WPI-CS-TR-97-6 (revised) February 1998

A Model for Designing Adaptable Software Components

by

George T. Heineman

Computer Science

Technical Report

Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department

100 Institute Road, Worcester, Massachusetts 01609-2280

A Model for Designing Adaptable Software Components

George T. Heineman
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609, USA
+1 508 831 5502
heineman@cs.wpi.edu
WPI-CS-TR-97-06

Abstract

The widespread construction of software systems from pre-existing, independently developed software components will only occur when application builders can adapt software components to suit their needs. We propose that software components provide two interfaces – one for behavior and one for adapting that behavior as needed. The ADAPT framework presented in this paper supports both component designers in creating components that can easily be adapted, and application builders in adapting software components. The motivating example, using JavaBeans, shows how adaptation, not customization, is the key to component-based software.

1 INTRODUCTION

An important aim of software engineering is to produce reliable and robust software systems. As software systems grow in size, however, it becomes infeasible to implement software systems from scratch. Most software developers are familiar with reusing code from component libraries to speed up tedious programming tasks, such as constructing graphical user interfaces. However, it is still an elusive goal to construct applications entirely from pre-existing, independently developed components. This paper presents a technique for designing software components that provide a mechanism for adapting their behavior. Typically, software components offer services defined by a public interface that hides the actual implementation of those services. We propose that software components provide two interfaces – one for behavior and one for adapting that behavior as needed. We also believe that the component must make visible its key policy decisions to allow application builders to adapt the component.

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, one must still overcome syntactic incompatibilities between interfaces, and implicit assumptions and dependencies that components may have. The motivation is great, since reusing a component avoids implementing the same functionality from scratch and (more importantly) reduces maintenance costs. However, using a software component in a different manner than for which it was designed is challenging because the new context may be inconsistent with implicit assumptions made by the component. Techniques such as component adaptors [25] that overcome syntactic incompatibilities between components do not address the need to adapt software components.

There will be an increasing problem (perhaps we may call it the software component crisis) in using components constructed by other developers. There may be no way to standardize these heterogeneous components (although consider component models such as JavaBeans [16]) and there is no guarantee that an application builder will find a component to *exactly* match a particular need. Thus we must support both component designers and application builders: the designers should create components that can easily be adapted (thus increasing reuse), and application builders need mechanisms for adapting software components.

A review of the literature on component-based software development reveals many types of components, such as calendars and calculators, but increasingly more powerful components are also being developed. Visual Components [13] are a collection of ActiveX components for Windows applications, such as spreadsheets, spell checkers, HTML browsers, and database front-ends. A recent NIST Advanced Technology Program [18] involves sixteen companies pursuing the automated composition of complex large-scale applications from "relatively small" fine-grained components.

These "Black box" components allow minimal customization and are reusable only if they exactly match a particular need in an application. For example, a groupware application builder will not be able to use a database component to store application data if the default transaction behavior of the component cannot be altered to share data among multiple users. The use of a component is thus heavily dependent upon (1) the match in functionality between the component's capabilities and the application's requirements, and (2) the ability for application builders to adapt the component to different applications; the latter observation is too often unrecognized.

We make the 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 (possibly radical) different use. If the component designer performed the adaptation, a very different sequence of actions would occur, since the designer has access to the source code, has a full understanding of the design of the component, and will likely select the optimal adaptation. The application builder has none of these advantages and thus may not be able to overcome the many obstacles to adapt the component. We therefore need to support component designer and application builder alike. It is also important to differentiate *adaptation* from *customization*. An end-user customizes a software component by choosing from a fixed set of options that are already pre-packaged inside the software component. An end-user adapts a software component to a new environment by writing new code to alter existing functionality.

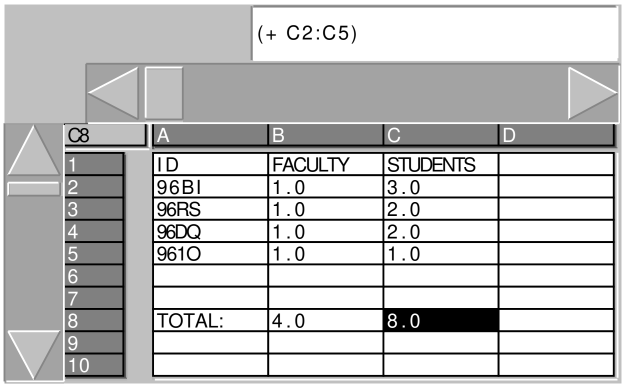

Figure 1: Spreadsheet composed of eight Java beans

1.1 Context

Our framework for adaptable software components is independent of programming language and software architecture. For this paper, however, we assume that the components are written in Java [4] and that applications follow the JavaBeans [16] software component model. A Java *Bean* is a reusable software component that can be manipulated visually in a design environment, such as the sample Bean Developers Kit (BDK) shipped with the initial release of JavaBeans. BDK allows application builders to instantiate a collection of Beans that communicate with each other using *events*. The JavaBeans event model provides a convenient mechanism for components to propagate state change notifications to one or more registered listeners. JavaBeans focuses on components that can be manipulated visually and customized for some purpose. Each Bean contains a set of state properties (i.e., named attributes) and BDK allows application builders to customize a Bean by modifying its properties. For example, one can change the font, background color, or dimensions of a Bean.

1.2 Motivating example

The motivating example is a simple spreadsheet application composed of eight interacting Beans as shown in Figure 1. 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. Lastly, an invisible Spreadsheet Bean *ss* maintains and calculates all values in the spreadsheet, of which only a few are visible as determined by *tb*. The entity responsible for creating these Beans and setting up the interactions between the Beans is the parent Java applet, *app*.

The components react to GUI events (i.e., mouse clicks) and communicate with each other by passing along special 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* (the only one visible) to the designated Column/Row while the contents of the spreadsheet cell (i.e., `(+ C2:C5)`) are shown in *textb*.

For this paper, we consider the following adaptations to highlight our approach: (A1) define notification functions to be invoked whenever the value (not just the contents) of a particular cell changes; (A2) adapt *ss* to only send to *tb* updated cells visible to *tb*; The original component designer can easily make these changes; a competent software engineer could do the same after understanding the source code. For example, some behavioral adaptations are typically implemented using subclasses in C++; this, however, requires that the component be recompiled. We show how to design components so they can be adapted without directly accessing their source code.

2 REQUIREMENTS

We have identified several requirements for designing software components that provide a mechanism for adapting their behavior.

Programming language independence

Since components are implemented in many different programming languages, the mechanism for adaptation must not depend on any language-specific feature. Thus, although the example components in this paper are programmed using Java, the adaptation solutions described in Section 3 do not rely on object-oriented features such as inheritance.

Handle existing code

The mechanism must work equally well for newly-developed code and existing software components. A component designer should not be required to follow a particular architectural style or design pattern; nor should the component classes (if object-oriented) be required to be subclasses of special adaptation superclasses. Thus, we seek the least intrusive means. The additional code needed to convert a legacy component to be adaptable must require only minimal understanding of the component itself.

Design sophisticated interfaces

There is an implicit assumption that the interface of a component is passive while the implementation contains the active execution. An *active interface* is involved in the execution of its member methods, allowing or disallowing method invocations much like a cell membrane allows or prevents substances from entering a cell. In this paper, we show how an application builder can adapt the behavior of a component as necessary without violating its encapsulation. Active interfaces provide an alternative to the common “wrapping” approach, whereby an extra layer is written around an object to alter its behavior.

ADL compatibility

We build upon the large body of work on Architectural Description Languages (ADLs). Thus the component specification language must be compatible with Acme [3], the standard interchange language for architectural descriptions. We also require that adaptations be specified at the architectural level so that the changes to the component can easily be integrated with the component.

3 ADAPT PROJECT

The goal of the ADAPT project is to increase the feasibility of component-based development of software applications by showing how to design adaptable software components. The main idea is that component designers must provide mechanisms that allow application builders to incorporate and adapt these components into their applications. We motivate our research using the example from Section 1.2. We first present our ADL in Section 3.1 then introduce in Section 3.2 active interfaces as a way that component designers can help application builders adapt components. We then show in Section 3.3 our arbitrator mechanism for adapting components based on semantic information. Throughout, we include ADL fragments showing how component specifications are modified to reflect their specifications.

3.1 Component specification language

We are developing a Component Specification Language (CSL) as a common means for describing the interface for a component and its adaptations. As an Interface Description Language [23], CSL describes the interface for adaptable components, and is used to define the adaptation policies. As an Architectural Description Language [3], CSL describes the interactions between components and the internal (private) structure of a component. CSL extends Acme [3] to describe the internal structure of a component without revealing its implementation. A benefit of this language-based approach is that the same language used by the designer to describe the interface of their component is used by an application builder when determining how to adapt the component.

The CSL specification describes where new code and functionality should be integrated to adapt a component, but there are several options for how this will be implemented. The code can be statically compiled and linked together with the component in traditional fashion. Alternatively, the component could dynamically load and execute the new code when needed (as BDK does).

Figure 2 describes the CSL specification for the Spreadsheet Bean component. It has two properties (Function, Value) that can be set, retrieved, or cleared; these are represented as ports. There is a public interface for Spreadsheet, with three functions that any external component can invoke directly. Then there are two ports for communicating using SpreadsheetEvents. A connector can attach to `SpreadsheetGenerator` if it wants to receive events; alternatively, a connector can attach to `SpreadsheetListener` if it wants to give the Spreadsheet component an event to handle. Lastly, there is a port explicitly for adaptation, as required by our ADAPT

```
component Spreadsheet = {
  Port FunctionProperty = {
    void      clearFunction ();
    Function  getFunction   (String s);
    void      setFunction   (String s, Function f);
  };

  Port ValueProperty = {
    void    clearFunction ();
    String  getFunction   (String s);
    void    setValue      (String s, String v);
  };

  Port PublicInterface = {
    float  calculateFunction(Expression e);
    float  getNumericValue(String s);
    void   installFunctions();
  };

  Port SpreadsheetGenerator = {
    void addSpreadsheetListener     (SpreadsheetListener sl);
    void removeSpreadsheetListener  (SpreadsheetListener sl);
  };

  Port SpreadsheetListener = {
    void handleSpreadsheetEvent (SpreadsheetEventObject seo);
  };

  Port Adaptable = {
    void  invokeCallback (String where, String method, Object args[]);
  };

  // Private, adaptable methods are listed
  void    evaluate (Node node);
  float   evaluateConstant(String s);
  void    generateRefreshEvents();
}
```

Figure 2: CSL specification for spreadsheet

framework. Each internal method of the component that can be adapted is also listed separately, not associated with any port. Such an accurate specification of a component's interface is necessary before any attempt to adapt that component.

CSL extends ADLs in two significant directions. First, it defines interfaces for ports (and roles) containing the set of associated functions that describe each port (and role); this provides a basis for attaching roles to ports. Second, as we shall see in the next section, CSL shows how a component designer can tell the application builder where to adapt a component, even if a private method is being adapted.

3.2 Active interfaces

The interface for a component must play a greater role in helping application builders adapt the component. Components are active computational entities whose interface defines methods to invoke, events to receive and/or send or complex access protocols [1]. An *active interface* decides whether to take action when a method is called, an event is announced, or a protocol executes. Following the Acme ontology [3], components have *ports* that represent interaction points between the component and other components in the system. A port is active when there is any communication between the port and its attached connector.

Although there are many different types of ports, there is no common agreement on the exact composition of a port. A port can be a *function* port, representing a function or method of a component. In this case, there are two phases to a port request: 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. A port can be a *data* port, representing the flow of data into or out of the component. In this case, there is only one "after-phase", after the data has been accepted by the port, but before it has been processed. As a third alternative, consider a *dataflow* port that follows the following protocol: the port receives a signal that data is ready, it receives the actual data, then it replies with an acknowledgment that the data was received. This dataflow port has four phases: one before, one after, and two "in-between". Ports define the public communication allowed a component. We also consider the internal component interface consisting of private methods. Although they are private, these internal methods are able to support an active interface and can have associated "before-phase" and "after-phase". Note that revealing the internal interface of a component does not mean that the implementation is revealed.

An active interface allows user-defined *callback* functions to be invoked at each phase for a port (or internal method), and thus may augment, replace, or even deny a port (or method) request. We claim this approach is more general than the standard means of interposing entities between components to intercept/alter port requests. Because such adaptation is likely to occur, the component should provide a mechanism for this purpose. Thus the designer offers great flexibility, and the responsibility for correctness is placed on the application builders that adapt the component.

Each component has an associated *component arbitrator* which maintains the set of all callback functions installed for the active interface. Consider the `void evaluate(Node node)` private method in Figure 2. Figure 3a

```
ComponentArbitrator ca = new ComponentArbitrator();

// before Evaluate (Node) function
Class params[] = ca.paramList ("adapt.spreadsheet.Node");
try {
  ca.insertCallback (ca.BEFORE, "evaluate", gl,
    glueClass.getDeclaredMethod ("beforeEvaluate", params));

  ca.insertCallback (ca.AFTER, "evaluate", gl,
    glueClass.getDeclaredMethod ("afterEvaluate", params));
} catch (NoSuchMethodException nsme) {
    System.err.println ("Unable to install callbacks.");
}
ss.setArbitrator (ca);
```

(a) Creating callbacks

```
private void evaluate (Node node) {

  if (arbitrator != null) {
    Object args[] = new Object[1];
    args[0] = node;
    arbitrator.invokeCallback (BEFORE, "evaluate", args);
  }

  // Original Evaluate Function...

  if (arbitrator != null) {
    Object args[] = new Object[1];
    args[0] = node;
    arbitrator.invokeCallback (AFTER, "evaluate", args);
  }
}
```

(b) Invoking callbacks

Figure 3: Installing and Invoking Active Interface

contains a fragment of code showing (from our Java implementation) how the application-builder sets up before- and after- callback functions for `evaluate`, from the class `glueClass`. Figure 3b shows how the `evaluate` function invokes the appropriate callback functions once installed. These fragments are specific for Java programs, but the concept is independent of the programming language used.

We now implement adaptation A1 using the active interface mechanism. Observe that not every recalculation of a spreadsheet changes the value of a cell. For example, if a cell contains the formula "(count A1:A10)" counting the number of non-empty cells in the given region, it will not change value if the existing non-empty cells are updated. The application builder decides to adapt the Spreadsheet component to include a `before-evaluate` function that records the value of the cell before its update and an `after-evaluate` function that compares the new value against the old. To adapt the component, the application builder modifies the CSL specification of *app* in Figure 4, as shown by the vertical line. The arbitrator is itself a separate component, so conceptually it communicates through the `Adaptable` port in *ss*. The arbitrator interprets the CSL specification, and essentially performs the same functionality as shown in Figure 3.

The `storeValue` and `compareValue` functions are coded (in the Java archive file `code.jar`) and become part of *ss*. Recall that the underlying implementation language for *ss* is Java; similar results can be achieved using C/C++ and dynamic loading. This example shows how additional functionality can be seamlessly integrated with low overhead if the component designers create an active interface. In object-oriented systems, inheritance is often used to extend the behavior of a class; we suggest that, in many cases, active interfaces provide an alternative mechanism.

The active interface mechanism is limited to adapting the behavior of a component at the standard interface boundaries. The arbitration mechanism described in the next section builds upon the active interface by creating special ports that allow policy decisions of the component to be adapted. The insight to arbitration is that the component designer can identify decisions that application builders can augment or replace with their own implementations.

```
System Application = {

  component vs         instanceof ScrollbarBean;
  component hs         instanceof ScrollbarBean;
  component tb         instanceof TableBean;
  component tbBox      instanceof TableBean;
  component tbC        instanceof TableBean;
  component tbR        instanceof TableBean;
  component textb      instanceof TextBean;
  component app        instanceof SpreadSheetApplet;

  component ss         adapts Spreadsheet {
    code     code.jar;
    action   Glue.storeValue (in Node);
    action   Glue.compareValue (in Node);

    void evaluate (Node node) {
      before Glue.storeValue (node);
      after  Glue.compareValue (node);
    };
  }

  connector SpreadsheetEvent = {
    Role generator {
      void addSpreadsheetListener (SpreadsheetListener sl);
      void removeSpreadsheetListener (SpreadsheetListener sl);
    };

    Role consumer {
      void handleSpreadsheetEvent (SpreadsheetEventObject seo);
    };
  }

  ...

  attachments {
    SpreadsheetEvent.generator to ss.SpreadsheetGenerator
    SpreadsheetEvent.consumer  to app.SpreadsheetListener
    ...
  };
}
```

Figure 4: Partial CSL specification for the final application

```
class Glue {
  private java.util.Hashtable values = new java.util.Hashtable (10);

  int storeValue (Node node) {
    Float fl = new Float (node.getNumericValue());
    values.put (node.toString(), fl);
    return 0;
  }

  void compareValue (Node node) {
    Float newValue = new Float (node.getNumericValue());
    Float oldValue = (Float) values.get (node.toString());
    values.remove (node.toString());
    if (oldValue.equals (newValue))
      return;

    notify (node);    // notify appropriate listener
  }
}
```

Figure 5: `storeValue` and `compareValue` code

3.3 Arbitrator to acquire semantic information

The options for an application builder using a component in ways not anticipated by its designer are: 1) modify the component (very hard to accomplish without knowing how the component was constructed); 2) rely on language-specific mechanisms (like inheritance) to replace fragments of the component; or 3) craft a special component adapter that "wraps" the component, interposing itself between the application and the component (requiring complex programming). Nearly twenty years ago, Parnas observed that software should be designed to be easily extended and contracted [20]; the difficulty, of course, lies in foreseeing exactly what features will be adapted. For example, when component $C1$ interacts with component $C2$, $C1$ knows its past history, its future actions, and its usage patterns of $C2$. $C2$ could benefit by having access to this semantic information since it could then select the most appropriate manner in which to process requests from $C1$. A more significant reason for $C2$ to have this information is that $C2$ could be adapted to perform differently in certain situations. Instead of forcing $C1$ to communicate this information directly to $C2$, we seek a generic method for $C2$ to acquire this information.

We extend the component arbitrator described in Section 3.2 to use CSL to model the semantic information of a component and to have mechanisms for acquiring this semantic information. This separation between a component and its arbitrator is essential since it reduces the complexity of the original component, which is not involved in accessing or acquiring the semantic information. It also allows us to reuse this generic mechanism for all components that can adapt their behavior based upon additional information. When a component makes a policy decision, it can ask the associated component arbitrator to invoke any special-purpose policies as determined by the application builder; the component will always have a default behavior in case there is no additional policy defined. The arbitrator then acquires the information and executes actions according to the CSL specification. A CSL specification describes the syntactic interface of a component and all policy decisions that are known to be adaptable.

In our previous work [12], which focused on extending concurrency control for databases, we called this a "mediator-based" approach since the arbitrator acted as a mediator between different system components. The CSL specification describes how new code written by the application builder will be integrated with the component; the application builder can define special functions that retrieve the semantic information from the desired components. These functions become part of the arbitrator and are executed when the arbitrator is asked to fetch the desired semantic information. The policies defined in this language describe situations when the component allow adaptation. Our current implementation has successfully been used to adapt the behavior of a a transaction manager to implement different extended transaction models [12].

Returning to our motivating example, the application builder implements adaptation A2 by efficiently filtering the update messages from *ss* to *tb*. The specification for *app* in Figure 6 has a `VisibleCells` port representing the current region of visible cells. The Spreadsheet component has a function `generateRefreshEvents` that sends to the listeners of the component all the refresh events of new values. The designer of the Spreadsheet component allows flexible update policies by having this function invoke the arbitrator to selectively limit (or increase) the

```
Component app  = {
  Port VisibleCells {
    CellRegion getVisibleCells ();
    void       setVisibleCells (CellRegion cr);
  };
}

Component ss adapts Spreadsheet {
  code     code.jar;

  action   Glue2.filterCells (inout Vector, in CellRegion);

  void generateRefreshEvents () {
    negotiate refreshPolicy (Vector refreshList)
      filterCells (refreshList, app.visibleCells);
  };
}
```

Figure 6: Adaptation of Spreadsheet component

number of refresh events, as shown in Figure 6. The component designer cannot foresee all possible uses of this refresh policy, but creates a place for this policy to be modified. The interface between the component and the arbitrator is defined by a set of negotiation entries (for example, refreshPolicy in Figure 6). Within the `generateRefreshEvents()` function, the designer has the Spreadsheet component invoke the component arbitrator directly:

```
\\ Negotiation Policy: refreshPolicy
arbitrator.resolve ("refreshPolicy", refreshList);
```

If the CSL specification contains any adaptations for this negotiation policy, they are interpreted by the arbitrator. In this example, the arbitrator receives the `refreshList` information from *ss* as a parameter. Second, it acquires the `visibleCells` property from *app* by calling a special `getVisibleCells` function (supplied by the application builder). This function determines the visible region given the property information from Figure 6. Finally, the `filterCells` action is executed by the arbitrator. This function is coded by the application builder to remove from the Vector of updated cells any non-visible cells. To maintain the separation between components, only the arbitrator directly communicates with both *ss* and *app* Note that this behavior could not have been created through either before- or after- callbacks.

A standard solution for implementing A2 would add a parameter to the function, such as `generate-RefreshEvents(visibleCells)`, that would restrict the list of refresh events generated. This is ill-advised, however, since it increases the coupling between the components, needlessly complicates the interface of the Spreadsheet component and limits the potential reuse of each component. This same behavior could have been produced by wrapping Spreadsheet with a layer that filters out refresh events at the listening components, but this would be very inefficient.

The arbitrator approach is useful when the component is solving a problem for which there is no

```
component TableBean {
  indexedProperty String tableValue (int, int);
}

component tbC adapts TableBean {
  code codeC.jar;
  property int leftColumn;
  action Glue4.retValue (int);

  String getTableValue (int col, int row) {
    before Glue4.retValue (col);
  };
}

component tbR adapts TableBean {
  code codeR.jar;
  property int topRow;
  action Glue5.retValue (int);

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

Figure 7: tbC and tbR adaptations

single "best" algorithm or implementation (see Template Method pattern for similar justification [8]). The component designer could produce multiple components, each one optimized for a different context, but this defeats the purpose of reuse. Alternatively one could pre-package a set of implementations (such as OIA/D [15]), but this continues to limit the possible solutions. The arbitrator allows the application builder to adapt the behavior of the component as required and retain the default behavior for most cases. As components become more autonomous and intelligent, the arbitrator will be essential when two interacting components must negotiate to make a common decision.

The arbitration mechanism also provides a convenient way for the component designer to supply code that monitors the use of the component and adapts it. The arbitrator can dynamically construct state information about the processing of the component to make not only performance-enhancing decisions but also decisions that extend the core functionality of the component (for example, servicing a request differently because of a change detected in the built-up state information).

We are currently investigating the many ways in which arbitrators can operate in component-based architectures. One possibility, for example, is to associate a different arbitrator with each component. Alternatively, a federated approach would allow multiple arbitrators, each with their own set of associated components. A centralized approach would have all the components in the application use one arbitrator. There are many issues involved, ranging from how the components and arbitrators communicate, how they resolve differences, and what architecture is suitable for multiple arbitrators. Some existing component-based architectures, for example, place restrictions on the component communication. Batory and O'Malley [5] define a hierarchical layering of components, each of which is limited to communicating with the component one higher/lower in the hierarchy. We believe that components should be relatively insulated from the application architecture, and the arbitrator should be in charge of acquiring semantic information.

The application in the motivating example contains four TableBean components. The column and row Beans, tbC and tbR, clearly demonstrate the distinction between our CSL approach, and the standard object-oriented approach. We could create, for example, a new ColumnTableBean component, sub-classed from Tablebean, that overrides key methods to return the column, such as A, B, AA. Alternatively, we could define an additional property *offset* and install callbacks for the adapted components. Figure 7 contains a partial CSL specification with adaptations.

4 RELATED WORK

There have been many efforts to describe software architectures using an ADL, but no concerted attempt to allow third party application builders to describe adaptations from the software architecture specification. This is the novel addition of our work. Few ADLs support any notion of evolution, and those that do are limited to structural sub-typing [17]. An ADL is not intended to capture all details of an application (that is the responsibility of the underlying programming language of the components), but it should provide some structural assistance when adapting components. CSL extends the Acme [3] language in significant ways as outlined in this paper.

Active interfaces are related to all work that seeks to alter the invocation of a component. From a syntactic point of view, the before- and after- phases are similar to the Lisp advice facility described in [21]. Alternatively, Filter objects [14] manipulate and/or disallow messages between objects and act transparently without violating the encapsulation of the target objects. This particular wrapping approach is heavily dependent upon C++ and adds an extra class/object layer instead of extending the responsibilities of the interface. Active interfaces have similar goals to *reflection* [2], a design principle whereby a system has a representation of itself that makes it easy to adapt the system to changing environments, and *meta-object protocols* [19], where an interface allows incremental modification of the behavior and implementation of a component. Active interfaces are similar to the wrapping of internal objects in C2 [24] except that they allow user-defined functions to be inserted and they can also affect private methods of the component. Finally, active interfaces are distinct from the pre-packaged implementation strategies of OIA/D from which the client selects [15]. OIA/D sketches a solution showing how the client can provide their own implementation strategy, but typically an entire method for a component is replaced. Our approach is more fine-grained, allowing adaptation to occur as needed and we do not violate the encapsulation of the component, since the methods invoked within the active interface do not directly access private information in the component.

The arbitrator is a more generalized instance of the Dialog and Constraint Manager (DCM) in the C2 architectural style [24]. In C2, the internal wrapped component does not initiate interactions with the DCM, but our framework depends on the component requesting assistance from the arbitrator when making key decisions. The arbitrator is also more powerful than techniques that simply monitor a component's usage. The designers of VINO [22], an extensible operating system, suggest that the operating systems kernel can monitor the usage of its resources and adapt to different workload conditions, but the ways in which the component adapts are all pre-determined, fixed implementations.

5 CONCLUSION

5.1 Past Success

There needs to be increased awareness that software components will become effective only when application builders can adapt them. Our previous work with the Programming Systems Laboratory at Columbia University involved constructing a Process Centered Environment, called Oz [6], that supported extended transaction models. As described in [11], we developed an architecture for constructing systems from pre-existing, independently developed software components. The primary difficulty we encountered was forcing components to adapt to fit within a larger application. This is distinct from the architectural mismatch described in [9]. As part of this earlier work, we designed an extensible transaction manager component (written in C) with an active interface and a component arbitrator with a special language for tailoring its behavior based upon user-defined scenarios [10, 12]. In [12] we re-engineered the active interface within the Exodus storage manager [7], thus allowing Exodus to negotiate with the same component arbitrator to change its behavior. The success of this preliminary work confirms that software components can provide an interface for adaptation.

Table 1 describes the size of the Bean components developed for this paper. The Java applet, *app*, that creates the components and directs their interact is only 15% of the entire application. These Beans can be downloaded from `www.cs.wpi.edu/~heineman/ADAPT`.

5.2 Summary

We presented three main ideas in this paper:

- Active interfaces – a language-independent solution for creating components that can be adapted

Component	*LOC*	*Number Classes*
Spreadsheet	1675	17
TextBean	375	4
ScrollbarBean	303	2
TableBean	1055	7
Applet	606	1

Table 1: Size of Beans and Application

by application builders.

- Component arbitrators – components typically do not make visible the underlying policies that dictate their behavior. If the component is to make reasoned decisions to alter one of its policies, it must be able to access semantic information. The component arbitrator provides a powerful mechanism for component designers to specify different policies that the application builder can adapt for their needs.
- CSL specification language – component designers and application builders use CSL to specify the adaptations allowed by a component, and the adaptations required by the application builder. In this way, we support both parties in their efforts.

We expect the impact of our research to increase the use of component architectures, such as JavaBeans, by showing the full potential of component adaptation. By focusing on two of the most complex/costly problems in software development – adapting existing code for new contexts, and designing code to be extensible – our contributions will help solve the difficult problem of developing large-scale, high-quality, and robust software applications.

5.3 Future Work

We are currently extending the Beans Development Kit to interpret CSL specifications during design time, and translate them into efficient implementations at execution time. The users of BDK will be able to compose applications from Bean components and will be able to adapt components, instead of simply customizing them. An active interface will not help an application builder in all adaptations. Changing internal data structures of a component, for example, requires redesign as well as recompilation. We plan to determine the extent to which active interfaces and arbitration help adaptation efforts.

References

[1] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–

364, October 1995.

[2] ACM press. *ACM Reflection '96*, San Francisco, CA, April 1996.

[3] 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. The research described in this report was conducted at Bull HN Information Systems, Inc.

[4] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[5] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[6] Israel Z. Ben-Shaul and Gail E. Kaiser. A Paradigm for Decentralized Process Modeling and its Realization in the OZ Environment. In *16th International Conference on Software Engineering*, pages 179–188, Sorrento, Italy, May 1994.

[7] Michael J. Carey, David J. Dewitt, Goetz Graefe, Favid M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Shekita, and Scott L. Vandenburg. The EXODUS Extensible DBMS Project: An Overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, chapter 7.3, pages 474–499. Morgan Kaufman, San Mateo CA, 1990.

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

[9] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts. In *17th International Conference on Software Engineering*, pages 179–185, April 1995.

[10] George T. Heineman. *A Transaction Manager Component Supporting Extended Transaction Models*. PhD thesis, Columbia University, May 1996.

[11] George T. Heineman and Gail E. Kaiser. An Architecture for Integrating Concurrency Control into Environment Frameworks. In *17th International Conference on Software Engineering*, pages 305–313, Seattle, WA, April 1995.

[12] George T. Heineman and Gail E. Kaiser. The CORD approach to Extensible Concurrency Control. In *13th International Conference on Data Engineering*, pages 562–571, Birmingham, UK, April 1997.

[13] Sybase Incorporated. ActiveX Components for Windows Applications, July 1997. Internet site (`http://www.visualcomp.com`).

[14] Rushikesh K. Joshi, N. Vivekananda, and D. Janakiram. Message Filters for Object-Oriented Systems. *Software – Practice & Experience*, 27(6):677–699, June 1997.

[15] Gregor Kiczales, John Lamping, Cristina Lopes, Chris Maeda, Anurag Mendherkar, and Gail Murphy. Open Implementation Design Guidelines. In *19th International Conference on Software Engineering*, pages 481–490, May 1997.

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

[17] 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.

[18] National Institute of Standards and Technology. ATP Focused Program: Component-Based Software. Internet publication (`http://www.atp.nist.gov/atp/focus/cbs.htm`).

[19] Xerox Parc. Metaobject Protocols.
Internet site (`http://www.parc.xerox.com/spl/projects/mops`).

[20] David L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(6):310–320, March 1979.

[21] T. V. Raman. Emacspeak: A Speech-Enabling Interface. *Dr. Jobb's Journal*, 22(1):18–23, September 1997.

[22] Margo I. Seltzer and Christopher Small. Self-monitoring and Self-adapting Operating Systems. In *Sixth Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 1996.

[23] R. Snodgrass. *The Interface Description Language: Definition and Use.* Computer Science Press, Rockville, MD, 1989.

[24] Richard Taylor, Nenad Medvidovic, Kenneth Anderson, James Whitehead, and Jason Robbins. A Component- and Message-Based Architectural Style for GUI Software. In *17th International Conference on Software Engineering*, pages 295–304, Seattle, WA, April 1995.

[25] Daniel M. Yellin and Robert E. Strom. Protocol Specification and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.