Streamlined Algorithm Deployment via JavaBeans

Patrick Chisan Hew

Theatre Operations Branch
Aeronautical and Maritime Research Laboratory
DSTO-TN-0335

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. The paper demonstrates JBCEs through a tutorial example, and discusses the use of JBCEs in a deployed product.

RELEASE LIMITATION

Approved for public release
Streamlined Algorithm Deployment via JavaBeans

Executive Summary

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.

Through JBCEs, DSTO could gain better value from its software development efforts, in the following three key ways:

1. Algorithm deployment in a form that encouraged transparency and reusability.
2. Having algorithm and GUI developers work in their areas of strength to a known, common, framework.
3. Preparing algorithms for use in collaborative, possibly distributed, environments.

JBCEs have already been of value to Theatre Operations Branch, in support of Task JNT 98/047 (Analysis of Australian Theatre Response Options). This paper discusses both tutorial and product examples of JBCEs, and the software supplied on the attached CD-ROM should enable deployment of new algorithms straightaway.
Authors

Patrick Chisan Hew
Theatre Operations Branch

Patrick Hew holds a PhD in Mathematics and Intelligent Information Processing Systems from The University of Western Australia, completed in 1999, the focus of which was in pattern recognition and image understanding. His present work is in operations analysis and software engineering.
List of Figures

Figure 1 Basic chain of events.
Figure 2 Adding Applet.
Figure 3 GUI communication with the AddingBean.
Figure 4 Screenshot of the SHuTTLE Applet.
Figure 5 Supplying distance to SHuTTLE through a map interface.
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOD</td>
<td>Air Operations Division.</td>
</tr>
<tr>
<td>CAT</td>
<td>Campaign Analysis Toolkit.</td>
</tr>
<tr>
<td>CE</td>
<td>Calculation Engine.</td>
</tr>
<tr>
<td>EXC3ITE</td>
<td>Experimental C3I Technology Environment.</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface.</td>
</tr>
<tr>
<td>JBCE</td>
<td>JavaBean Calculation Engine.</td>
</tr>
<tr>
<td>LOD</td>
<td>Land Operations Division.</td>
</tr>
<tr>
<td>MOD</td>
<td>Maritime Operations Division.</td>
</tr>
<tr>
<td>TOB</td>
<td>Theatre Operations Branch.</td>
</tr>
</tbody>
</table>
1. Introduction

It is increasingly attractive to use Java for software deployment. Unfortunately, the expertise for developing algorithms need not be coupled with an expertise in software engineering or GUI development, and this can result in software where the computational code is intertwined with code written 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.

Section 2 discusses the origins of this work, with Section 3 outlining the concept. Section 4 then works through a tutorial of deploying an algorithm into a JBCE, and using it in a GUI. We finish in Section 5 with a look at how JBCEs were used in a product. Supporting source code may be found on the attached CD-ROM.

2. Background

2.1 Motivation

This work arose out of the needs of Theatre Operations Branch (TOB) under Task JNT 98/047 (Analysis of Australian Theatre Response Options). This task provides a campaign plan analysis capability to the Australian Theatre through a series of studies of key campaign operations. An important activity under this task is the development of a Campaign Analysis Toolkit (CAT), to enable the rapid assessment of campaign plans.

CAT is envisaged as a “loosely-coupled” suite of software tools, independently developed with maximum reusability and commonality of data. The expertise in TOB is, however, in algorithm development, not software engineering; moreover, while the algorithms were envisaged as “qualified products”\(^1\) and thus stable, reusable assets [1], the GUIs were expected to evolve to meet client needs. The need, therefore, was for a “factory process” that packaged algorithms in a form enabling “black box” GUI development.

\(^1\) TOB expects to adopt work from component specialists – AOD, MOD, LOD …
2.2 JavaBeans

Object Oriented Programming has, at its heart, the construction, use, and reuse of software objects. Well-designed objects encapsulate custom functionality in a form that can be manipulated, customised, stored and called upon without having to know what is being done internally. The use of an object, however, requires an understanding of how external influences affect the internal workings, and how these effects feed back to the outside world.

A JavaBean is an object whose “face” to the outside world is standardised, through the JavaBean specification [2] [3]. This specification defines the manner in which the properties of the object are treated, in the following key respects:

1. The way in which we can access an object’s properties.
2. The way in which we are informed of changes to those properties.

The specification is straightforward: access to property \( xxx \) is provided through a method \( \text{get} \text{XXX}() \), change via a method \( \text{set} \text{XXX}() \), and when a property is changed, the JavaBean is expected to transmit a \text{PropertyChange} event to all registered listeners. JavaBeans also provide for the registration of listeners, and storage for later re-use.

The most visible examples of JavaBeans are GUI components, and we shall see that the use of a JBCE is similar to the use of GUI controls. As a result, the skill set for working with a JBCE may reasonably be expected from “mainstream” Java developers. By providing a generic procedure for matching algorithms to the JavaBean specification, we streamline their deployment.

3. Concept

This section introduces the model for Calculation Engines (CEs), and then indicates how JavaBeans are used to actually construct JBCEs.

3.1 Calculation Engines – Key Principles

The key principles underlying Calculation Engines 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.

2 There are variations for working with properties that are lists of items.
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.

The interaction between CE and Clients is basically the “Publish and Subscribe” Observer design pattern [4]. The point of this work, however, is in streamlining the construction to this pattern – setting up the “dashed lines” of Figure 1 for an arbitrary Input-Output Algorithm.

For software engineering, the abstract Input-Output Algorithm could be regarded as the “true” reusable element [5], with the CE being an artifact [6]. Input-Output form is useful in its own right [7], however the CE concept goes beyond the provision of high-level access to low-level code [8] [9] – it enables multiple Clients to “gather around” a problem. We shall see the benefits to GUI programming.

### 3.2 Using the JavaBean Framework

The JavaBean specification may be regarded as an implementation of the Observer design pattern [4], and this handles the communication with Clients. 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. Section 3.3 presents a tutorial example of using this package, both to deploy an algorithm and to integrate the resulting JBCE into a GUI. A technical discussion of the package is given in Appendix A, and the source code is on the attached CD-ROM.

Since Java can wrap native code, JBCEs can be used to wrap algorithms written in languages other than Java. The CE concept could also be applied in other Object Oriented Programming languages, so long as there was an equivalent to the PropertyChange mechanism [10].
3.3 Significance to DSTO

There are a number of ways in which JBCEs are of particular significance to DSTO:

1. **Algorithms as reusable assets.**
   
   “Re-inventing the wheel” is widely regarded as being undesirable. Unfortunately, the cost of adopting other work may be perceived as being greater than working from scratch. It is difficult enough to decide whether existing work is conceptually appropriate – having to “cannibalise” software compounds the situation\(^3\)\(^4\).
   
   While DSTO bodies should continue to work in their areas of strength, the increasing interest in joint operations points to algorithms being used away from their original designers. There are, therefore, advantages to DSTO in adopting procedures that encourage reuse from the start.

2. **Streamlined, incremental GUI development.**
   
   While modern analysis may require software development, it should be recognised that GUI programming is a skilled task. Unfortunately, the situation in DSTO sees researchers being pulled into GUI programming, a domain in which they may be ill-trained and under-equipped. Conversely, specialist programmers are hampered by ad hoc interaction between GUI and algorithm.
   
   The pay-off from JBCEs is that their construction rewards the algorithm developer by making their work accessible “like a GUI component”. JBCEs could thus encourage DSTO people to work in their areas of strength, to a common interface.

3. **Collaborative analysis tools.**
   
   The Campaign Analysis Toolkit\(^5\), Distributed Study Environment\(^6\) and Theatre Situation Awareness\(^7\) projects are just 3 examples in DSTO envisaging analysis tools being used in a collaborative, and possibly distributed, manner. Enabling this will, at its core, require that algorithms be packaged in a form that enables multiple entities to “gather around” an algorithm, like some form of “enhanced whiteboard”.
   
   The JavaBean technology underlying JBCEs handles multiple entities, and feeds forward to Enterprise JavaBeans for distributed software [12] [13]. Moreover, the “factory process” of producing a JBCE from an algorithm may be a useful asset to the EXC3ITE project.

---

\(^3\) The software implementation of an algorithm may include optimisations, simplifications and specialisations. Working backwards from software to algorithm could, therefore, lead to a false picture of what the algorithm does, and how it works.

\(^4\) It is worth noting that the TTCP “Guidance on Good Practice for Sharing Models” is “based on model sharing at the “whole-model” level; it does not address other ways of sharing software, for example … the establishment of common software libraries” [11, Paragraph 7].

\(^5\) The Campaign Analysis Toolkit is under Task JNT 98/047 (Analysis of Australian Theatre Response Options). Point of Contact is R Hughes.

\(^6\) The Distributed Study Environment is under Task ARM 00/047 (Distributed Study Environments). Point of Contact is R Nunes-Vaz.

\(^7\) The Theatre Situation Awareness project is under Task JNT 99/056 (Theatre Situation Awareness). Point of Contact is D Lambert.
4. Tutorial 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. The full source may be found on the attached CD-ROM.

![Adding Applet screenshot](image)

*Figure 2 Adding Applet.*

4.1 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 –

```java
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. For the sake of the exercise, however, we implement `addingalgorithm()` as –

```java
public static double addingalgorithm(double num1, double num2) {
    // Generally the computational code for an
    // algorithm will be much more complicated.
    double theReturn = num1 + num2;

    // The following represents an algorithm having an
    // internal problem that throws a runtime exception.
    if(num2 < 0) {
        double bombOut = 1/0;
    } // if

    return(theReturn);
} // addingalgorithm(...)
```

Generally, algorithm designers would not deliberately place “bombs” inside their algorithms. There may, however, be bugs in the code that could cause run-time errors. As we shall see, the JBCE functionality enables some recovery from problems.

### 4.2 Wrapping into a Calculation Engine

#### 4.2.1 Setting Up

A JBCE is a JavaBean, so we need to import this behaviour. In wrapping `addingalgorithm()` into a JBCE, `AddingBean`, we will adopt the functionality provided by `DefaultJavaBeanCalcEngine` –

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

public class AddingBean extends DefaultJavaBeanCalcEngine {
    public class AddingBean extends DefaultJavaBeanCalcEngine {

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

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

```java
/**
 * @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() from DefaultJavaBeanCalcEngine; to summarise details from Appendix A, the setJavaBeanEngineInput():
1. Provides Clients the opportunity to veto the impending change to Num1.
2. Calls addingalgorithm() to recalculate\(^8\).
3. Assigns\(^9\) Num1 the new value of newNum1, informing Clients of the change.

In making this summary, 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.

Parameter num2 is handled in a similar fashion.

\(^8\) If automatic recalculation is active. If not, the AddingBean notes that it is out of equilibrium.

\(^9\) For technical reasons discussed in Appendix A, the field Num1 must be declared public.
4.2.3 Output Property

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

```java
/**
 * @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.
 */
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.
 */
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.
4.2.4 Initialisation
When initialising the AddingBean, we must ensure that it is in equilibrium after being instantiated –

```java
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.

4.3 GUI Development
In developing the GUI for AddingApplet, we shall 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.

4.3.1 Obtaining a Calculation Engine
We start by having AddingApplet set up an AddingBean for use –

```java
... // 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 [12] [13].

4.3.2 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 simplicity and 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 the 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.**
4.3.3 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. Much of the functionality is common, and placed in the superclass doubleJSliderListener. This, in turn, inherits from LiveAddingAppletControl and DefaultLiveInputEffector, with the last being supplied in the javabeancalcengines package. We shall see how these inherited classes streamline the production of our slider listeners.

To listen to user input, we implement the ChangeListener interface, defining a stateChange() method -

```java
public void stateChanged(ChangeEvent evt) {
    if(!isExternallyForcedChange() & !isAlreadyTryingToReact()) {
        JSlider source = (JSlider) evt.getSource();
        double theNewValue = (double) source.getValue();

        // Do not check for source.valueIsAdjusting(), since the
        // AddingBean is fast enough to recalculate. If the calculations
        // took longer, we might need to check for a finally selected value.
        //
        //if(!source.getValueIsAdjusting()) {

        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(PropertyVetoException)

        stopReacting();
    } // if
} // if
} // stateChanged(ChangeEvent)
```
The listeners Num1Listener and Num2Listener provide the final link to the AddingBean by implementing setdoubleInput(); for Num1 -

```java
/**
 * 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() and note the comments regarding the test for source.valueIsAdjusting(), 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”. The DefaultLiveInputEffector superclass thus supplies 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. The LiveAddingAppletControl superclass thus supplies isExternallyForcedChange(), which inspects the AddingBean for isForcingUpdates(). With this in place, we can listen for changes coming from the AddingBean, by implementing the PropertyChangeListener and VetoableChangeListener interfaces –

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

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(PropertyChangeEvent)
```
It is worth emphasising the roles of these two methods. The AddingBean will call vetoableChange() first, providing the opportunity to veto a proposed change, and we see this in the tests for exceeding the bounds of the slider. The call to propertyChange() is made later, and at this point, the slider’s value can be set.

The final step is to instantiate the listeners and supply them to the sliders. This will be discussed in Section 4.3.6.

4.3.4 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\(^\text{10}\). The textfield is thus equipped with an actionPerformed() handler, implemented\(^\text{11}\) as below for Num1 –

```java
void jTextFieldNum1_actionPerformed(ActionEvent e) {
    String inputText = jTextFieldNum1.getText();
    try {
        double parsedValue = Double.parseDouble(inputText);
        getAddingBean().setNum1(parsedValue);
    } catch(Exception x) {
        // Something went wrong. Take value from Engine.
        jTextFieldNum1.setText(Double.toString(getAddingBean().getNum1()));
    } // catch(Exception)
} // jTextFieldNum1_actionPerformed(ActionEvent)
```

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

\(^{10}\) We note that until the user presses Enter, the textfield is out of synch from the AddingBean.

\(^{11}\) The implementation used Borland JBuilder Pro Ver3.00, so the code reflects the automatic generation idioms.
4.3.5 Farming Out Updates to Components

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

```java
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(...)
```
4.3.6 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 -

```java
// 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);
}
```

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\(^{12}\).

\(^{12}\) propertyChange() methods often check the source of the event. Since we have controlled the registration process, this source checking is unnecessary.
4.4 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.

We recall from Section 4.1 that the algorithm had a “bomb” for \( \text{Num2} < 0 \). The GUI knows nothing about it; however, it has inherently written to recover from problems in execution. Hence, when using the slider for \( \text{Num2} \), the user is unable to make \( \text{Num2} < 0 \).

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 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, using Java RMI for example [13].

5. Deployed Example

To see how JBCEs can streamline algorithm deployment, we look at a deployed product. Our choice is SHuTTLE, developed for the Campaign Analysis Toolkit to support the analysis of amphibious operations. The SHuTTLE Algorithm [14] was deployed in a JBCE, and has been used in the SHuTTLE Applet [15] shown in Figure 4.
5.1 *SHuTTLER JBCE*

The key point in constructing the *SHuTTLER JBCE* was in specifying its Input-Output algorithm. The resulting method header is given below -

```java
public static SHuTTLEReturn shuttlealgorithm(
    double distShipToLZOneWay_km,
    int troopsTotalAllShips,
    double[][] heliAliveProfiles,  
    double heliFlightSpeed_kmperhr,  
    int troopsPerHeliFlight,  
    double pilotFlyingTime_hr,  
    double pilotRestTime_hr,  
    double timeRefuelHeli_hr,  
    double timeTroopsLoad_hr,  
    double timeTroopsUnload_hr,  
    double fuelHeliCapacity_kg,  
    double fuelBurnFlightSpeed_kgperhr,  
    double fuelBurnIdling_kgperTrip,  
    double evolveUpperLimit_hr
)
```

where the return object *SHuTTLEReturn* has the following fields -

```java
public class SHuTTLEReturn {
    public boolean isHeliSchedulingRequired;
    public double evolveTimeStep_hr;
    public double timeTransit_hr;
    public double fuelBurnPerTrip_L;
    public int tripsPerRefuel;
    public double [] evolveRawFlyingTimes;
    public double [] evolveRefuelModTimes;
    public double [] evolvePilotModTimes;
    public double [] evolveRealTimes;
    public int [] evolveTroopsAtLZ;
    public int [] evolveNumHeliAlive;
    public double [] timesTroopsLoaded;
    public double [] timesTroopsDropped;
    public double [] timesHeliRefuel;
} // SHuTTLEReturn
```

As can be seen, the inputs to and outputs from the algorithm range over both primitive and complex types\(^\text{13}\). The principles of the JBCE still hold, however - each input parameter becomes a property of the *SHuTTLER JBCE* with a *get* and *set* method, and the output becomes a property with a *get* method.

\(^{13}\) In retrospect, the author would choose to use a *Vector of Doubles* rather than a `double [][]` - it is difficult to correctly implement the *set* methods in the latter case.
5.2 SHuTTLE Applet

The design of the SHuTTLE Applet follows from the inputs that it collects, and the results that are to be displayed. The top and left portions take user input, and the results are displayed graphically at centre. We note from that we are viewing 2 different scenarios\(^{14}\), named “Alpha” and “Bravo” and displayed in Blue and Green respectively. This shows how a JBCE encapsulates the “solution” to an algorithm – each of the 2 scenarios has its own SHuTTLE JBCE, with the user choosing the “active” one through the radio buttons.

The slider controls at the top are “live”, with functionality similar to those discussed in Section 4.3.3 for the AddingApplet; the SHuTTLE algorithm being fast enough for this “live” interaction to be feasible. The buttons at top-right link to dialogs that take more inputs, which are then supplied to the SHuTTLE JBCEs when the user presses “OK”; in terms of the interaction with the JBCEs, this is conceptually equivalent to the process used in Section 4.3.4 for the AddingApplet.

The development of the SHuTTLE Applet underwent considerable evolution, with the GUI being refined from use. The underlying SHuTTLE algorithm was, however, relatively mature, so it was important to ensure that changes to the GUI did not compromise the computational code. The encapsulation of SHuTTLE in a JBCE made this straightforward.

5.3 Support for Incremental GUI Development

The first parameter to the SHuTTLE algorithm, distShipToLZOneWay_km, is the distance from the amphibious ships to the helicopter landing zone. In the SHuTTLE applet, the user specifies the value through a slider control. It is highly conceivable that this might be supplied through a map interface, as illustrated in Figure 5, with the distance being calculated and supplied to the SHuTTLE JBCE.

\[\text{setDistShipToLZOneWay_km()}\]

\[\text{propertyChange()}\]

Figure 5 Supplying distance to SHuTTLE through a map interface.

\(^{14}\) The software structure can handle an arbitrary number of scenarios, however, as a concept demonstration, the implementation was simplified to the fixed number of 2.
The `setDistShipToLZOneWay_km()` method uses `setJavaBeanEngineInput()`, just as was discussed for the `AddingBean` in Section 4.2.2. As a result, the `SHuTTLE` J BCE ensures that, if such a Map Client were built, then it would be automatically integrated with the existing `SHuTTLE Applet`. We note, furthermore, that, no changes would be required to the `SHuTTLE Applet`; indeed, all the `SHuTTLE Applet` sees are `propertyChange` events, which it already knows how to handle. We thus see how J BCEs support the incremental and evolutionary development of GUIs.

6. Summary

This paper introduced `JavaBean Calculation Engines` for deploying Java computational code as reusable modules, ready for GUI integration. For the algorithm developer, J BCEs are a finishing point for deploying an algorithm as a “qualified product”. For the GUI developer, J BCEs shield internal computations behind a JavaBean interface. J BCEs 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 J BCE, and the J BCE maintains equilibrium between Inputs and Outputs on the behalf of subscribing Clients. As the tutorial example showed, subsequent use of a J BCE follows patterns used in other Java GUI software, and as seen in the deployed example, this aided the evolutionary development of a complex GUI product. Support for J BCE construction is provided by the `javabeancalcengines` package, available on the attached CD-ROM.

Acknowledgements

The author thanks Matthew Phillips, DSTO and Don Weiss, Step 1 Inc, for their constructive comments and suggestions.
Appendix A: JavaBean Calculation Engines Package

A.1. JavaBeanCalcEngine

The JavaBeanCalcEngine interface defines services that are required for JBCEs, namely support for the following:
1. The JBCE being in equilibrium.
2. Automatic recalculation being active/inactive.
3. The JBCE forcing updates.
4. Registration of Clients for PropertyChange and VetoableChange events.

A.2. DefaultJavaBeanCalcEngine

The DefaultJavaBeanCalcEngine class provides a default implementation of the JavaBeanCalcEngine interface, and utilities for writing set and get methods. Most JBCEs could thus be written by extending DefaultJavaBeanCalcEngine.

The JBCE properties of being in equilibrium, auto-recalculation and forcing updates are handled through boolean flags as bound properties. Registration of Clients uses the support objects supplied by the JavaBean API for PropertyChange and VetoableChange events.
The main custom behaviour is in the `setJavaBeanEngineInput()` methods. As introduced in Section 4.2.2, this method performs a number of functions -

```java
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(String, String, double)
```

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 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.

**A.3. DefaultLiveInputEffector**

GUI development, while necessarily crafted to the problem at hand, does tend to follow design patterns. The `DefaultLiveInputEffector` class provides two services towards this end:

1. Support for already reacting to user input.
2. A hook for distinguishing user changes from those coming from the JBCE.
References

# Streamlined Algorithm Deployment via JavaBeans

## 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. The paper demonstrates JBCEs through a tutorial example, and discusses the use of JBCEs in a deployed product.