**Adaptation and Software Architecture**

WPI-CS-TR-98-13
George T. Heineman
Worcester Polytechnic Institute
Worcester, MA 01609

http://www.cs.wpi.edu/~heineman

# Abstract

This paper focuses on the need to adapt software components and software architectures. Too often, examples from the literature avoid the difficult problem of constructing a software system from sets of (possibly independently constructed) software components. Consider the following example: given a set of components, an application builder attempts to construct a software system. Along the way, however, some components are adapted with new code to suit the requirements of the final application. If the components have architectural specifications, how are these adaptations reflected? In particular, changes in the component implementation (i.e., new classes, modified methods, new lines of code) must be reflected in the specification of the component (i.e., ports, roles). This problem is more important than simply keeping documentation synchronized with software -- the architectural description of a software system is essential in understanding the interconnections between the various software components.

# 1 Introduction

As the size of software applications increases, it becomes infeasible to implement software systems from scratch. Software developers are responding to this growing complexity by constructing software systems based on software components. However, it is still an elusive goal to construct applications entirely from pre-existing, independently developed software components. One aim of software architecture research is to better specify the high-level design and overall structure of a software system, focusing on the individual components and their interconnections [GS93]. An Architectural Description Language (ADL) provides the syntax for describing a software architecture. As software developers construct systems from components, they can describe the developing software architecture using an ADL specification. This position paper addresses two issues. (1) Mapping programming-level constructs such as methods and classes to ADL constructs such as Ports and Roles. (2) Modifying ADL descriptions in response to component-specific adaptations.

## 1.1 Context

We explore these issues by investigating the JavaBeans [SUN97] component model. A Java *Bean* is a reusable software component that can be manipulated visually in a design environment, such as the sample Bean Development Kit (BDK). A Bean is defined by its *Properties*, *Methods*, and *Events*. JavaBeans relies on an implicit naming scheme of its public methods; for example, if a Bean has public methods `setHeight (int h)` and `int getHeight()`, one can infer that the Bean has a property (i.e., a named state attribute) *Height*. All public methods of a Bean are made available for other Beans to invoke as needed. Finally, Beans communicate with one another through events that allow components to propagate event notifications to one or more registered listeners. If a Bean has methods `void addControlEventListener (ControlEventListener cel)` and `void removeControlEventListener (ControlEventListener cel)` then one can infer that the Bean is capable of generating *ControlEvent* events.

The simplicity of JavaBeans is offset by the drawbacks of such an implicit approach to component construction. First, because there is no explicit specification of the interactions between the component

and its environment, the standard ADL concepts of Connectors, Ports, and Roles are implicit in JavaBeans. Second, interoperability between JavaBeans and other non-Bean Java components is made more difficult. Third, because a Bean interface is simply a collection of methods, the interface of complex Beans is often a confused list of (seemingly arbitrary) methods. To overcome these drawbacks, we have concretely defined the various Port and Role types that are possible in JavaBeans [OH98]. In doing so, we uncovered some problems in mapping programming-level constructs such as methods to standard ADL constructs such as Ports.

## 1.2 Motivating Example

Consider the simple Spreadsheet application in Figure 1 composed of nine interacting Beans. A TableBean *tb* displays a matrix of information with C columns and R rows. The column header TableBean *tbC* has height of 1 and width of C. The row header TableBean *tbR* has width of 1 and height of R. A status TableBean *tbBox* (showing C8 in Figure 1) has height and width of 1. There are two ScrollbarBeans, one vertical (*vs*) and one horizontal (*hs*), that allow users to select values from within a particular range. A TextBean *textb* allows users to enter text. A List Bean *selectb* allows users to switch between sheets, or create new ones. Lastly, an invisible SpreadsheetBean *ss* maintains and calculates all values in the spreadsheet, of which only a few are visible as determined by *tb*. A Java applet *app* creates these Beans and registers the interactions between them.

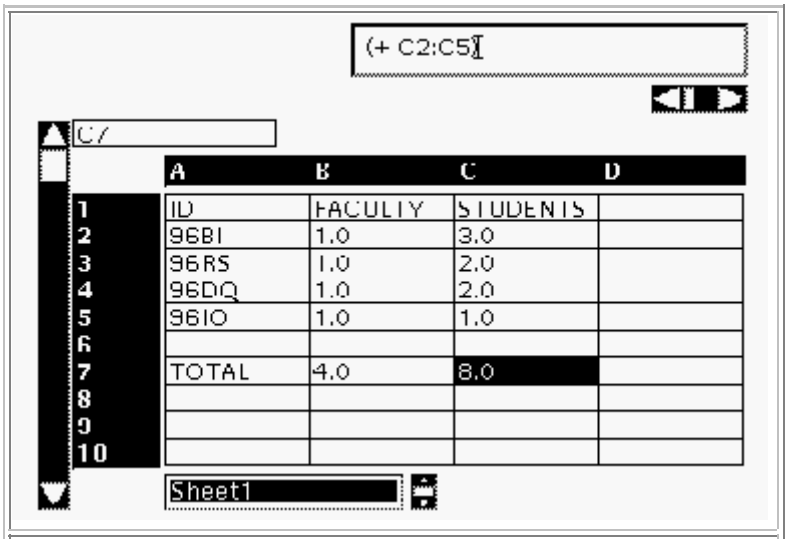


**Figure 1. JavaBeans Spreadsheet Application**

The components react to GUI events (i.e., mouse clicks) and communicate with each other using events. For example, when the user selects an entry in *tb* using the mouse, *tb* generates a `TableEventObject` event. *app* processes this event by setting entry (1,1) for *tbBox* to the designated cell while the contents of the spreadsheet cell (i.e., `(+ C2:C5)`) are shown in *textb*. This type of integrated collaboration is typical of component-based software systems.

We have defined a Component Specification Language (CSL) [Hein97] for specifying components. This language is based on ACME [GMW96], the standard interchange language for ADLs. The specification in Figure 2 was automatically generated by a *Classifier* tool we have written that uses the Java reflection model to inspect a component's methods. Note that there is a separate Port for each Java *interface* implemented by the component. The Classifier also detects all properties and events by reviewing the names of all public methods for the Bean. Finally, there is a special port *publicInterface* that groups all methods that are not based on the JavaBeans implicit naming scheme.

```
component adapt.gui.TableBean extends java.awt.Component {
  property java.lang.String         TableValue (int, int);
  property int                      RowHeight;
  property int                      ColumnWidth;
  property java.awt.Color           Foreground;
  property java.awt.Color           Background;
  property java.awt.Font            Font;
  property adapt.gui.TableRegion    SelectedRegion;
  property int                      NumColumns;
  property int                      NumRows;
  property adapt.ComponentAdapter   Adapter;

  listener TableListener;
  listener ActionListener;
  listener PropertyChangeListener;

  Port adapt.gui.TableListener = {
    void handleTableEvent (adapt.gui.TableEventObject);
  }

  Port java.io.Serializable = { }
  Port java.awt.event.MouseMotionListener = {
    void mouseDragged (java.awt.event.MouseEvent);
    void mouseMoved   (java.awt.event.MouseEvent);
  }
  Port java.awt.event.MouseListener = {
    void mouseClicked  (java.awt.event.MouseEvent);
    void mousePressed  (java.awt.event.MouseEvent);
    void mouseReleased (java.awt.event.MouseEvent);
    void mouseEntered  (java.awt.event.MouseEvent);
    void mouseExited   (java.awt.event.MouseEvent);
  }

  Port publicInterface = {
    void                paint  (java.awt.Graphics);
    void                redraw (java.awt.Graphics, int, int);
    boolean             select (adapt.gui.TableRegion);
    boolean             select (adapt.gui.TableElement, adapt.gui.TableElement);
    java.awt.Dimension  getPreferredSize ();
  }
}
```

**Figure 2. CSL specification of TableBean**

# 2 Adaptation

We make a distinction between software *evolution*, where the software component is modified by the component designer, and *adaptation*, where an application builder adapts the component for a different use. We also differentiate adaption from *customization*; an end-user customizes a software component by choosing from a fixed set of options (such as OIA/D [Kicz97]). An end-user adapts a software component by writing new code to alter existing functionality.

If the source code for a software component is available, a software developer can theoretically make any change to the code. Typically, however, programmers tread cautiously when modifying third party code, making slight extensions or carefully modifying a small number of lines. Ideally, a software component should be designed to allow easy extension and adaptation. We consider two situations when adaptation can occur: either the component provides a mechanism for adaptation or the source code is available. Our ADAPT framework [Hein97] defines a style for adaptable software components using *active interfaces*. There are two phases to all interface requests: the "before-phase" occurs before the component performs any steps towards executing the request; the "after-phase" occurs when the component has completed all

execution steps for the request. These phases are similar to the Lisp advice facility described in [Rama97]. A standard way to alter the behavior of a component is to interpose an entity to intercept messages and/or events. Because such adaptation is likely to occur, the component should provide an interface for this purpose.

Consider the use of the TableBean components *tbR* and *tbC*. The application designer would like to reuse the basic functionality of the TableBean component, but *tbR* and *tbC* behave differently from *tb*. In particular, the values can be calculated and the standard `setTableValue()` interface can be bypassed. The TableBean component has been designed to allow "before-phase" methods to be registered for the `getTableValue (int col, int row)` method. As shown in Figure 3, the application developer provides extra code that will be invoked before the `getTableValue` request is processed. For *tbR*, as shown in Figure 4, the developer inserts special code to calculate the actual row number based upon the current topRow, a property maintained by the new *Glue5* class.

```
component tbR adapts TableBean {
  Port   Glue5.topRow  = { ... };
  action Glue5.retValue (int);

  String getTableValue (int col, int row) {
    before Glue5.retValue (row);
  };
}
```

**Figure 3. CSL specification of adaptation**

```
public String retValue (Integer colI, Integer rowI) {
  int col = colI.intValue();
  int row = rowI.intValue();

  int actualRow = row + topRow - 1;
  return new String (new Integer (actualRow).toString());
}
```

**Figure 4. Adaptation code inserted before `getTableValue()` for *tbR***

The TableBean component allows per-instance adaptations, so both *tbC* and *tbR* have special code that adapt the Bean for both specialized uses. This is possible only because the designer of TableBean has foreseen that other developers may use the Bean in unanticipated ways. The benefit for the component designer is that the Bean can be used by a wider range of developers, and the benefit for the application builder is easy adaptation of existing code without source code modification. Our architectural approach to component adaptation maintains a clean separation between the original component and all future adaptations of it, allowing components to be released without source code and yet still allow adaptation.

The Connectors and Roles have similar structure that we briefly present in Figure 5. This connector is responsible for integrating the various Beans to process events occuring in the TableBean. In Figure 1, for example, when the user clicks on the cell C7, *tbBox* is updated to contain the cell reference, *textb* is updated to contain the cell contents, and *tbR* and *tbC* are updated to unselect any regions (i.e., entire columns or rows) that were previously selected). The connector that accomplishes these tasks is specified in Figure 5.

```
Connector handleTBEvent:BeanConnector composedof TextRole extended with {
  Role consumeTableBeanEvent : ConsumeEvent;

  Role getSelectedRegion_tb  : GetProperty;
  Role getTableValue_tbR     : GetProperty;
  Role getTableValue_tbC     : GetProperty;
  Role getValue              : GetProperty;
  Role getLeftColumn         : GetProperty;
  Role getTopRow             : GetProperty;

  Role setTableValue         : SetProperty;
}

Attachments {
  tbR.Leftcolumn.getLeftColumn         to  handleTBEvent.getLeftColumn;
  tbC.TopRow.getTopRow                 to  handleTBEvent.getTopRow;
  tbC.SelectedRegion.setSelectedRegion to  handleTBEvent.setSelectedRegion;
  tbR.SelectedRegion.setSelectedRegion to  handleTBEvent.setSelectedRegion;
  tbBox.TableValue.setTableValue       to  handleTBEvent.setTableValue;
  ss.Value.getValue                    to  handleTBEvent.getValue;

  textBean.Text                        to handleTBEvent.RoleText;
}
```

**Figure 5. Connector Description**

# 3 Conclusion

In this short paper, there is not enough room to discuss how Roles are also defined using CSL. We are making progress, however, at capturing the complexity inherent in the interface of software components and showing how the specification can reflect adaptations to the components. In [OH98] we defined three possible views of a component's interface -- informational, behavioral, and functional. Our initial findings support the idea that composition and inheritance should be more widely used in defining Ports and Roles. The essential understanding we reached is that **the interface of a Port clearly defines how the Role for a Connector should be attached.** More work now needs to be performed on how useful these complex specifications are when adapting software components to build software systems. This paper is based on work sponsored in part by National Science Foundation grant CCR-9733660.

1. **[GS93]** David Garlan and Mary Shaw. *An Introduction to Software Architecture*, volume I of *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, New Jersey, 1993.
2. **[GMW96]** *ACME: An Architectural Interchange Language*. D. Garlan, R. Monroe, and D. Wile. Submitted for publication, 1996.
3. **[Hein97]** *A Model for Designing Adaptable Software Components*. George Heineman. To Appear in Compsac98.
4. **[OH98]** *Complex Ports and Roles within Software Architecture*. Helgo Ohlenbusch and George Heineman. Submitted for publication. Techreport WPI-CS-TR-98-12.
5. **[Kicz97]** *Open Implementation Design Guidelines*. Gregor Kiczales, et al. 19th International Conference on Software Engineering, pages 481-490, May 1997.
6. **[Rama97]** *A Emacspeak: A Speech-Enabling Interface*. T. V. Raman. Dr. Dobb's Journal, 22(1):18-23, September 1997.
7. **[SUN97]** *JavaBeans 1.0 API Specification*. Sun Microsystems, Inc. December 4, 1996.