The SwapBox: A Test Container and a Framework for Hot-swappable JavaBeans

L. Tan, B. Esfandiari, B. Pagurek
Carleton University
Ottawa, Ontario, Canada K1S 5B6
babak@sce.carleton.ca

1. Introduction

Software hot swapping refers to the process in which a part of a running software application is replaced by a new version of the program during runtime, mainly for maintenance, upgrade and big-fixing reasons. While most software systems can be effectively shut down and upgraded off-line, certain safety and mission critical applications such as in telecommunications and air traffic control cannot afford any such down time.

Several approaches to hot swapping, or runtime software evolution (reconfiguration) in other word, have been proposed in the literature [1, 2, 3, 4, 6, 7, 8, 9].

This work builds on earlier research [1, 2, 10] in which the main issues in software hot-swapping were identified and tackled by decomposing the application in an arbitrary set of swappable modules called s-modules. The hot-swapping process was then executed by a "swap manager", which took care of state transfer between the old and the new s-module. The *referential transparency problem* [10] was solved through the use of the proxy pattern, but this meant that proxies had to be provided for each s-module.

In this paper we study the benefits provided by the use of standard component technologies such as JavaBeans to simplify and manage the hot-swapping process. Our contribution here is twofold:

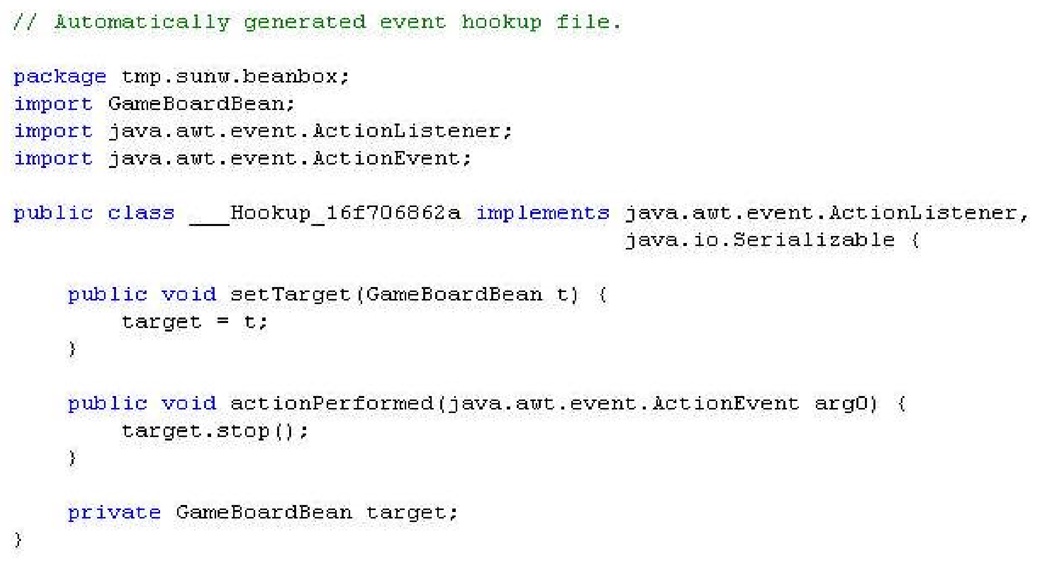
- we describe SwapBox, a test container for hot-swappable JavaBeans. The SwapBox is an extension of the BeanBox provided in the Bean Development Kit (BDK) by Sun Microsystems (http://java.sun.com/beans). We chose to extend the BeanBox because of the availability and the simplicity of its source code.
- we describe one of the two distinct JavaBean hot-swapping processes which are embedded in the SwapBox as reference implementations. By design, the SwapBox can be used as a framework to try out alternative hot-swapping solutions.

This paper is organized as follows: we first review the design of the original BeanBox. We then introduce the SwapBox and describe its design. Following that we show how we tackle the different hot-swapping issues in the context of the SwapBox.

2. Overview of the BeanBox

The BeanBox is developed by Sun Microsystems as a test container for Java Beans. Users are able to load JavaBean components, drag and drop them into a panel called the BeanBox, and finally configure them and hook them up with other JavaBeans. Furthermore, users are able to serialize a configured bean into a file, and recover it into memory later. The property sheet allows users to modify beans properties at runtime.

JavaBeans are event driven. In applications composed in the BeanBox, a bean has no direct reference to beans with which it interacts, since a bean cannot know in advance which other beans it will interact with. The BeanBox automatically generates, compiles and loads an adapter class on the fly, which is inserted between a source bean and the target bean. The source bean only knows the reference to the event adapter, and the event adapter in turn knows the reference to the target bean. When the source bean attempts to invoke a method at the target bean, it actually invokes the same signature method at the event adapter, which in turn invokes the method at the target bean. Because the "glue code" (event adapter) is created dynamically, it can defer the establishment of interactions between beans to the last minute, and creates the potential to replace a bean at runtime. The following figure shows an example of an automatically generated event hook up code:

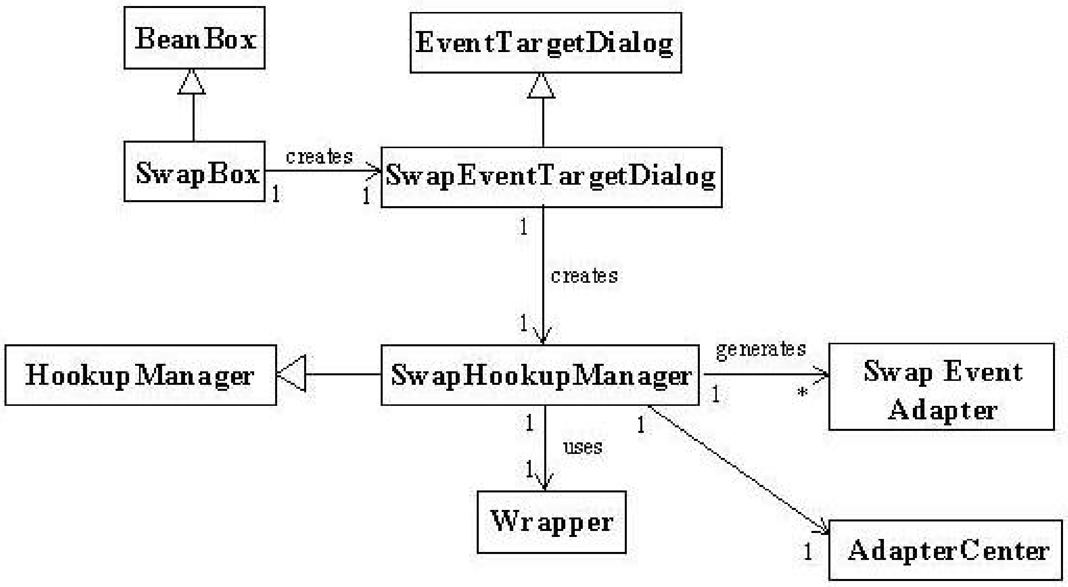


```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import GameBoardBean;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ___Hookup_16f706862a implements java.awt.event.ActionListener,
                                             java.io.Serializable {

    public void setTarget(GameBoardBean t) {
        target = t;
    }

    public void actionPerformed(java.awt.event.ActionEvent arg0) {
        target.stop();
    }

    private GameBoardBean target;
}
```

Three classes, BeanBox, EventTargetDialog, and HookupManager are involved in wiring beans. BeanBox enables the selection of the source bean and the target bean, as well as the outgoing event. The EventTargetDialog enables the selection of the target method. HookupManager is used to automatically generate the adapter, compile the adapter, and load it into the memory. In the next section we will show how we use and extend these classes to allow component hot-swapping.

3. Design and Implementation of the SwapBox

The SwapBox follows the event-based communication mechanism used in the BeanBox. However, there are major modifications to the role of the event adapter, which is no longer a simple “relay station” for method invocation. Also, the SwapBox provides a facility to make up mapping rules at runtime, which allow to “map” the state of the old bean to the state of the new bean, in case the beans in question do not follow the same interface. The following figure shows a class diagram of the main pieces of the SwapBox:



3.1 The AdapterCenter

In order to swap out an old bean, the SwapBox should switch event adapters attached to the old bean to the new bean. For this reason, the SwapBox should be able to know what event adapters are attached to a particular bean. The BeanBox does not provide such facility. Therefore we provide an AdapterCenter class in the SwapBox. The AdapterCenter is used to store information about event adapters. For each event adapter, it stores a SwapEventInfo object which contains the source bean, target bean, the outgoing event, the target method, and a reference to the adapter. All these information are needed when a swap transaction is requested by the SwapManager.
During a swap transaction the AdapterCenter is consulted, and if the following conditions are NOT met:

- The set of events fired by the old bean A is the subset of the events fired by A’;
- The set of methods at A invoked by other beans via event delivery is the subset of methods provided at A’;

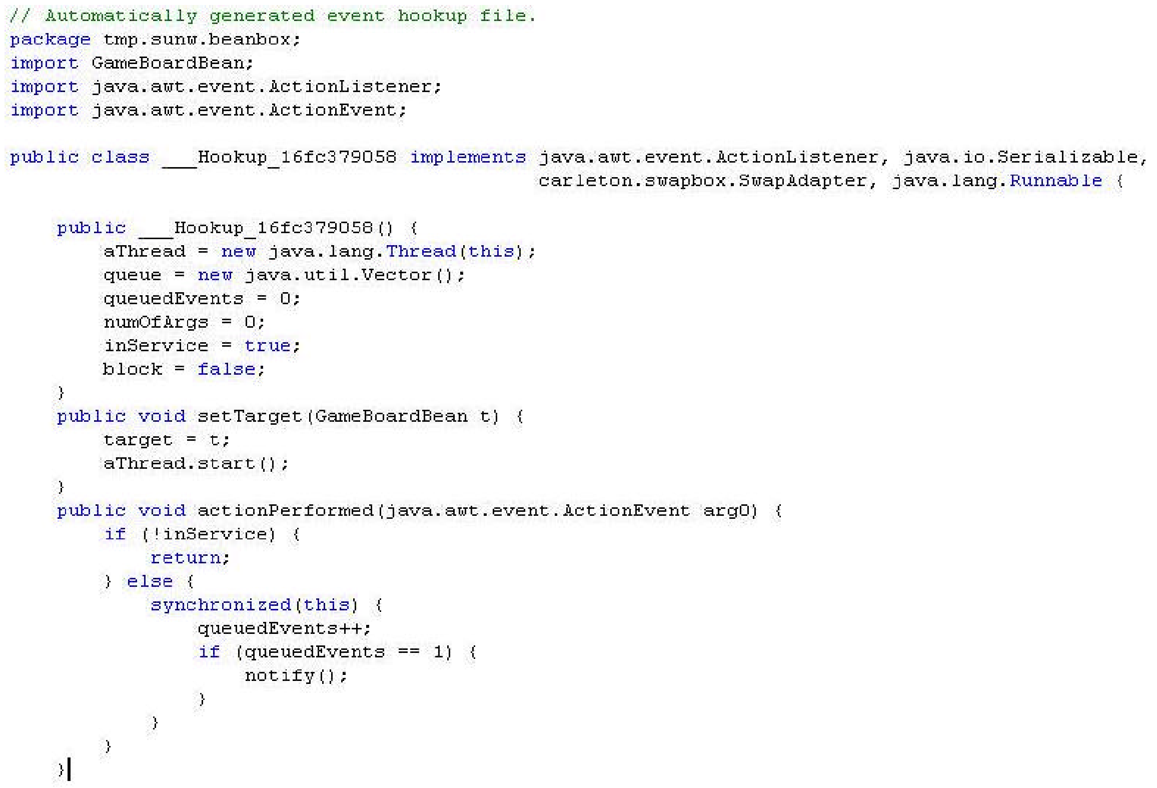
then an exception will occur to signal the potential discrepancy.

Many ways could be thought of in dealing with such an exception:

- the concerned beans could be consulted to find out whether the discrepancy is acceptable or should the swap transaction be aborted. However this requires the beans to have some extra knowledge;
- the user of the SwapBox could be prompted to either proceed with or abort the transaction, knowing that proceeding with the transaction could potentially lead to a partial or total disconnection between two beans!

3.2 The SwapEventAdapter

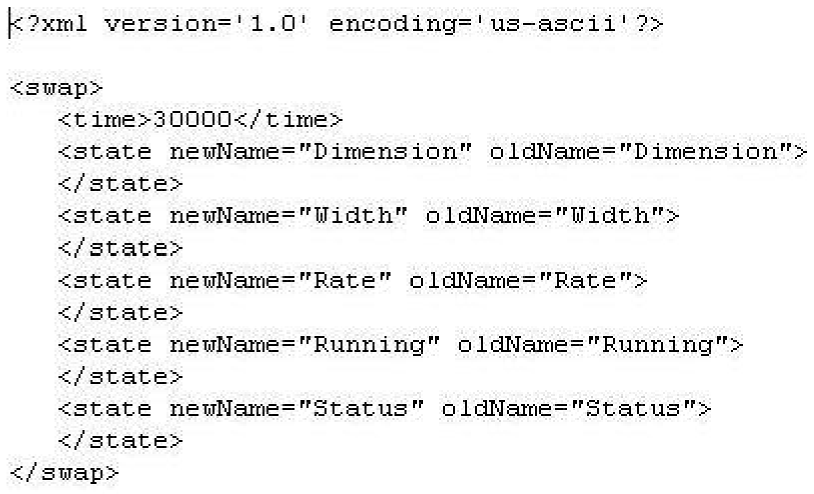
The fact that the adapter generated by the BeanBox has a setTarget method is a good premise for hot-swapping. However, there is a need to block the service requests and queue incoming events during the swap transaction ([1] discusses the conditions to be met for a component to be ready to be swapped out). The following figure shows part of the adapter code generated by the SwapBox:



```
// Automatically generated event hookup file.
package tmp.sunw.beanbox;
import GameBoardBean;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ___Hookup_16fc379058 implements java.awt.event.ActionListener, java.io.Serializable,
                                         carleton.swapbox.SwapAdapter, java.lang.Runnable {

    public ___Hookup_16fc379058() {
        aThread = new java.lang.Thread(this);
        queue = new java.util.Vector();
        queuedEvents = 0;
        numOfArgs = 0;
        inService = true;
        block = false;
    }
    public void setTarget(GameBoardBean t) {
        target = t;
        aThread.start();
    }
    public void actionPerformed(java.awt.event.ActionEvent arg0) {
        if (!inService) {
            return;
        } else {
            synchronized(this) {
                queuedEvents++;
                if (queuedEvents == 1) {
                    notify();
                }
            }
        }
    }
```

3.3 State Transfer

In order to transfer the state of an old bean to the new bean, an intuitive and elegant approach would be to rely of the serialization mechanism: the SwapBox would save the old bean's state by serializing it, and recover it by unserializing it into the new bean. This way, the responsibility to determine what constitutes the state of a bean would be delegated to the bean itself. Unfortunately, Serialization has currently some shortcomings: for instance, one cannot unserialize into a bean with a different class name. Also, the interface of the new bean might be different from the old one. Therefore there is a need to specify some mapping rules between the two interfaces. We have decided that such mapping rules should be specified by the user right before the swap transaction takes place: the SwapBox analyses the old bean and the new bean's properties using reflection and displays them to the user so that he/she can map the properties of the old one to the properties of the new one. The user is also prompted to specify a time out value for the swap transaction. The mapping rules are then saved in an XML document. In the figure below, the old bean and the new bean have identical properties but that need not be the case:



```
<?xml version='1.0' encoding='us-ascii'?>

<swap>
   <time>30000</time>
   <state newName="Dimension" oldName="Dimension">
   </state>
   <state newName="Width" oldName="Width">
   </state>
   <state newName="Rate" oldName="Rate">
   </state>
   <state newName="Running" oldName="Running">
   </state>
   <state newName="Status" oldName="Status">
   </state>
</swap>
```

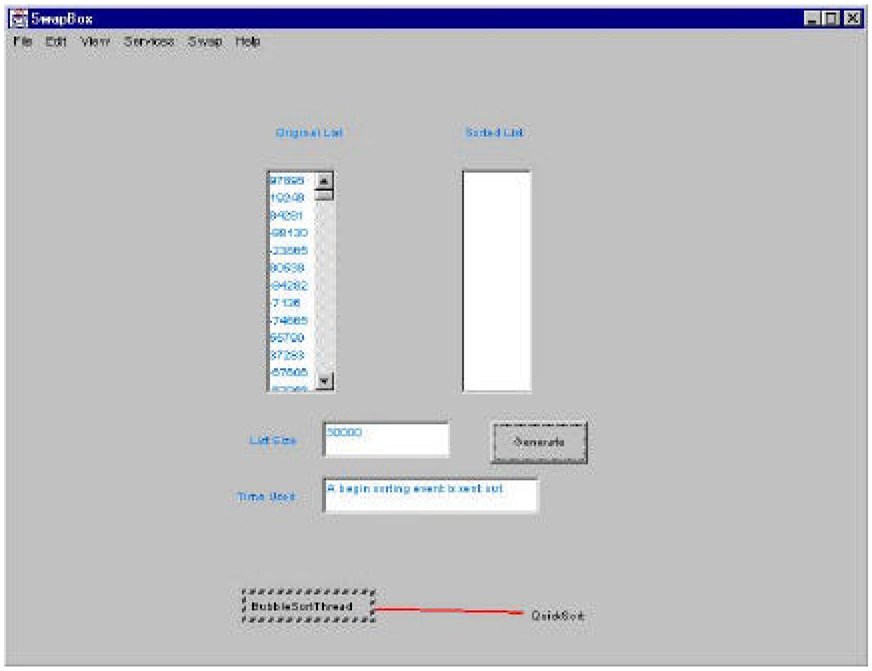
The values of the properties are then accessed using the getter methods of the old bean, and set in the new bean via its setter methods. Sun is currently working on an XML-based alternative to serialization, which would in particular remove the class name constraint. We are planning to use such a feature so that we can truly rely on the beans to decide what their state should consist in. The mapping rules between the XML files could then be stored in a XSLT stylesheet instead of using our current custom tags.
The SwapBox is not necessarily where the real swap transaction should take place. Instead, the SwapBox can be used to simulate and test whether the swap transaction is feasible and quick enough for given beans. Therefore, one of the benefits of storing the mapping rules in an XML file is that once the rules are created, they can be shipped together with the new bean to the location where the actual transaction should take place.

3.4 Putting it all together: the SwapManager

When a swap transaction is requested, the SwapManger comes into play. Its main responsibilities include:

- Set a timeout for the swap transaction
- Parse the swap transaction file to retrieve parameters specific to the particular transaction
- Create event adapters for the new bean
- According to the states in the old bean, create new states
- Clean up the old bean in case of successful transaction
- Roll back to the old bean in case of exception or timeout

Our swap transaction technique is simple. We create a set of event adapters for the new bean at first. In this stage, the old bean behaves as usual. Then we block services for the old bean and transfer the states from the old bean to the new bean. If no exceptions occur, we swap the old bean with new bean and remove the old bean from the SwapBox. Otherwise, the event adapters created for the new bean will be removed and the old bean continues its service. The following picture is a snapshot of the SwapBox at work, where a BubbleSort bean is being replaced with a QuickSort bean at runtime:



97895
10248
84281
-98130
-23585
80638
-94282
-7136
-74665
55790
87283
-57505

50000

A begin sorting event is sent out

4. Conclusion and Future Work

The swapping process described in this paper is one of the two methods that have been integrated in the SwapBox. Indeed, we want the SwapBox to also be a framework to help evaluate the merits of various techniques. We are planning to apply our techniques to the telecommunications domain. In the meantime, the SwapBox source code is available for evaluation and feedback, and the reader is welcome to contact the authors to access it.

Reference:

[1] G. Ao, Software *Hot-Swapping Techniques for upgrading Mission Critical Applications On the Fly*, Masters Thesis, System and Computer Engineering Department, Carleton University, February 2000

[2] N. Feng, *S-Module Design for Software Hot Swapping*, Masters Thesis, System and Computer Engineering Department, Carleton University, September 1999

[3] M.M. Gorlick, R.R. Razouk, *Using Weaves for Software Construction and Analysis*, Proceedings of the 13th International Conference on Software Engineering, IEEE Computer Society Press, May 1991

[4] D.Gupta, P.Jalote, G. Barua. *A formal framework for on-line software version change*. IEEE Transactions on Software Engineering, vol 22, no 2, February 1996

[5] J. Hopkins, *Component Primer*, Communications of ACM, Vol 43, No. 10, October 2000

[6] J.Kramer, J. Magee, *The Evolving Philosophers Problem: Dynamic Change Management*, IEEE Transactions on Software Engineering, Vol 16, No 11, November 1990

[7] P. Oreizy, N. Medvidovic, R. N. Taylor, *Architecture-Based Runtime Software Evolution*, Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), pages 177-186, Kyoto, Japan, April 19-25, 1998.

[8] J. Peterson, P.Hudak, G.S. Ling, Principled Dynamic Code improvement, Yale University Research Report YALEU/DCS/RR-1135, Department of Computer Science, Yale University, July 1997

[9] D. B. Stewart, R. A. Volpe, P. K. Khosla, *Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Object*, IEEE Transactions on Software Engineering, Vol. 23, No. 12, December 1997

[10] Feng, N., Gang, A., White, T., and Pagurek, B. Dynamic Evolution of Network Management Software by Software Hot-Swapping. Accepted for publication at the Seventh IFIP/IEEE International Symposium on Integrated Network Management (IM 2001), Seattle, May 14-18, 2001.