type of parameters required for an *Operation* can potentially be changed at run-time.
- Validation of the values for a parameter is encapsulated within the *Parameter* class that defines it.
- Each *Operation* is encapsulated within its own class, making updates and modifications easier to manage.
- Grouping a set of *Operations* together to form a more complex *Operation* can be easily achieved due to their modularity.
- Snapshots of intermediate results can be achieved when applying a sequence of *Operations* to a data source. This can also lead to convenient undo functionality.
- A new look and feel standard can be applied to the user interface with a minimal amount of effort.

There are also some drawbacks to applying this pattern:

- *Operations* must access certain values from *Parameter* object which is more cumbersome than simply using a primitive variable/Object of the same type. Also, since *Parameters* are retrieved by their name instance variable, there is a potential for errors which will not be caught at compile time.

- The addition of new types of parameters requires modifications to both the *Parameter* and *ParameterView* classes. If sub-classing of these classes is employed, then two new classes must be created for each new type of parameter that is introduced. This may also require modification of a factory class that builds instances of sub-classes of the *ParameterView* class for their associated *Parameter* subclass.
- In some contexts, splitting the functionality of a software application into distinct modules of functionality (i.e. *Operations*) may be unwieldy or even impossible. Similarly, it may be troublesome or impossible to divide the data of the model into pieces that can be processed by a single *Operation*. Under these circumstances, the application of this pattern may be ill-advised.

### **Rationale**

The solution successfully eliminates the overhead of writing code to acquire values for parameters of an operation from the user.

Once *Parameter* classes and their corresponding *ParameterView* classes are implemented, there is very little overhead involved to get parameter values from the user for a new *Operation*. The bulk of this overhead is spent determining which variables of the *Operation* should be made into *Parameter* objects to allow for user input. The work to accomplish the instantiation of these *Parameter* objects and store them in a list is trivial when compared to the work required to manually acquire parameter values from the user.

The use of *Parameter* objects requires the *Operations* to retrieve certain values necessary for their execution via instance methods of each *Parameter*. While this might incur a slight performance penalty, it is a price worth paying for the efficient reuse of existing source code.

### **Related Patterns**

- This pattern can be seen as an extension to the *Model-View-Controller* Pattern [1].
- The *Abstract Factory* can be used to create the UI components and allow the look and feel standard of the UI to be changed easily/dynamically [2].
- The *Prototype* pattern can be used to create instances of an *Operation* which independent users can execute [2].
- The *Bridge Pattern* can be used to allow the user interface component implementation to be changed easily/dynamically even when there are several *ParameterView* subclasses. [2].
- The *Composite* pattern can be used to allow several *Operations* to be grouped together to form a more complex *Operation* [2].
- The *Operation* objects introduced by this pattern have several similarities with the *Command* objects from the *Command* pattern. [2]

- The *Memento* Pattern is linked with the *Command* pattern, and might be useful for keeping track of state before and after *Operations* are performed, allowing some form of undo functionality.
- The *Interpreter* pattern can be used to allow complex parameter validation rules based on other parameters of the same operation. [2]

### **Known Uses**

Harris, J., *Road-Network Graph Extraction from Map Images*, Carleton University Undergraduate Honours Project, 2001.

### **References**

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, 2<sup>nd</sup> edition, John Wiley & Sons Ltd., 2000.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.