JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, no. 4, September-October 2002

Streamlined Algorithm Deployment via JavaBeans

Patrick Chisan Hew, Defence Systems Analysis Division, Defence Science and Technology Organisation, Canberra, Australia

Abstract

This paper introduces JavaBean Calculation Engines (JBCEs), a procedure for deploying algorithms in a standard JavaBean framework. For the algorithm developer, JBCEs are a finishing point for deploying an algorithm as a "qualified product". For the GUI developer, JBCEs shield internal computations behind a JavaBean interface. By standardising the interface between computational and GUI code, JBCEs improve reusability and maintainability.

1 INTRODUCTION

It is increasingly attractive to use Java for software deployment. Unfortunately, the expertise for developing algorithms need not be coupled to an expertise in software engineering or Graphical User Interface (GUI) development, and this can result in software where the computational code is intertwined with the code for the user interface. The reuse, therefore, of computational code may well be impossible without extensive "reverse engineering", and the maintenance of GUI code may require an understanding of the computations. Both situations are unattractive.

This paper introduces *JavaBean Calculation Engines (JBCEs)*, a procedure for deploying Java computational code as reusable modules, ready for GUI integration. For the algorithm developer, JBCEs are a finishing point for deploying an algorithm as a "qualified product". For the GUI developer, JBCEs shield internal computations behind a JavaBean interface. JBCEs could thus streamline software development, by standardising the interaction between "research" algorithm code and "deployment" GUI code, and having algorithms packaged for reuse.

This paper formulates the Calculation Engine concept, and then demonstrates the construction of a JBCE through a simple example. It then discusses how the JBCE functionality is implemented.

Cite this column as follows: Patrick Chisan Hew: *Streamlined Algorithm Deployment via JavaBeans*, in Journal of Object Technology, vol. 1, no. 4, September-October 2002, pages 75-91. http://www.jot.fm/issues/issue_2002_09/column6

2 BACKGROUND

Motivation

This work arose out of the needs of the Theatre Operations Branch (TOB) at the Defence Science and Technology Organisation. TOB supplied operational analysis support to the Australian Defence Force, motivating a software toolkit that deployed algorithms in a user-friendly form.

While skilled practitioners, TOB developers were not professional "code cutters"; moreover, the GUI needed to be developed in an evolutionary fashion. The algorithms themselves, however, were expected to be relatively "stable" as reusable assets (Schmidt 1999a). Hence, and in order to get the best value out of contractor support, algorithms needed to be packaged for streamlined deployment into a GUI, while maintaining "black box" isolation.

JavaBeans

JavaBeans[1] are a popular commercial standard for software components (Doherty et al 2000). The most visible examples of JavaBeans are GUI libraries like Swing, and JavaBeans can be further wrapped into Enterprise JavaBeans[2], enabling distributed applications (Asbury et al 1999).

JavaBeans are characterised by their standardised interaction with the outside world. The specification defines how *properties* are *accessed*, and how we can be *informed* of changed properties; to summarise: access to property `XXX` is provided through a method `getXXX()`, change via a method `setXXX()`, and when a property is changed, the JavaBean is expected to transmit a `PropertyChange` event to registered listeners. JavaBeans must also handle registration of listeners, and self-storage for later re-use.

We thus see that our goal of wrapping algorithms into JavaBeans is highly desirable – it makes an algorithm accessible to Java developers with "mainstream" skills, rather than just its inventors. The problem, then, is in building a generic procedure for packaging an algorithm in a JavaBean. This requires some assumptions about the algorithms, but we shall see that these are weak.

[1] http://java.sun.com/beans/
[2] http://java.sun.com/products/ejb/

3 CONCEPT

We first formulate *Calculation Engines* (CEs), and then consider implementation via JavaBeans.

Calculation Engines – Key Principles

The key principles behind CEs are:

1. *Input-Output Algorithm.*
   An *Input-Output Algorithm* is a process that takes *Inputs* and returns *Outputs*. The term emphasises the responsibility of the algorithm designer to fully specify the inputs to the algorithm, and the outputs that are returned.
2. *Equilibrium*
   A Calculation Engine is set up as an object that maintains *equilibrium* between the Inputs and Outputs of the internal Algorithm. In essence, the CE keeps itself in a "solved state", and is only out of equilibrium when recalculating.

The basic chain of events is illustrated by Figure 1. As shown, we have an Input-Output Algorithm fully enclosed by the Calculation Engine, and a number of subscribing *Clients* outside. When a Client modifies an Input, the CE calls the Algorithm to recalculate the Outputs, and communicates the changed Input and Outputs to all subscribing Clients. The CE is "out of equilibrium" in the time between receiving the modified Input and communicating the change.

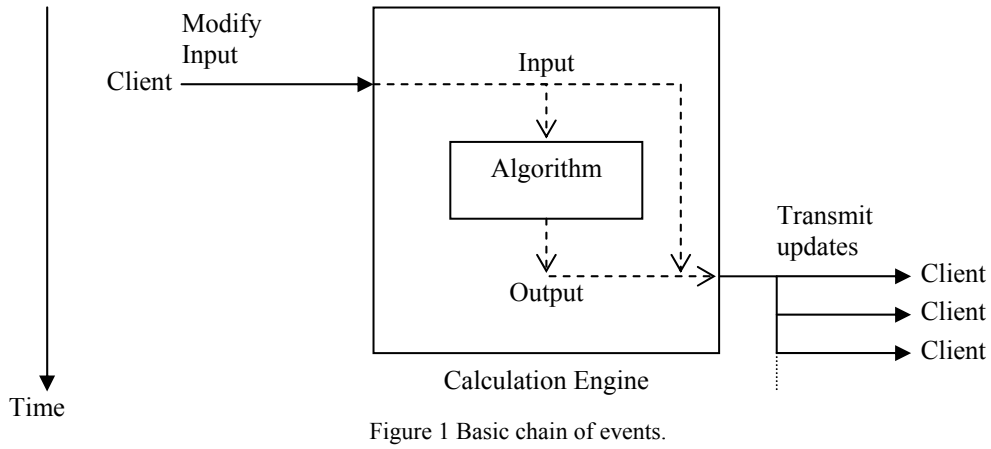


Figure 1 Basic chain of events.

The interaction between CE and Clients is basically the Observer (Model-View-Controller) design pattern (Gamma et al 1995). The point, however, is in streamlining the *construction* to this pattern – setting up the dashed lines of Figure 1 for an *arbitrary* Input-Output Algorithm.

The abstract Input-Output Algorithm could be regarded as the "true" reusable element (Krueger 1992), with the CE being an artifact (Mili et al 1995). Input-Output form is useful in its own right (Kühne 1997), however the CE concept goes beyond the provision of high-level access to low-level code (Schmidt 1999b) (Novak 1997) – it enables multiple Clients to "gather around" a problem. We shall see the benefits to GUI programming.

Using the JavaBean Framework

To implement a CE as a JavaBean, the Inputs and Outputs are declared to be properties of the JavaBean; specifically, the Inputs are set up as bound and constrained properties, with `get` and `set` methods, and the Output is a bound property with a `get` method. The remaining work lies in connecting Input to Output through the enclosed Algorithm.

The JBCE functionality has been coded into a package `javabeancalcengines`, the use and construction of which will be discussed. Since Java can wrap native code, JBCEs can package algorithms written in languages other than Java. The Calculation Engine concept could also be applied in other Object Oriented Programming languages, using equivalents to the `PropertyChange` mechanism (Larman 1999).

4 EXAMPLE

To illustrate the construction and use of JBCEs, we work through *Adding Applet*, a screenshot of which is shown in Figure 2. We show how the JBCE is constructed to deploy the algorithm, and then how the JBCE can be used from the GUI.

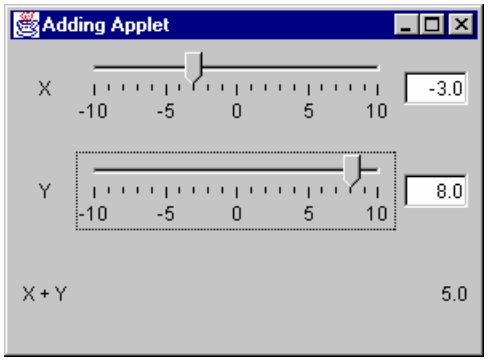
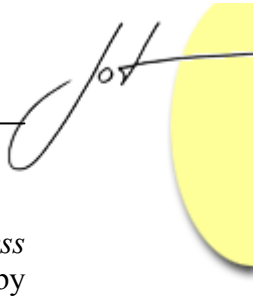


Figure 2 Adding Applet.

Algorithm in Input-Output Form

The starting point is to have the algorithm fully specified in Input-Output form. Here, we do this through a class `AddingAlgorithm` and a method with signature –

```
public static double addingalgorithm(double num1, double num2)
```



The `static` declaration reflects the fact that `addingalgorithm()` is only the *process* for transforming `num1` and `num2` to the output; the actual *instance* of a solution is held by the overarching JBCE. We note that the Output from an algorithm may require a return class of some kind, particularly for multiple outputs.

To build the JBCE, we do not need to know anything about how `addingalgorithm()` works. Furthermore, we can actually build the JBCE with some resistance to run-time errors in the algorithm.

Wrapping into a Calculation Engine

We are now able to produce the JBCE, `AddingBean`. In the description that follows, we assume the use of the `javabeancalcengines` package, though the focus will be on how this package fulfills the Calculation Engine concept.

Setting Up

A JBCE is a JavaBean, so we need to import this behaviour –

```
import java.beans.*;
import javabeancalcengines.*;

public class AddingBean extends DefaultJavaBeanCalcEngine {
```

The use of `DefaultJavaBeanCalcEngine` is not compulsory, but it gives us access to utility methods that will simplify construction.

Input Properties

We recall that the inputs to `addingalgorithm()` were `num1` and `num2`. To formulate `num1` as a property, we have –

```
/**
 * @serial the first number to add
 */
public double Num1;

/**
 * Set the first number to add.
 *
 * @param num1 the new first number to add
 */
public void setNum1(double newNum1) throws PropertyVetoException {
  setJavaBeanEngineInput("Num1", "Num1", newNum1);
} // setNum1(double)
```

```
/**
 * Get the first number to add.
 *
 * @return the first number to add
 */
public double getNum1() {
  return Num1;
} // getNum1()
```

The Input parameter `num1` is mapped to the field `Num1`. The `setNum1()` method leverages `setJavaBeanEngineInput()`, details of which will follow; to summarise, the method:

1. Provides Clients the opportunity to veto the impending change to `Num1`.
2. Calls `addingalgorithm()` to recalculate[3].
3. Assigns `Num1` the new value of `newNum1`, informing Clients of the change.

Parameter `num2` is handled in a similar fashion. It is worth noting that the Clients are given the opportunity to veto *before* any changes are made. This means that when Clients do react to a change, they can do so without having to allow for rollback.

Output Property

With the Inputs in place, we can set up the code for the Output –

```
/**
 * @serial the Adding Algorithm result
 */
private double AddingResult;

/**
  * Recalculate after an input has been specified.
  */
public void doRecalculate() {
  doCalcAddingResult();
} // doRecalculate()

/**
  * Force a calculation now.
  */
public void doCalculateNow() {
  doCalcAddingResult();
} // doCalculateNow()

/**
  * Calculate the Adding result.
  */
```

[3] If automatic recalculation is active. If not, the `AddingBean` notes that it is out of equilibrium.

```
private void doCalcAddingResult() {
  double oldAddingResult = AddingResult;
  double newAddingResult = Double.NaN;

  startCalculating();
  newAddingResult = AddingAlgorithm.addingalgorithm(Num1,Num2);
  stopCalculating();

  AddingResult = newAddingResult;
  firePropertyChange("AddingResult",
       new Double(oldAddingResult), new Double(newAddingResult));
} // doCalcAddingResult()

/**
 * Return the Adding result.
 *
 * @return the Adding result
 */
public double getAddingResult() {
  return(AddingResult);
} // getAddingResult()
```

The Output is mapped to the field `AddingResult`. The calls in `doCalcAddingResult()` preserve `AddingResult` until `addingalgorithm()` is finished, and the enclosing `startCalculating()` – `stopCalculating()` pair keeps Clients informed about the `AddingBean`'s equilibrium state.

Initialisation

When initialising the `AddingBean`, we must ensure that it is in equilibrium –

```
public AddingBean() {
  super();
  initSafeState();
} // AddingBean()

private void initSafeState() {

  // Input.

  Num1 = 0;
  Num2 = 0;

  // Output.

  AddingResult = AddingAlgorithm.addingalgorithm(Num1,Num2);
} // initSafeState()
```

The call to `initSafeState()` ensures that the `AddingBean` is in a "solved" state.

GUI Development

In developing the GUI for `AddingApplet`, we will see its compartmentalisation from the `AddingBean`. Furthermore, while GUI development is very specific to the interface being built, the mechanisms for working with a JBCE like `AddingBean` parallel those for GUI components.

Obtaining a Calculation Engine

We start by having `AddingApplet` set up an `AddingBean` for use –

```
    // Start the applet.
    public void start() {
      getAddingBeanInstance();
      …
    }

…

    AddingBean itsAddingBean;

    public AddingBean getAddingBean() {
      return(itsAddingBean);
    } // getAddingBean()

    private void getAddingBeanInstance() {
      itsAddingBean = new AddingBean();
    } // getAddingBeanInstance()
```

The call to `getAddingBeanInstance()` instantiated an `AddingBean` directly. We could, however, have created the `AddingBean` elsewhere and supplied it to the `AddingApplet`; indeed, the `AddingBean` might even have been packaged as an Enterprise JavaBean and supplied off a server.

Identifying the Clients

Referring back to the screenshot in Figure 2, we notionally have 5 Clients:

1. A slider and textfield for `Num1`.
2. A slider and textfield for `Num2`.
3. A label field displaying `AddingResult`.

A rigorous analysis would establish the correct implementation of these notional Clients. However, for illustration, we will do the following:

1. Create `Num1Listener` and `Num2Listener` to listen to the sliders and `AddingBean`.
2. Equip the textfield to forward user input to the `AddingBean`.
3. Configure the applet to forward `AddingBean` changes to components.

Figure 3 illustrates these communication responsibilities, to be discussed below. It is worth comparing Figure 3 with the basic chain of events schematic of Figure 1 – we see the User calls to set `AddingBean` Inputs, and the `propertyChange()` update events that are sent back to the Clients in turn.

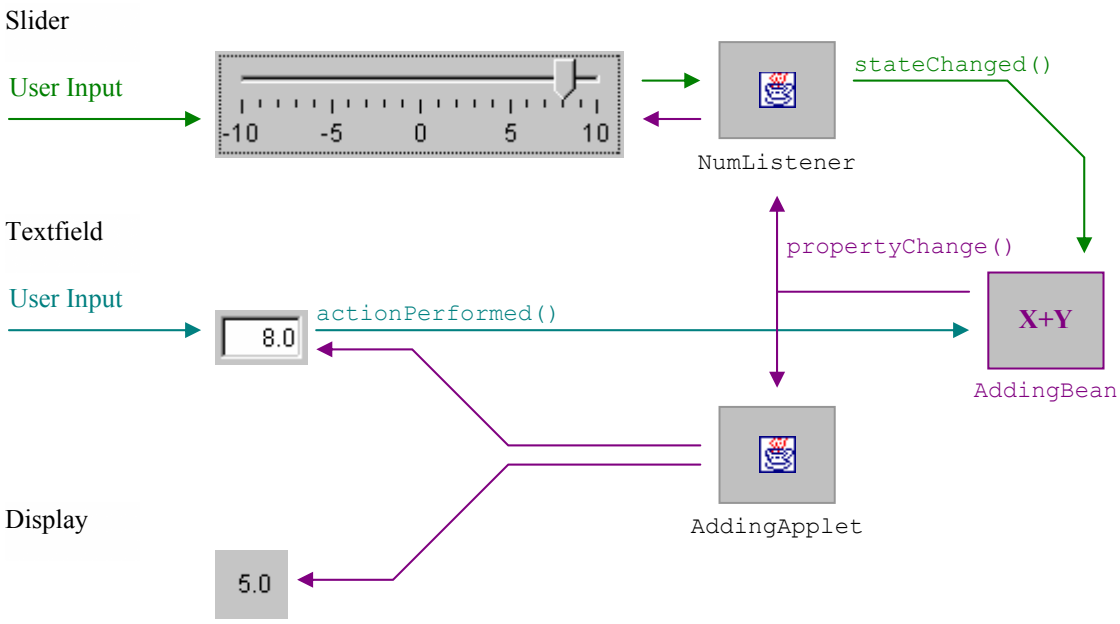


Figure 3 GUI Communication with the AddingBean.

Equipping a `JSlider` – Live Control

The listeners `Num1Listener` and `Num2Listener` listen to the sliders for user changes to `Num1` and `Num2`, and to the `AddingBean` for external changes. For user input, we implement the `ChangeListener` interface, defining a `stateChange()` method –

```
public void stateChanged(ChangeEvent evt) {
  if(!isExternallyForcedChange() & !isAlreadyTryingToReact()) {

    JSlider source = (JSlider) evt.getSource();
    double theNewValue = (double) source.getValue();

    startReacting();
```

```
      try {
        setdoubleInput(theNewValue);
      } catch(
        PropertyVetoException ePV) {

        PropertyChangeEvent ePC = ePV.getPropertyChangeEvent();
        double restoreValue = ((Double)
               ePC.getOldValue()).doubleValue();

        // The following causes a recursive call that is caught by
        // isAlreadyTryingToReact().

        source.setValue((int) restoreValue);
        source.updateUI();

      } // catch(...)

      stopReacting();
    } // if
  } // stateChanged(ChangeEvent)
```

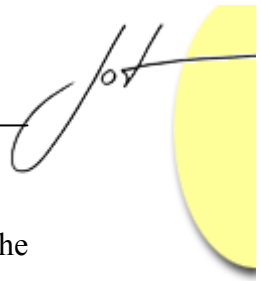
The listeners `Num1Listener` and `Num2Listener` provide the final link to the `AddingBean` by implementing `setdoubleInput()`; for `Num1` –

```
  /**
   * Set the first number to add.
   *
   * @param aFromSliderValue is the value on the JSlider control.
   */
  public void setdoubleInput(double aFromSliderValue)
      throws PropertyVetoException {
    getParent().getAddingBean().setNum1(aFromSliderValue);
  } // setdoubleInput(double)
```

If we look again at `stateChanged()`, we can see why the sliders could be called "live" – changes by the user are immediately forwarded to the `AddingBean`. We note that this requires the internal `doCalcAddingResult()` algorithm to be fast enough to take advantage of this immediate input.

If an exception is thrown, the `catch` handler users the `PropertyVetoException` to restore a safe value. Since this requires that the slider be set to a value, we need to trap the infinite recursion by testing for "already reacting to the user". This is handled through `isAlreadyTryingToReact()` and the `startReacting()`-`stopReacting()` pair.

In a similar vein, if the `AddingBean` is setting the value of the slider on behalf of another client, we need to know not to react. We thus have `isExternallyForcedChange()`, the code for which has been omitted, but it suffices to say that it interrogates a flag stored by the `AddingBean`. With this in place, we can



listen for changes coming from the `AddingBean`, through the `PropertyChangeListener` and `VetoableChangeListener` interfaces –

```
    public void propertyChange(PropertyChangeEvent evt) {
      double newVal = ((Double) evt.getNewValue()).doubleValue();
      itsJSlider.setValue((int) newVal);
    } // propertyChange(...)

...

    public void vetoableChange(PropertyChangeEvent evt)
        throws PropertyVetoException {

      double newVal = ((Double) evt.getNewValue()).doubleValue();
      if(newVal > itsJSlider.getMaximum()) {
        throw new PropertyVetoException("Too Big.", evt);
      } // if
      else if(newVal < itsJSlider.getMinimum()) {
        throw new PropertyVetoException("Too Small.", evt);
      } // else if
    } // vetoableChange(...)
```

It is worth emphasising the roles of the two methods. The `AddingBean` will call `vetoableChange()` first, providing the opportunity to veto a proposed change, as seen in the tests for exceeding the bounds of the slider. The call to `propertyChange()` is made later, at which time the slider's value can be set.

The final step is to instantiate the listeners and supply them to the sliders. This is discussed later.

Equipping a `JTextField` – Action on Return

The model for the textfield was for the user to type in what they wanted, with the contents being sent to the `AddingBean` when they press Enter[4]. The textfield is thus equipped with an `actionPerformed()` handler, implemented[5] as below for `Num1` –

```
  void jTextFieldNum1_actionPerformed(ActionEvent e) {

    String inputText = jTextFieldNum1.getText();

    try {
      double parsedValue = Double.parseDouble(inputText);
      getAddingBean().setNum1(parsedValue);
    } catch(Exception x) {
```

[4] We note that until the user presses Enter, the textfield is not synchronized with the `AddingBean`.
[5] The implementation used Borland JBuilder Pro Ver3.00, so the code reflects the automatic generation idioms.

```
      // Something went wrong. Take value from Engine.

      jTextFieldNum1.setText(Double.toString(getAddingBean().
              getNum1()));
    } // catch(Exception)
  } // actionPerformed()
```

The exception handler deals with bad user input and external vetoes in the same way – restoring from the current value in the `AddingBean`.

Farming Out Updates to Components

Although the textfields forward user input to the `AddingBean`, they are not equipped to handle changes coming the other way. Similarly, the label displaying the `AddingResult` needs to be kept up to date. Rather than building listeners for each of these components, we make the overarching applet do the work. We thus implement the `AddingApplet` against `PropertyChangeListener` –

```
  public void propertyChange(PropertyChangeEvent evt) {

    String propName = evt.getPropertyName();

    if(propName == "AddingResult") {
      jLabelAddingResult.setText(evt.getNewValue().toString());
    } // if
    else if(propName == "Num1") {
      jTextFieldNum1.setText(evt.getNewValue().toString());
    } // else if
    else if(propName == "Num2") {
      jTextFieldNum2.setText(evt.getNewValue().toString());
    } // else if
  } // propertyChange(...)
```

Registering the Listeners

With the `AddingBean` instantiated and the listeners defined, we may now connect them all up. This is done in `AddingApplet` in the `start()` method –

```
    // Start the applet.
    public void start() {
      getAddingBeanInstance();
      registerComponents();
    }

  …

    private void registerComponents() {
```

```
        // Initialise initial values for controls.

        double theNum1 = getAddingBean().getNum1();
        jSliderNum1.setValue((int) theNum1);
        jTextFieldNum1.setText(Double.toString(theNum1));

        double theNum2 = getAddingBean().getNum2();
        jSliderNum2.setValue((int) theNum2);
        jTextFieldNum2.setText(Double.toString(theNum2));

        double theAddingResult = getAddingBean().getAddingResult();
        jLabelAddingResult.setText(Double.toString(theAddingResult));

        // Register listeners.

        getAddingBean().addPropertyChangeListener(this);

        Num1Listener lisNum1 = new Num1Listener(this, jSliderNum1);
        jSliderNum1.addChangeListener(lisNum1);
        getAddingBean().addPropertyChangeListener("Num1", lisNum1);
        getAddingBean().addVetoableChangeListener("Num1", lisNum1);

        Num2Listener lisNum2 = new Num2Listener(this, jSliderNum2);
        jSliderNum2.addChangeListener(lisNum2);
        getAddingBean().addPropertyChangeListener("Num2", lisNum2);
        getAddingBean().addVetoableChangeListener("Num2", lisNum2);

    } // registerComponents()
```

We can see the importance of ensuring that the `AddingBean` starts in equilibrium, for we have used its initial values as the initial values of the controls. We also see that the overarching applet will receive all `PropertyChange` events posted by the `AddingBean`, but that the listeners for `Num1` and `Num2` will only receive `PropertyChange` events for their respective properties[6].

The User Experience

The user will be able to use the sliders or textfields to specify the numbers to be added, and will see the result in the bottom-right corner. Changes on the slider will be automatically forwarded to the textfield, and vice-versa.

The components in the GUI – `JSlider`, `JTextField` and `JLabel` – constitute multiple Clients to the `AddingBean`. The fact that they were all co-located within the single `AddingApplet`, and instantiated all at once, simplified the GUI side, but as far as

[6] `propertyChange()` methods often check the source of the event. Since we have controlled the registration process, this is source checking is unnecessary.

the JBCE is concerned, the Clients could be anywhere, and register at any time. This aids incremental GUI development, and opens up possibilities for distributed collaborative applications.

5 IMPLEMENTING JBCE FUNCTIONALITY

The `AddingBean`-*AddingApplet* example used the `javabeancalcengines` package to implement the JBCE functionality. We now discuss key aspects of this package, to see how the functionality is met.

Maintaining Clients

Since a JBCE is just a JavaBean, Clients are maintained through the support objects supplied by the JavaBean API for `PropertyChange` and `VetoableChange` events.

Calculation Engine Properties

A JBCE needs to maintain 3 boolean properties:

1. Being in equilibrium.
2. Automatic recalculation on/off.
3. Forcing updates.

These properties are used by subscribing Clients to distinguish user-initiated changes from those coming from the JBCE, or to control behaviour under those changes. The auto-recalculation property allows multiple JBCE Inputs to be changed before a calculation is undertaken.

Setting Inputs

The main services from the `javabeancalcengines` package, and `DefaultJavaBeanCalcEngine` in particular, are to do with setting Input properties. We saw this in `AddingBean` when setting `num1`, for which we used `setJavaBeanEngineInput()` –

```
public void setJavaBeanEngineInput(String inputProp, String
            fieldName, double newInput)
    throws PropertyVetoException {
  // Get the field corresponding the property.
  Field thePropField = getFieldKnownToExist(fieldName);

  // Check for external vetoes, then change locally.
  double oldInput = getKnownFieldValueDouble(thePropField);
  fireVetoableChange(inputProp,
            new Double(oldInput), new Double(newInput));
  setKnownFieldValueDouble(thePropField, newInput);
```

```
    // Recalculate from the new field value.
    if(isAutoRecalc()) {
      try {
        doRecalculate();
      } catch (Exception e) {
        // If we have a run-time error, we generically trap it and throw a
        // PropertyVetoException.
        // Clients can thus restore a safe value.

        setKnownFieldValueDouble(thePropField, oldInput);
        PropertyChangeEvent ePC =
               new PropertyChangeEvent(this, inputProp,
                     new Double(oldInput), new Double(newInput));
        throw new PropertyVetoException("doRecalculate", ePC);
      } // catch(Exception)
    } // if
    else {
      setNewInputNoRecalc();
    } // else

    // Can finally inform listeners of the change.
    fireForcingPropertyChange(inputProp,
               new Double(oldInput), new Double(newInput));
  } // setJavaBeanEngineInput(...)
```

The calls to `getFieldKnownToExist()`, `getKnownFieldValueDouble()` and `setKnownFieldValueDouble()` translate the external property `inputProp` into the internal field `fieldName`. These methods use introspection, so the target field named `fieldName` *must* be declared `public`. This goes against usual JavaBean practice, in which the fields would be declared as `private`, but it does allow the `set` method for a JBCE input to be written as a single call to `setJavaBeanEngineInput()`. If fields do need to be declared as `private`, then the functionality can be replicated manually.

We note the calling order of `fireVetoableChange()`, `doRecalculate()` and `fireForcingPropertyChange()`. This order gives Clients the opportunity to veto the proposed change, and the enclosed algorithm the chance to execute, before informing Clients that a change is to be made. As a result, when the change is actually made, Clients can act without having to allow for rollback.

The `try-catch` handler will trap exceptions from the internal algorithm, posting a `PropertyVetoException` that enables the calling Client to restore a safe value. Unfortunately, this presupposes the automatic calculation is active. The problem here is that, typically, automatic recalculation would be turned off if a set of inputs were to be supplied to the JBCE en masse, with the JBCE calculating on the new set as a whole. In this situation, if the algorithm encounters an error, it is difficult to know *which* of the inputs are problematic.

6 SUMMARY

This paper introduced *JavaBean Calculation Engines* for deploying Java computational code as reusable modules, ready for GUI integration. For the algorithm developer, JBCEs are a finishing point for deploying an algorithm as a "qualified product". For the GUI developer, JBCEs shield internal computations behind a JavaBean interface. JBCEs thus standardise the interaction between custom, computational code and external, GUI code.

The only requirement on the algorithm developer is to fully specify the Inputs to, and Outputs from, their algorithm. The Inputs and Outputs become properties of the JBCE, and the JBCE maintains equilibrium between Inputs and Outputs on the behalf of subscribing Clients. As the example showed, subsequent use of a JBCE follows patterns used in other Java GUI software, and this can aid evolutionary development of GUI products.

ACKNOWLEDGMENTS

The author thanks Matthew Phillips, Defence Science and Technology Organisation, and Don Weiss, Step 1 Inc, for their constructive comments and suggestions.

REFERENCES

Asbury, S. and Weiner, S.R. *Developing Java Enterprise Applications*. Wiley Computer Publishing, New York, 1999.

Doherty, D., Leinecker, R. et al. *JavaBeans Unleashed*. Sams Publishing, USA, 2000.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading Massachusetts, 1995.

Krueger, C.W. "Software Reuse". ACM Computing Surveys. Vol 24, Num 2, June 1992, pp 131-183.

Kühne, T. "The Function Object Pattern". C++ Report. Vol 9, Num 9, October 1997, pp 32-42.

Larman, C. "Implementing the Java Delegation Event Model and JavaBean Events in C++". *C++ Report*. Vol 11, Num 4, April 1999, pp 34-43.

Mili, H., Mili, F., Mili, A. "Reusing Software: Issues and Research Directions". IEEE Transactions on Software Engineering. Vol 21, Num 6, June 1995, pp 528-562.

Novak, G.S. “Software Reuse by Specialization of Generic Procedures through Views”. IEEE Transactions on Software Engineering. Vol 23, Num 7, July 1997, pp 401-417.

Schmidt, D.C. “How to Make Software Reuse Work for You”. C++ Report. Vol 11, Num 1, January 1999, pp 46-52,57.

Schmidt, D.C. “Wrapper Facade: A Structural Pattern for Encapsulated Functions within Classes”. C++ Report. Vol 11, Num 2, February 1999, pp 40-50.

About the author

Patrick Chisan Hew is an analyst with the Defence Systems Analysis Division of the Defence Science and Technology Organisation. His current work is in operational and strategic analysis, with an ongoing interest in software engineering/management. He holds a PhD in Mathematics and Intelligent Information Processing Systems from the University of Western Australia, the focus of which was in pattern recognition and image understanding. He can be reached at Patrick.Hew@dsto.defence.gov.au.