User’s Guide to the uiGenerator

John A. Konglathu

21 January 1999
uiGenerator is a program that generates a form-based interface for Java objects by inspecting their structure. This interface can be customised to a certain extent. It recognizes JavaBeans patterns which might be present in the object and uses these patterns to advantage in generating a better interface. This document should help a new user in setting up and running a few examples using the uiGenerator.

1 Getting started

uiGenerator should work with JDK1.2. It relies on the swing libraries for its default user interface components. All you have to do to start using the program is to include the JAR file in your CLASSPATH.

1.1 Basic use

We first need an object to edit. Consider a simple budget application, which computes the function

\[ \text{Total} = \text{DirectCosts} + \text{NumberOfResearchers} \times 2000 \]

The code to accomplish this is given below. DirectCosts and NumberOfResearchers are public fields in the class. A method called computeTotal, performs the computation and returns Total.

```java
public class Budget {
    public float DirectCosts;
    public int NumberOfResearchers;

    public Budget() {
        DirectCosts = 0;
        NumberOfResearchers = 0;
    }

    public float computeTotal() {
        return DirectCosts + NumberOfResearchers*2000;
    }
}
```

Now we need the code to generate the interface for this object. For this, we invoke a static method on the uiGenerator class passing the object to be edited as an argument. This method returns a uiFrame object (a subclass of javax.swing.JFrame) which can then be displayed with a setVisible(true) method call.
import bus.uigen.*;

public class BudgetEditor {
    public static void main(String args[]) {
        Budget b = new Budget();
        uiFrame editor =
            uiGenerator.generateUIFrame(b, null);
        editor.setVisible(true);
    }
}

On running the BudgetEditor program, we get a window with two text fields labelled DirectCosts and NumberOfResearchers. These fields can be edited by typing the values into their corresponding text fields. Once the values, say 1000.00 for DirectCosts and 2 for NumberOfResearchers has been entered, the computeTotal() method can be invoked from the Methods menu. On selecting the computeTotal method from this menu, a method invocation window pops up with an Invoke Method button. Since this method has no parameters, no arguments need to be filled in the method invocation window. When the Invoke Method button is pushed, the computeTotal() method is called on the Budget object and since it returns a float value, this value is displayed in a new window. This is a really cumbersome interface to the Budget object, but it works and it didn’t take any special coding techniques. We could get a much better and simpler to use interface by following certain coding patterns.

1.2 A slightly better interface

What we really need is a text field for Total in the editor window. We could put in a public variable called Total in the Budget class to do this, but then this field would be editable and the object could potentially be in an inconsistent state. In order to make the Total field uneditable, we encode it as a JavaBean read-only property. To do this we provide a getTotal method which returns a float value representing the Total costs. The improved Budget class is presented below.

public class Budget {
    public float DirectCosts;
    public int NumberOfResearchers;

    public Budget() {
        DirectCosts = 0;
        NumberOfResearchers = 0;
    }
}
public float getTotal() {
    return DirectCosts + NumberOf Researchers * 2000;
}

This code is not very different from the earlier version. The computeTotal method has been renamed to getTotal which tells the uiGenerator that we intend it to be a read-only field. Let's examine the interface that BudgetEditor generates for this class. We get an additional read-only field labelled Total which represents the total costs. After changing DirectCosts and NumberOfResearchers, the value returned by getTotal has changed, but the editor has no way of knowing this. To see the current state of the Budget object, we refresh the editor by selecting Refresh from the Edit menu. This causes the Total field to display the new value returned by getTotal. This is a much better interface than the earlier one for this application, but it would really be nice if the Total field could change when either of the other two variables are changed. To do this we have to rewrite the Budget application such that it generates events, whenever these variables are changed, which the budget editor can listen to and interpret.

1.3 Using property change events

To get the interface described in the previous subsection, we have to rewrite the DirectCosts and NumberOfResearchers fields as read-write JavaBeans properties. This is done by replacing the field with a getField method that returns its value and a setField method that sets its new value. Field is the property’s name. Now the value of the property can be changed only through its set method. By adding code in the set method, we can arrange for the Budget editor to be notified when either of the two fields is changed. The code that does this is included below.

```java
import java.beans.*;

public class Budget {
    private float DirectCosts;
    private int NumberOfResearchers;

    private PropertyChangeSupport propertyChange;

    public Budget() {
        DirectCosts = 0;
        NumberOfResearchers = 0;
        propertyChange = new PropertyChangeSupport(this);
    }
}```
public float getDirectCosts() {return DirectCosts;}
public void setDirectCosts(float cost) {
    DirectCosts = cost;
    propertyChange.firePropertyChange("directCosts", null,
            new Float(cost));
    propertyChange.firePropertyChange("total", null,
            new Float(getTotal()));
}

public int getNumberOfResearchers() {
    return NumberOfResearchers;
}
public void setNumberOfResearchers(int n) {
    NumberOfResearchers = n;
    propertyChange.firePropertyChange("numberOfResearchers", null,
            new Integer(n));
    propertyChange.firePropertyChange("total", null,
            new Float(getTotal()));
}

public float getTotal() {
    return getDirectCosts()+getNumberOfResearchers()*2000;
}

public void addPropertyChangeListener(PropertyChangeListener l) {
    propertyChange.addPropertyChangeListener(l);
}

When BudgetEditor is run, is puts up a window with three fields, two editable fields representing DirectCosts and NumberOfResearchers and one uneditable field representing Total. When either of the editable fields is changed, Total changes to reflect it. We did need some extra coding to achieve this. This would be required even if the interface was hand-coded. The semantics of Total depending on DirectCosts and NumberOfResearchers would have to be encoded someway or the other.

2 More features

There are a few more features supported by /uigen. These include configuration of a generated interface, hiding certain fields and saving and restoring object state.
2.1 Load/Save

In order to let the Budget Editor load and save the Budget object, the Budget class has to implement the java.io.Serializable interface. If the object implements the Serializable interface, the generated editor will have menu items under the File menu to save the object into a file or load it from a previously saved file. The code that does this for the original Budget class is given below.

```java
import java.io.*;

class Budget implements Serializable {
    float DirectCosts;
    int NumberOfResearchers;

    public Budget() {
        DirectCosts = 0;
        NumberOfResearchers = 0;
    }

    public float getTotal() {
        return DirectCosts + NumberOfResearchers * 2000;
    }
}
```

This code enables the Budget Editor application to load and save instances of the Budget class.

2.2 Selecting a sub-object

It might become necessary to name a sub-object within an object. This is useful while invoking methods, or configuring the interface for example. Instead of naming the sub-objects explicitly, we have a pointer-based selection method. To select a field or sub-object within the main object, simply click on it. The highlighted area represents the current selection. Only one field can be selected at a time. Another way of selecting a sub-object that is not a simple field is to select one of its constituent fields and use the Select-Up/Select-Down menu items under the Edit menu to select the immediate parent/child of the current selection. The current selection can be used as a parameter to a method invocation by using the Paste menu item from the Popup menu associated with the parameter name.
2.3 Invoking Methods

The generated user interface allows a user to invoke any of the public methods of its object. These methods are listed under the Methods menu. The Methods menu is divided into sub-menus for inherited methods (those defined in parent classes), bean methods (those that represent a JavaBeans property) and the rest. On selecting an item from the Methods menu, a window pops up. It has fields representing method parameters and a button to invoke the method. Each parameter is represented by a label for the parameter number, an option box for the parameter type and a text field, which shows the current value of the argument. There are several ways of filling in the arguments. This is done using the popup menu associated with the parameter label. The menu has an item for each way of filling in the parameter. Before filling in an argument, though, the argument type can be changed to a subclass of the displayed type, by typing in the new class name in the option box, or selecting one of its choices.

2.3.1 Creating an argument

Selecting the Create option from the parameter popup menu brings up a list of constructors for the type specified in the type option box. Choose a constructor. This brings up a method invocation window similar to the one we started with. Fill up the constructor arguments and "invoke" the constructor to create the object which will be a method argument.

2.3.2 Editing the argument

Selecting the Edit option from the popup menu brings up an editor window which represents an instance of the parameter class created by using one of its constructors. Once the object has been edited, it can be added to the parameter list by selecting the Done menu item from the File menu.

2.3.3 Pasting an argument

Choosing the Paste option from the menu results in using the current selection as the argument. If the current selection is of a type unsuitable for the parameter, an error message is printed.

Once the argument has been filled in, its value appears in the text field to the right of the parameter type option-box.

2.4 Configuring a user interface

2.5 Configuration at program-writing time