

WPI-CS-TR-97-6

September 1997

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 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 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.

KEYWORDS

Software components, JavaBeans, Adaptation

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 design and construct 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 and mechanism 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 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: one must first locate a component with the exact functionality needed. 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 eliminates 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 [22] that overcome syntactic incompatibilities between components do not address the need to adapt software components.

In the future, we believe many software applications will be composed of software components, as described in [16, 21]. There will be an increasing problem (perhaps we may call it the software component crisis) in using components constructed by other developers. There is no way to standardize these heterogeneous components (although consider component models such as JavaBeans [15]) and there is no guarantee that an application builder will find a component to *exactly* match a particular need. This paper aims to support both component designers and application builders: the designers will be aided in creating components that can easily be adapted (thus increasing reuse), and for the first time application builders will have 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 [12] 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 [16] 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.

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 [3] and that applications follow the JavaBeans [15] 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 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. The need for visual manipulation of Beans, currently assumed by

	A	B	C	D
1	ID	FACULTY	STUDENTS	
2	96BI	1.0	3.0	
3	96RS	1.0	2.0	
4	96DQ	1.0	2.0	
5	96IO	1.0	1.0	
6				
7				
8	TOTAL:	4.0	8.0	
9				
10				

Figure 1: Spreadsheet composed of eight Java beans

JavaBeans, is a self-imposed limitation that we ignore. Our goal is to extend BDK to construct a design environment that helps application builders adapt software components as they construct applications.

1.2 Motivating example

The motivating example is a simple spreadsheet application composed of eight interacting Java Beans [15] (i.e., software components written in Java) 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 shown 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*.

In the JavaBeans model, Beans interact with other Beans by registering with, and processing events from, each other. The TableBean components react to mouse events and generate TableEventObjects for which *app* has registered. 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 are shown in *textb*.

For this paper, we consider the following three adaptations: (A1) define notification functions to be invoked whenever the value of a particular cell changes; (A2) alter *ss* to only send to *tb* updated cells visible to *tb*; (A3) define new functions for *ss* to use. The original

component designer should easily be able to make these modifications, but in this paper we focus on the support needed for the application builder to adapt these components. We chose these adaptations to highlight the different features of our approach.

2 REQUIREMENTS AND CHALLENGES

We have identified several requirements and challenges for our work:

2.1 Requirements

Be language independent

Since components are implemented in many different programming languages, the mechanisms 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. Even so, the ADAPT language has a distinctive object-oriented flavor.

Handle existing code

The mechanisms we develop must work equally well for newly-developed code and existing software components. A component designer should not be required to follow a particular architecture 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 the interface of a component to be adaptable is small, and requires only minimal understanding of the component itself. This makes it possible to reengineer legacy code.

2.2 Challenges

Exploit semantic information in components

Although it is desirable for components to be encapsulated to hide private information, a component must be able to understand its environment to operate appropriately. The performance of a component can be improved if the application builder knows how it will be used. Client patterns in OIA/D [14] describe patterns of use that a component can exploit when determining how best to operate. For example, a component that creates an indexed set of elements can choose different optimal algorithms if it knows whether the frequency of element deletions is low or high. One technique for acquiring such *semantic information* is to have the client present the information to the component, as suggested by OIA/D, but this complicates the component's interface. An alternative we successfully pursued in [10] is for the component to acquire this information from the application (or other components) through an *arbitrator*. The need for applications to make decisions based on semantic information has been shown in diverse do-

main such as operating systems, database management systems, workflow management systems, and software development environments. Since there is no agreed-upon representation of semantic information, we need to provide a language to model semantic information and a generic means to acquire the information.

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* becomes 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. Some approaches already affect the invocation of a component. Filter objects [13], for example, manipulate and/or disallow messages between objects and act transparently without violating the encapsulation of the target objects. This "wrapping" approach is heavily dependent upon C++ and adds an extra layer when instead we should extend the responsibilities of the interface. Other approaches (such as component adaptors [4]) alter the behavior of a component by revealing internal mechanisms of the component for direct invocation by clients. In this paper, we show how an application builder can adapt the behavior of a component as necessary without violating its encapsulation. Since wrapping is a common mechanism in object-oriented programming, the component should provide some mechanism that avoids the need to create extra classes whose sole purpose is to wrap particular aspects of the component.

3 ADAPT

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. The ADAPT project consists of the following research directions:

- Active interfaces
- Arbitrators to acquire semantic information
- A component specification language for specifying the interface of a component and how it is adapted
- A design methodology to aid developers in creating adaptable software components, and a reengineering methodology for converting existing code

We motivate these research directions using the example from Section 1.2. We use our ADAPT language to describe the components and their adaptations. An ADAPT specification describes the interface for a component and how it is adapted. Figure 2, for example,

```

component Spreadsheet {
    implements Serializable,
        SpreadsheetListener;

    // one-dimensional property.
    indexedProperty Function function(String)

    // one-dimensional property.
    indexedProperty String Value(String);

    // Basic state properties of this component
    property boolean debug;

    // Methods
    float getNumericValue(String);
    void installFunctions();

    float evaluateConstant(String);
    void evaluate(Node);
    float calculateFunction(Expression);

    // expects add/remove
    void addSpreadsheetListener(SpreadsheetListener);
    void removeSpreadsheetListener(SpreadsheetListener);

    // SpreadsheetListener Interface
    void handleSpreadsheetEvent(SpreadsheetEventObject);
}

```

Figure 2: ADAPT specification for the Spreadsheet component

contains the initial description of the Spreadsheet component. Note, only public members are mandated to be here – private functions can be included additionally to make private interfaces visible. This specification is either provided by the component designer or generated from the component source code. Properties are state information that can be modified and retrieved using API function calls `getPropertyX()` and `setPropertyX(newValue)` (if the component uses different function names, they can be mapped accordingly). Figure 3 contains the ADAPT specification for the overall application.

3.1 Active interfaces

Our first observation is that the interface must play a greater role in helping application builders adapt the component. The component interface is more than a syntactic description of the method invocations accepted by the component. As defined in [1], components are active computational entities whose interface defines methods to invoke, events to receive and/or send, or complex access protocols. An *active interface* decides whether to take action when a method is called, an event is announced, or a protocol executes. 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 [18]. The significant occurrences for event-based components are (1) when an event is sent or received, and (2) when an event is handled. An active interface allows *callback* functions to be invoked during these phases, and thus may augment, replace, or even deny a client request.

A standard way to alter the event communication between components is to interpose entities that intercept events. Because such adaptation is likely to occur, the component should provide an interface for this purpose. Components with complex protocols [2] should also allow adaptation. The goal of the x-Kernel [11] operating system is to allow new network protocols to be defined using the same kernel; our active interface is applicable in this case as well.

Active interfaces are different from the pre-packaged implementation strategies of OIA/D from which the client selects [14]. 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 when needed. 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; they have special privileges, and are able to access private methods of the component as part of the adaptation. Thus the designer has great flexibility, and can place the responsibility for correctness on the application builders that adapt the component.

Application builders adapt component behavior using its active interface. To implement adaptation A1, for example, observe that not every recalculation of a spreadsheet changes the value of a cell. The application builder adapts 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 ADAPT specification of *app* in Figure 3, as shown by the vertical line.

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.

The arbitration mechanism described in the next section builds upon the active interface by allowing the core

```

application app {

    implements ActionListener,      TableListener,
           SpreadsheetListener,    TextBeanListener,
           Serializable;

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

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

    property int      viewHeight;
    property int      viewWidth;
    property int      spreadsheetHeight;
    property int      spreadsheetWidth;
    property CellRegion tableSelected;
    property CellRegion tableRSelected;
    property CellRegion tableCSelected;
    property int      leftColumn;
    property int      topRow;

    // public methods
    void init();
    void init_gui();
    void init_components();

    // Various Listener Interfaces
    void actionPerformed(ActionEvent)
    void handleTableEvent(TableEventObject)
    void handleSpreadsheetEvent(SpreadsheetEventObject)
    void handleTextBeanEvent(TextBeanEventObject)
}

```

Figure 3: ADAPT specification for the final application

```

Hashtable values = new 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 4: storeValue and compareValue code

capabilities of the component to be adapted. Similar approaches to replace functionality (i.e., through inheritance or user-supplied code) forget that the component exists for a cohesive purpose and arbitrary method replacement violates the encapsulation of the component and is unlikely to succeed without deep knowledge of the component. The insight to arbitration is that the application builder can specify alternative policies where the component’s functionality will still apply.

3.2 Arbitrator to acquire semantic information

Currently, the only options for an application builder wishing to use 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); or 2) 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 [17]; 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 efficient 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 have developed the notion of a *component arbitrator* that uses ADAPT to model the semantic information of a component and has mechanisms for acquiring the 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 ADAPT specification.

In our previous work [10], 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 ADAPT language specifies properties of compo-

```

application app {
  property CellRegion visibleCells;

  component ss adapts Spreadsheet {
    code code.jar;
    property Vector refreshList;
    action filterCells (inout Vector, in CellRegion);

    generateRefreshEvents ()
      negotiate refreshPolicy:
        filterCells (refreshList, app.visibleCells);
  }
}

```

Figure 5: Adaptation of Spreadsheet component

ment, and describes how new code written by the application builder will be integrated with the component; in an ADAPT specification, one can refer to special functions written by the application builder 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 semantic information. The policies defined in this language describe situations when the component allow adaptation. Our current implementation has successfully been used to adapt a transaction manager to allow the behavior required to implement different extended transaction models [10].

Returning to our motivating example, the arbitrator implements adaptation A2 by filtering the update messages to *tb*. The application specification defines the `visibleCells` state property that represents 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 Spreadsheet allows flexible update policies by having this function invoke the arbitrator to selectively limit (or increase) the number of refresh events, as shown in Figure 5. The interface between the component and the arbitrator is defined by a set of negotiation entries (for example, `refreshPolicy` in Figure 5). Within the `generateRefreshEvents` function, the designer has the Spreadsheet component invoke the component arbitrator directly:

```

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

```

If the ADAPT specification contains any adaptations for this negotiation policy, they are interpreted by the arbitrator. In this example, the arbitrator first 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 3. Second, the component arbitrator gets the `refreshList` information from *ss*. 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 hidden cells. Only the arbitrator directly communicates with both *ss* and *app* thus maintaining the separation of the components. 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 `generateRefreshEvents(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 component designer may try and foresee all future use (and adaptations) of the component; often, this is not possible, so at least there is this chance to use the ADAPT language to specify the different policy decisions that can be adapted.

The arbitrator approach is useful when the component is solving a problem for which there is no single “best” algorithm or implementation. 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 [14]), 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 designers of VINO [19], an extensible operating system, suggest that the operating systems kernel can monitor the usage of its resources and adapt to different workload conditions. In similar fashion, the arbitrator we propose can use the before- and after- callbacks to 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, ser-

ving a request differently because of a change detected in the built-up state information).

We plan to investigate 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.

3.3 Component specification language

We are developing the ADAPT (Architectural Description of adAPTable components) language as a common means for describing the interface for a component and its adaptations. As an Interface Description Language [20], ADAPT describes the active interface for adaptable components, and is used to define the adaptation policies. One 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. ADAPT provides an unambiguous description of the adapted component to help the application builder better understand the architecture of the final application.

The ADAPT specification describes where to integrate the new code and functionality to adapt a component, but there are several options for how this will be accomplished. 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. The current JavaBeans component architecture [15] generates some code dynamically when connecting Beans together. JavaBeans requires each Java Bean to be capable of running in a *design environment* and the *generated application*. The design environment guides developers in constructing an application from components, and is optimized to support the designers. The generated application is optimized for efficient execution. We are currently extending the Java Beans beanbox to interpret ADAPT

```

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

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

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

component tbR adapts TableBean {
    code codeR.jar;
    property int topRow;

    action retValue (int);

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

```

Figure 6: tbC and tbR adaptations

component specifications during design time, and translate them into efficient implementations for execution time.

We now describe adaptation A3 that allows the arbitrator to define new functions for the spreadsheet. The Spreadsheet component has a function `evaluate(Node node)` that calculates the value for a given spreadsheet cell. The designers planned for new user-defined functions to be added, so this function in Spreadsheet invokes the component arbitrator whenever an unknown function appears. The application builder has defined a policy `userFunction` in ADAPT:

```

component ss adapts Spreadsheet {
    code code.jar;
    action calcFunc (in Node);

    evaluate (Node node) {
        negotiate userFunction:
            calcFunc (node);
    };
}

```

The application builder provides the functions `calcFunc`, that determines whether the given Expression is a user defined function, and if so, performs the calculation. `ss` invokes the component arbitrator at a negotiation policy within the `evaluate` method.

The application in the motivating example contains four TableBean components. The column and row Beans, `tbC` and `tbR`, clearly demonstrate the distinction between our ADAPT 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 6 contains a partial ADAPT specification with the adaptations marked with a vertical line.

3.4 Design Methodology

We propose a Design For Adaptation (DFAD) design methodology to support the creation of adaptable software components. Most design methodologies suggest by implication that the designed software will be easily extensible; what is missing, however, is an explicit plan for how the software component can be adapted. DFAD helps component designers determine the fixed aspects of a component versus the adaptable aspects. The guiding principle behind DFAD is that software components must be constructed to be easily adapted without requiring complex or extensive modifications.

The design methodology that comes closest to our approach is Open Implementation Analysis/Design (OIA/D) [14]. A software module that allows client control of the implementation strategies (i.e., which algorithm to use) within the module has an open implementation. OIA/D requires designers to analyze the various scenarios under which a module is to be used and construct an interface allowing clients to select one or more pre-packaged implementation strategies. In this manner, the client (and not the component) is able to select the most efficient implementation given its particular context. The purpose of OIA/D is thus to improve performance without altering functionality.

DFAD helps component designers identify those key policy decisions within a software component that can be adapted. There are several possible strategies to determine the adaptable aspects of a software component ranging from functional to behavioral. A purely functional approach determines the actions performed and allows individual actions to be modified or replaced. A purely behavioral approach determines the externally visible states of the software component, and modifies the transitions between these states, or adds entirely new states. We suggest that the component implementations remain private while the key policy decisions made by the implementations should be exposed so that they can be adapted (see [14] for a supporting view); in doing so, the encapsulation of the component is not violated. Following DFAD, the component designer constructs an active interface for a component and uses the ADAPT language to define the policies that can be adapted. In our spreadsheet example, the designers realized that the Spreadsheet component would need to

allow user-defined functions, so they constructed the interface to include the `userFunction` negotiation policy to allow future application builders to adapt Spreadsheet.

DFAD systematically helps the component designer construct a consistent interface for adaptation. This promotes greater reuse and adaptation of the component. It also replaces all ad hoc approaches to modify and adapt components. It is not possible, however, to determine all adaptations in advance, and since adaptation is incremental, we need to address the evolution of adaptable software. It should be possible, for example, for the component designers to safely continue to evolve a component while application builders are adapting it. For this reason, we have designed a companion reengineering method.

In conjunction with DFAD, the Reengineer For Adaptation (RFAD) method shows how, with limited effort, existing software components can be reengineered to become adaptable software components. The motivation for RFAD is our success at reengineering the Exodus storage manager [7] to support an active interface upon which we adapted the concurrency control policy. We introduced a negotiation policy within Exodus to contact the arbitrator component when two clients requested the same page with conflicting lock modes. The arbitrator then adapted Exodus by allowing both clients access only when the existing semantics determined that the two clients were cooperating (and thus were using their own correctness criterion).

The first step to RFAD is to determine the interface to the software component, install the active extensions, and specify the component in ADAPT. This may be a challenging task if the component is poorly documented and designed, but we envision semi-automatic tools can help. If it is infeasible to modify an existing component, either because of difficulty or missing source code, one can still “wrap” the component with an additional layer that becomes the active interface. The second, more complex, step identifies the key policy decisions within the component that can be adapted. This is a hard process, but the component designer understands the component, and the work would have to be performed anyway if the component were to be adapted. The benefit of reengineering a component to conform to our proposed adaptation model is that there is a consistent framework within which adaptation takes place. DFAD and RFAD serve the same goal – to create an adaptable component. Existing C, C++, and Java software repositories should be reengineered using the RFAD method, while all new code development should be developed under the guidance of DFAD. In both cases, the ADAPT language will accurately describe components and their allowed adaptations.

<i>Component</i>	<i>LOC</i>	<i>Number Classes</i>
Spreadsheet	1675	17
TextBean	375	4
ScrollbarBean	303	2
TableBean	1055	7
Applet	606	1

Table 1: Size of Beans and Application

4 CONCLUSION

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 [9], 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. 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 [8, 10]. In [10] 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.

4.1 Impact of Work

We presented three main ideas in this paper:

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

- ADAPT specification language – component designers and application builders use ADAPT 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.

There is growing interest in component-based software development, and this research will make such efforts practical and possible. 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 impact all fields of computer science struggling with the difficult problems of developing large-scale, high-quality, and robust software applications. We will continue our efforts at extending BDK to parse and understand ADAPT specifications of components. 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.

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] Robert Allen and David Garlan. Beyond Definition/Use: Architectural Interconnection. In *ACM Interface Definition Language Workshop*, 1994. SIGPLAN Not. 29, 8.
- [3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.
- [4] Roger Barga and Calton Pu. A Practical and Modular Method to Implement Extended Transaction Models. In *21st International Conference on Very Large Data Bases*, Zurich, Switzerland, 1995.
- [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] George T. Heineman. *A Transaction Manager Component Supporting Extended Transaction Models*. PhD thesis, Columbia University, May 1996.
- [9] 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.
- [10] 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.
- [11] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
- [12] Sybase Incorporated. ActiveX Components for Windows Applications, July 1997. Internet site (<http://www.visualcomp.com>).
- [13] Rushikesh K. Joshi, N. Vivekananda, and D. Janakiram. Message Filters for Object-Oriented Systems. *Software – Practice & Experience*, 27(6):677–699, June 1997.
- [14] 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.
- [15] Sun Microsystems, Inc. JavaBeans 1.0 API Specification. Internet site (<http://www.javasoft.com/beans>), December 4, 1996.
- [16] National Institute of Standards and Technology. ATP Focused Program: Component-Based Software. Internet publication (<http://www.atp.nist.gov/atp/focus/cbs.htm>).
- [17] David L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(6):310–320, March 1979.
- [18] T. V. Raman. Emacspeak: A Speech-Enabling Interface. *Dr. Jobb’s Journal*, 22(1):18–23, September 1997.
- [19] 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.
- [20] R. Snodgrass. *The Interface Description Language: Definition and Use*. Computer Science Press, Rockville, MD, 1989.
- [21] Business Week. The Software Revolution. December 4, 1995.
- [22] 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.