Complex Ports and Roles within Software Architecture

Helgo Ohlenbusch and George T. Heineman
Computer Science Department
WPI
Worcester, Massachusetts USA
{helgo,heineman}@cs.wpi.edu
WPI-CS-TR-98-12

Abstract

The Software Architecture community has developed a common vocabulary for describing software components and their interconnections. However, the structure of ports and roles have been too simplistic for capturing even simple examples. This paper explores the part that composition, inheritance, and interfaces play in defining ports and roles. We discuss these concepts within the context of the JavaBeans component model and show how to capture the complexity inherent in the interfaces of components and connectors.

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. There are many obstacles to reusing software components. First, one must locate a component with the exact functionality needed; then, once a component is found that (perhaps only closely) matches the desired need, the software developer must overcome incompatibilities between interfaces, implicit assumptions, and any hidden dependencies that components may have. The motivation for reuse is great, since reusing a component reduces implementation costs and, most importantly, maintenance costs.

One aim of Software Architecture research is to better specify the high level design and overall structure of the system, focusing on the individual components and their interconnections [3]. 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.

The focus of this paper is to better understand the impact that *composition*, *interfaces*, and *inheritance* have on the ADL specifications of components and systems. Object-oriented programming languages have greatly increased the opportunities for reuse through composition and inheritance. As described by Gamma [2], these common techniques are complementary and allow functionality to be easily reused. Their experience leads them to two principles of reusable object-oriented design: (P1) Program to an interface, not an implementation; and (P2) Favor object composition over class inheritance. In this paper, we take a closer look at how to capture the complexity of the interfaces of components and connectors.

1.1 Context

We explore the notions of composition, inheritance, and interfaces by investigating the JavaBeans [5] component model. A Java *Bean* is a reusable software component that can be manipulated visually in a design environ-

This paper is based on work sponsored in part by National Science Foundation grant CCR-9733660.

ment, such as the sample Bean Development Kit (BDK). BDK allows application builders to instantiate a collection of Beans that communicate with each other using *events*. The JavaBeans event model allows components to propagate state change notifications to one or more registered listeners. Each Bean contains a set of state *properties* (i.e., named attributes) that can be customized by application builders. For example, one can change the font, background color, or dimensions of a Bean. A Bean also has public *methods* that other Beans can invoke.

JavaBeans is designed for simplicity – in theory every Java Class is already a Bean; there is no need to subclass from a special `Bean` class. A Bean is defined by its properties, events, and methods. Properties simply reflect an implicit naming scheme of public methods; for example, if a Bean has public methods `void setHeight (int h)` and `int getHeight()`, one can infer that the Bean has a property *Height*. Similarly, if a Bean implements a Java interface `ControlListener` and has a method `void handleControlEvent (ControlEvent e)` then *ControlEvent* is an event that the Bean is prepared to handle.

There are many 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, Connectors, Ports, and Roles are implicit in JavaBeans. An architectural description of the component will simply be documentation of a very abstract nature. Also, interoperability between components from different component models is hindered if each follows their own implicit scheme. A guiding principle of Software Engineering that we follow in this paper is to explicitly define all implicit dependencies and definitions. To overcome this implicit approach, we concretely define the various Port and Role types that are possible in JavaBeans. In this way, we provide a means for a Bean developer to accurately document the interface of the Bean, thus allowing an application builder to fully understand how to reuse this Bean within a software application.

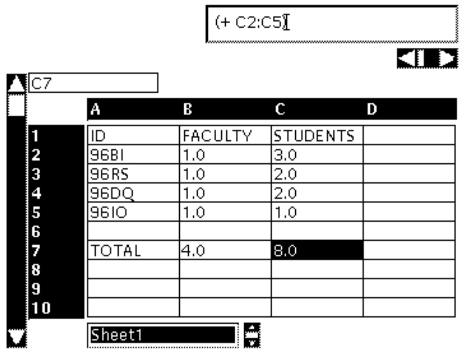

Figure 1: JavaBeans Spreadsheet Application

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 their interactions.

The components react to GUI events (i.e., mouse clicks) and communicate with each other through 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*.

In Section 2 we briefly describe the building blocks of ADLs and how they are reflected in JavaBeans. Section 3 presents the main results of this paper, namely, that composition, inheritance, and interfaces help define the complex structures embodied by ports and roles. Section 5 summarizes our conclusions.

2 Software Architecture

At the architectural level, the first step is to characterize the style selected for the software application. The results of this paper are focused on JavaBeans, but they can be equally applied to any other component model. The seven basic entities common to ADLs [6] are: components, connectors, ports, roles, configurations, representations, and representation-maps. Typically, representations have been responsible for describing the hierarchy found in software systems. However, these have only been applied to components and connectors; in this paper, we argue that hierarchy is essential to capture the complexity of ports and roles.

Architectural descriptions use the terminology of components, ports, connectors and roles to describe a system. Ports define the points of interaction of a component with its environment. Similarly, a role defines the interaction between a connector and a component. The design of a system is an arrangement of components where the connectors represent their interactions. A connector is composed of roles that are connected to specific ports. A connector can be as simple as a function call, but it can also be a complex protocol or an application in itself (e.g. distributed systems using CORBA). Within a style, ports and their corresponding roles have different properties.

Components with complex interfaces are overloaded with many different ports. Ports should have an interface to allow interrelated functionalities to be grouped into one port. Interfaced-ports increase the level of abstraction of the interaction between components. The roles are used to specify which interfaces of the port are being used. There are many ways to partition methods into different ports. If we allow a port to be composed of other ports, we can build a hierarchy of component interaction points.

2.1 JavaBeans Connections

In JavaBeans, a Bean is characterized by its properties, methods and by the type of events it can fire. The JavaBeans model provides two ways for components to communicate with each other. The first type is direct method invocation, whereby a component directly invokes a method on another component. This corresponds to a simple connector with roles such as {caller, callee}. Connectors of this type can only be connected to ports that match the public interface of the Bean. There is no code required for such connectors, but by requiring a Bean to access directly another Bean, this type of communication increases the coupling between the different components.

The second type uses events for the interactions. A Bean registers itself as a listener for a specific event produced by another Bean. The listener Bean only needs to understand how the event is constructed, and it can listener to any Bean that produces the same type of event. We refer to these connectors as event transporters with roles such as {consume-event, deliver-event}. The underlying implementation for this connection is a simple method invocation to a pre-defined method specified by an interface, such as `handleEvent(Event e)`. Again no code is required for the connector; the Bean provides the necessary connection code as defined by the JavaBeans specification. Methods such as `AddEventListener` and `RemoveEventListener` serve this purpose.

Beans can only listen to events that they have been created to handle. To overcome this limitation, the Bean Development Kit (BDK) dynamically generates new Java classes that transform events received from one Bean into method calls (most likely) on another Bean. The connector that is generated knows how to handle a particular event from a source Bean and can directly invoke any public method from the target Bean. This enables Beans to react to a wider set of events.

3 Ports and Roles

Ports and roles define the interaction between components and connectors. We now completely classify the interactions of a Bean with its environment into two different types of ports, `IncomingMethod` and `OutgoingMethod`. These two types constitute the foundation of the hierarchy of all the ports used to describe the JavaBeans architectural style.

The `IncomingMethod` type defines the public interface of the Bean. All public methods of the Bean are identified by ports that inherit from this port type. This type is then further refined using the naming convention defined by the JavaBeans model specification to create a hierarchy of all the services requested by another Bean.

The `OutgoingMethod` type identifies method calls that are initiated by the Bean itself. Interactions of this kind can occur spontaneously (if a Bean has its own thread of control) or by the direct result of an `IncomingMethod` invocation. This second type captures the dependencies between different components and helps to trace outgoing method invocations. These ports are not part of the Bean's public interface, and capture the external dependencies that the Bean has on its environment. Events in JavaBeans are announced using a `FireEvent` port type that inherits from `OutgoingMethod`.

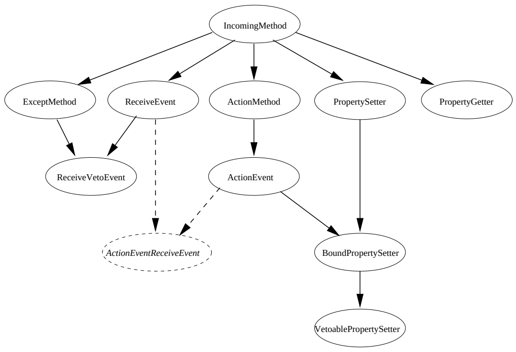

Figure 2: The incoming port type hierarchy

3.1 Type Hierarchy

We now fully describe the JavaBeans core specifications by creating a hierarchy of port and role types. The ACME [1] specification for these hierarchies are contained in Appendix A.

3.1.1 The IncomingMethod hierarchy

The first type hierarchy is based on the `IncomingMethod` port type. This type is refined to capture the different types of method defined in the JavaBeans specifications. The port type associated with a method defines the interaction with the Bean. We first differentiate between method invocations that result in an outgoing method invocation and those that don't. Ports that correspond to methods that do not generate outgoing method invocation are of the primary type `IncomingMethod`. This type is refined to capture the specificity of the simple set and get methods, respectively `PropertySetter` and `PropertyGetter` that are subtypes of the `IncomingMethod` type. The port type `ReceiveEvent` that corresponds to the `HandleEvent` type of methods is a subtype of `IncomingMethod` that recognizes this method as an event handler.

The `ActionMethod` port type is for methods that result in one or more outgoing method invocations. This type is refined to create the `ActionEvent` port type for methods that will produce events. JavaBeans provides a consistent mechanism for announcing state changes, whereby an event is fired whenever a bound property is changed. The port type that corresponds to this type of method invocation is `BoundPropertySetter` and it is a sub-type of both `ActionEvent` and `PropertySetter`. The `VetoablePropertySetter` port type further determines that the particular property is a constrained property and other Bean may deny the set request.

When a Bean is a veto-listener, it uses exceptions to oppose the change of a property. This is the first feature of JavaBeans that relies on the exception handling provided by the Java language. To capture this mechanism we must refine the base type `IncomingMethod` to a new type `ExceptMethod` to record that the method throws exceptions. Using multiple subtyping, we can then build a new type `ReceiveVetoEvent` that is a subtype from `ReceiveEvent` and `ExceptMethod`.

Figure 2 summarizes the port types introduced in this section that constitute the basic types defines in the JavaBeans model. By using multiple subtyping we can create more complex types to better fit real Beans. For example, a method that is initiated by an event (type `ReceiveEvent`) that also results in the firing of one or more events will be of a type that will subtype from `ReceiveEvent` and `ActionEvent`.

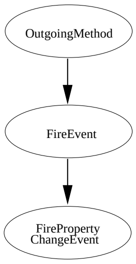

Figure 3: The outgoing port type hierarchy

3.1.2 The OutgoingMethod hierarchy

A Bean can also invoke methods from other components. The port types that capture this interaction are based on the `OutgoingMethod` port type are are shown in Figure 3. This first specialization is for the case where a Bean fires an event. Even though the `fireEvent` method of a bean is typically a private method, we need to specify a port for the interaction between the Bean and the other components. To differentiate between different event types that are fired, we can further specialize the `FireEvent` port type. The only event defined by the JavaBeans model is the `PropertyChangeEvent`, corresponding to the `FirePropertyChangeEvent` port type.

3.1.3 Role type hierarchy

Connection between beans are formed from one or more `OutgoingMethod` ports to one or more `IncomingMethod` ports. The roles used to build this type of connector have different types, with a hierarchy similar to the one of the port types. The two base types of roles are `InvokeMethod` and `ReceiveMethod`, and they correspond to the `IncomingMethod` and `OutgoingMethod` port type hierarchy.

The `InvokeMethod` role is refined to `GetProperty`, `SetProperty`, `DeliverEvent`. One reason why the role `SetProperty` is not split into different roles (bounded and constrained) is that it is out of the scope of the connector; its task is only to perform the change and not to monitor it. To monitor a change, the connector would have to be connected to the corresponding `FireEvent` port. A role of type `SetProperty` can be connected to several types of port (any port type in the `SetProperty` hierarchy). The `ConsumeEvent` role is a subtype of the `ReceiveMethod` role and can be connected to any `FireEvent` port.

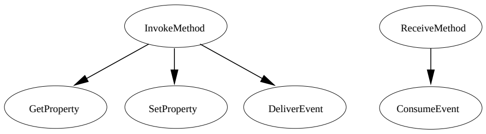

Figure 4: The role type hierarchy

When designing an application, each Bean is specified by one or more instances of the port types. The connectors then are defined using different instances of the role types defined in the style. When the connectors are attached to Beans, the different types of the ports and roles enable static checking for compatibility and traceability.

3.2 Interface

A one to one mapping from methods to ports is sufficient when designing small application; however, the design becomes unwieldy as the complexity increases. Once we view ports as having interfaces, then we can create complex structures describing the composition of ports. Ports need to have an interface for two reasons. First, a port with an interface supports functional overloading. For example, a port that defines `setHeight(int)` can be extended to support `setHeight(float)`, `setHeight(string)`. Second, interfaced-ports enable "similar functionalities" to be grouped into one port, and thereby leads to simplification of the design.

When *functional* partitioning is used, all ports that affect the Bean in the same way are grouped together. This reduces only somewhat the complexity of a Bean's interface. A more dramatic reduction is seen when all ports with the same type are grouped together. For example, there would be one port of type `PropertySetter` that would be composed from the many individual `PropertySetter` ports for each of the Bean's properties. This is a *behavioral* partitioning, since the ports are grouped based upon the external behavior as seen by the environment; Figure 5 describes this extreme. This partitioning presents a simple interface to

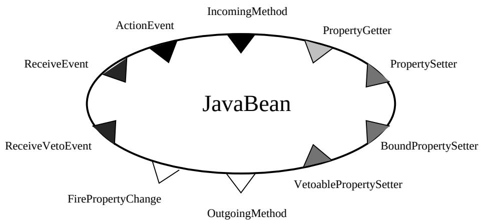

Figure 5: Representation of a Bean with interfaced-ports

connectors seeking to attach to the Bean. It dramatically reduces the number of ports and introduces a simple two-level structure to the Bean's methods which are too often viewed as a flat list of methods. For an event-based model such as JavaBeans, this approach makes it easier to trace event propagation. From the design we can directly see that when a connector is connected to the interfaced-port of type `BoundPropertySetter` this will cause the bean to fire an event using the `FirePropertyChange` port. This view enables the tracing of the different interactions. However, this partitioning may obscure the many semantic differences between the ports and methods.

A third possible partitioning is *informational* that groups all ports that affect specific properties together. For example, the `PropertySetter` and `PropertyGetter` for a particular Bean property would be composed into the same port. This grouping most closely matches the spirit of JavaBeans, and can readily support composition of interrelated properties. There are problems, however, with classifying methods that affect multiple properties.

Since the ports have a complex structure, the roles must correspondingly reflect this complexity. We thus move away from the vague use of roles such as {caller, callee} and – for the first time in the software architecture literature – associate a structure and interface with roles. By doing so, we finally allow ports and roles to capture the many complex interactions between components and connectors. As a visual aid, the outgoing ports in Figure 5 are represented with triangles pointing outside the Bean.

3.3 Composition

To simplify the design, we need to be able to group different ports or interfaced-ports into one entity. For example, a behavioral approach would group together all the ports that inherit from `PropertySetter` (i.e. `PropertySetter, BoundPropertySetter` and `VetoablePropertySetter`).

An informational approach allows designers to partition the ports based on their relationships. Some attributes of an object can be interrelated; for example, a Bean may have a *Width* and *Height* property, as well as a combined *Dimension* property. To capture this type of nested interdependencies we use a port Dimension that is composed of two sub-ports, one for Height and one for Width. The Dimension port is then composed of 4 ports corresponding to get-dimension, set-dimension and two other ports for Width and Height.

Composing ports makes it possible to have different level of depth at the design level. The designer groups the methods and their associated ports into logical and inter-dependent entities. The same composition rule can be applied to the roles of the connector. Without modifying the complexity of the specification of the design, composition of ports associated to interfaced-ports introduce a hierarchy in the architectural design. In the same way we can look into the design of one component or connector by "zooming" into the connection at the design level. Figure 6 presents a visual metaphor (based on fractals) for the hierarchy of ports. Connector "B" simply attaches to the Dimension port while connector "A" attaches to specific elements of the port, including the sub-port Height and the individual method `getDimension()`.

4 Putting it all together

Returning to our motivating example, we would like to interconnect the various Beans to produce a fully-functioning spreadsheet. The TableBean objects react to mouse events by generating `TableEvent` events. The applet *app* has installed itself as a listener on *tb* for these events; the TableBean thus invokes the `handleTableEvent` object on its

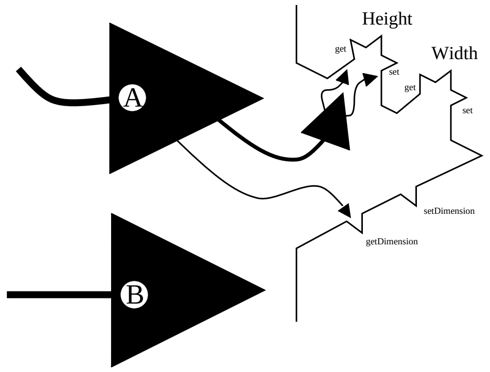

Figure 6: Details of a “complex” connection

```
public void handleTableEvent (TableEventObject teo) {
  TableElement from = teo.getFrom();
  TableElement to   = teo.getTo();

  // Offset these table events (by leftColumn/topRow) to
  int offsetR = tbR.getTopRow() - 1;
  int offsetC = tbC.getLeftColumn() - 1;

  Cell fromCell = new Cell (from.getColumn() + offsetC,
                            from.getRow() + offsetR);
  Cell toCell = new Cell (to.getColumn() + offsetC,
                          to.getRow() + offsetR);

  if (teo.getSource() == tb) {
    // on MousePress, un-select any column or row

    if (teo.getType() == TableEventObject.MOUSE_PRESS) {
      tbC.setSelectedRegion (null);
      tbR.setSelectedRegion (null);
    }

    // Show this region in the tbBox TableBean.
    CellRegion cr = new CellRegion (fromCell, toCell);
    tbBox.setTableValue (1, 1, cr.toString (true));

    // Show value of the first cell of cr in textBean
    Cell cell = cr.getStart();
    String s = spreadsheet.getValue (cell.toString());
    if ((s == null) && (! textBean.getText().equals ("")))
      textBean.setText ("");
    else {
      textBean.setText (s);
      textBean.select (0, s.length());
    }
  }
}
```

Figure 7: Connector for processing TableEvent events

listeners whenever it generates events. The `handleTableEvent` method is part of the applet and performs the following functions: (1) determine the actual spreadsheet cell based upon the `leftColumn` and `topRow` offsets; (2) unselect any selected Columns or Rows in *tbC* and *tbR*, respectively; (3) update *tbBox* to contain the absolute reference of the desired cell; (4) retrieve the value of this cell from the spreadsheet; and (4) display this value in *textBean*.

This fragment is really the code for a connector between the five collaborating beans: *tb*, *textB*, *tbBox*, *tbC*, and *tbR*. The roles for this connector are implicit, and the ports for the various Beans are also implicit; our task now is to identify the roles and ports explicitly. As described in Section 3.2, there are multiple ways in which to group the individual ports of a component. Each grouping affects the definition of the roles, and hence the connectors. For space reasons, we can only present the details of the *TableBean* component. The language used to describe the ports and components is our Component Specification Language (CSL) [4].

4.1 Behavioral

The specification of the *TableBean* interface in Figure 8 shows how the various ports are combined based upon their type; for example, all the `getProperty()` methods are now part of the port *TBGetter*. The *TableBean* component type is defined to contain an instance of each of these port types. This partitioning provides a simple two-level structure for the methods of the Bean and enables some rudimentary analysis of the system. For example, when a connector attaches to the *TBGetter* port, one can infer that no property values will change.

```
Port Type TBGetter extends PropertyGetter with {
  String       getTableValue (int, int);
  int          getRowHeight ();
  int          getColumnWidth ();
  Dimension    getPreferredSize ();
  Color        getForeground ();
  Color        getBackground ();
  TableRegion  getSelectedRegion ();
  int          getNumRows ();
  int          getNumColumns ();
};

Port Type TBSetter extends PropertySetter with {
  void setTableValue(int, int, String)
};

Port Type TbBoundSetter extends BoundPropertySetter with {
  void        setRowHeight (int);
  void        setColumnWidth (int);
  void        setForeground (Color);
  void        setBackground (Color);
  bool        setSelectedRegion (TableRegion);
  void        setNumColumns (int);
  void        setNumRows (int);
};

Port Type TBIncomingMethod extends IncomingMethod with {
  void       paint (Graphics);
  void       redraw (Graphics, int, int);
  bool       select (TableRegion);
  bool       select (TableElement, TableElement);
  void       addTableListener (TableListener);
  void       removeTableListener (TableListener);
  void       addPropertyChangeListener (Listener);
  void       removePropertyChangeListener (Listener);
  void       clearTableValue ();
};

Port Type TBActionEvent extends ActionEvent with {
  void mouseClicked  (MouseEvent);
  void mousePressed  (MouseEvent);
  void mouseReleased (MouseEvent);
  void mouseEntered  (MouseEvent);
  void mouseExited   (MouseEvent);
  void mouseDragged  (MouseEvent);
  void mouseMoved    (MouseEvent);
};

Component Type TableBean extends JavaBean with {
  Port Getter                    : TBGetter;
  Port Setter                    : TBSetter;
  Port BoundSetter               : TBBoundSetter;
  Port FireTableEvent            : FireEvent;
  Port FireEvent                 : TBActionEvent;
  Port FirePropertyChangeEvent   : FireBoundPropertyEvent;
  Port MethodCall                : TBIncomingMethod;
};
```

Figure 8: Behavioral partitioning

4.2 Informational

The specification of the *TableBean* interface in Figure 9 creates a richer structure for the ports, and groups individual methods together based upon their effect on the Bean's properties. The *TableValue* port, for example, has `setTableValue` and `getTableValue` – the basic ones assumed from the JavaBeans model – and a third method that clears out all values, `clearTableValue`. A connector can attach to the entire port, in which case it has access to all its functionality, or it could attach simply to the the `getTableValue` subport. This partitioning creates more ports than a behavioral partitioning, but it more closely follows the object-oriented approach. Note how the *SelectedRegion* port is composed of the basic functionality of a *BoundProperty* port, with the additional functionality as provided by the two `select` methods. Finally, consider the *TableEvent* port which reacts to mouse events from the user interface. This port contains a sub-port that is responsible for firing the *TableEvent* events. Through composition, it is clear that the *TableEvent* events are directly related to mouse events, an inference we could not have drawn from Figure 8.

Figure 10 contains the informational specification for the connector whose implementation is shown in Section 4. This notion of defining an interface before the actual implementation is a key concept from object-oriented technology. Note how the connector describes each of the functionalities that it needs to function properly; at a glance, we can see that this connector will process a `TableBean` event, retrieve various property values from other components, and set the `TableValue` property of a component. Through composition, `handleTBEvent` accurately describes the interactions that occurs between the various beans. Also, the attachments clause shows exactly the elements from the various ports that are accessed by the role.

```
Port Type PropertyPort extends PropertySetter,
                               PropertyGetter with { };
Port Type BoundPropertyPort extends BoundPropertySetter,
                                    PropertyGetter with { };
Port Type EventPort extends ActionEvent, FireEvent with { };

Port TableValue : PropertyPort extended with {
  Port void setTableValue (int,int,String) : PropertySetter;
  Port String getTableValue (int,int)      : PropertyGetter;
  Port void clearTableValue ()             : PropertySetter;
};

/* Similar for ColumnWidth (deleted for space reasons) */
Port RowHeight : BoundPropertyPort extended with {
  Port int getRowHeight  ()     : PropertyGetter;
  Port void setRowHeight (int)  : PropertySetter;
};

Port Dimension : BoundPropertyPort composedof RowHeight,
                 ColumnWidth extended with {
  Port Dimension getPreferredSize() : PropertyGetter;
};

/* Similar for Background (deleted for space reasons) */
Port Foreground : BoundPropertyPort extended with {
  Port Color getForeground ()      : PropertyGetter;
  Port void  setForeground (Color) : PropertySetter;
};

Port Graphic : BoundPropertyPort composedof Foreground,
               Background, Font extended with {
  Port void paint  (Graphics)           : IncomingMethod;
  Port void redraw (Graphics, int, int) : IncomingMethod;
};

Port SelectedRegion : BoundPropertyPort extended with {
  Port TableRegion getSelectedRegion ()    : PropertyGetter;
  Port bool setSelectedRegion (TableRegion): PropertySetter;

  /* Interfaced-port in a composed port */
  Port select : IncomingMethod extended with {
    bool select (TableRegion);
    bool select (TableElement, TableElement);
  }
};

Port TableEvent : EventPort extended with {
 Port void mouseClicked  (MouseEvent evt) : ActionEvent;
 Port void mousePressed  (MouseEvent evt) : ActionEvent;
 Port void mouseReleased (MouseEvent evt) : ActionEvent;
 Port void mouseEntered  (MouseEvent evt) : ActionEvent;
 Port void mouseExited   (MouseEvent evt) : ActionEvent;
 Port void mouseDragged  (MouseEvent evt) : ActionEvent;
 Port void mouseMoved    (MouseEvent evt) : ActionEvent;

 Port void addTableListener(TableListener) :IncomingMethod;
 Port void removeTableListener(TableListener):IncomingMethod;
 Port FireTableEvent                         :FireEvent;
};

/* Similar for NumRows (deleted for space reasons) */
Port NumColumns : BoundPropertyPort extended with {
  Port void setNumColumns (int) : PropertySetter;
  Port int getNumColumns ()     : PropertyGetter;
};

Port BoundPropertyChangeEvent : EventPort extended with {
 Port addPropertyChangeListener    (Listener):IncomingMethod;
 Port remove PropertyChangeListener(Listener):IncomingMethod;

 Port FirePropertyChangeEvent : FireBoundPropertyEvent;
};
```

Figure 9: Informational partitioning

```
Role Text: PropertyRole extended with {
  Role getText: GetProperty;
  Role setText: SetProperty;
};

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 with {
    Text.set to RoleText.set;
    Text.get to RoleText.get;
  }
};
```

Figure 10: Connector for Figure 9.

5 Conclusion

This work is a preliminary investigation on capturing the complexity inherent in the interface of software components. We evaluated a simple application composed of nine JavaBeans and, in doing so, revealed unexpected complexity in specifying the interconnections between the components. The goal of our efforts was to thoroughly specify the application using ACME, and our failures at doing forced us to question the simple notion of ports and roles.

We defined three possible views – informational, behavioral, functional – of a component's interface and fully specified the interfaces of the individual components. 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 descriptions are when constructing applications from software components.

Our ultimate research goal is to develop techniques for adapting software components. This paper outlines a necessary first step – the detailed specification of a complex component's interface. Next we will define techniques to help application builders adapt a component by modifying the specification and providing new code to extend the component. It is our expectation that sophisticated specifications will be necessary before sophisticated components can be reused and adapted.

About the Authors

Helgo Ohlenbusch is a graduate student at WPI in Worcester, Massachusetts. He can be reached at helgo@cs.wpi.edu.

George T. Heineman is an assistant Professor in the Department of Computer Science at WPI in Worcester, Massachusetts. Heineman was a visiting researcher at the Centre for Advanced Studies at IBM during the summer of 1993. He can be reached at heineman@cs.wpi.edu.

References

[1] John E. Arnold and Steven S. Popovich. Integrating, Customizing and Extending Environments with a Message-Based Architecture. Technical Report CUCS-008-95, Columbia University, Department of Computer Science, September 1994.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Software*. Addison-Wesley, Reading, MA, 1995.

[3] 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.

[4] George T. Heineman. A Model for Designing Adaptable Software Components. In *Twenty-Second Annual International Computer Software and Applications Conference*, Vienna, Austria, August 1998. To appear.

[5] Sun Microsystems, Inc. JavaBeans 1.0 API Specification. Internet site (`http://www.javasoft.com/beans`), December 4, 1996.

[6] Nenad Medvidovic and Richard N. Taylor. A Framework for Classifying and Comparing Arhchitectural Description Languages. In *Proceedings of the 6th European Software Engineering Conference ESEC '97*, 1997.

A ACME description of JavaBeans

The following is a complete description in ACME of the base ports and roles as defined by the JavaBeans specification.

```
/* PORTS */
Port Type IncomingMethod = {
  Property comment = "may throw an exception";
};

Port Type ActionMethod    extends IncomingMethod with { };
  Port Type ActionEvent       extends ActionMethod with { };

Port Type ReceiveEvent    extends IncomingMethod with { };
  Port Type ReceiveVetoEvent  extends ReceiveEvent with { };

Port Type PropertyGetter  extends IncomingMethod with { };
Port Type PropertySetter  extends IncomingMethod with { };

Port Type BoundPropertySetter extends ActionEvent, PropertySetter with {
  Property comment = "FirePropertyChangeEvent";
};
Port Type VetoablePropertySetter extends BoundPropertySetter with { };


Port Type OutgoingMethod = {
  Property comment = "Call of an external method";
};
Port Type FireEvent extends OutgoingMethod with = { };
Port Type FirePropertyChangeEvent extends FireEvent with {
  Property comment = "Catches exception from the Vetos" ;
};

Component Type JavaBean = { };

/* ROLES */
Role Type InvokeMethod = {
  Property comment = "may catch an exception";
};
Role Type ReceiveMethod = { };

Role Type ConsumeEvent extends ReceiveMethod with = { };

Role Type GetProperty extends InvokeMethod with = { };
Role Type SetProperty extends InvokeMethod with = { };
Role Type DeliverEvent extends InvokeMethod with = { };

Connector Type BeanConnector = { };
```