An Evaluation of Component Adaptation Techniques[*]

George T. Heineman   Helgo M. Ohlenbusch
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609, USA
{heineman,helgo}@cs.wpi.edu

WPI-CS-TR-98-20

March 4, 1999

Abstract

Adapting existing code to include additional functionality or behavior is a common theme in software engineering. These efforts are complicated because the developer adapting the code will rarely be the designer of the code. As object-oriented and component-based software development achieve greater widespread use, there will be a distinct need to support such third-party adaptation. This paper describes the issues and challenges surrounding component adaptation and surveys various approaches in the literature. We evaluate these adaptation techniques by comparing their use in adapting an existing component in a sample application. Our experience leads us to a better understanding of the similarities and differences between existing adaptation techniques.

1 Introduction

A driving force behind component-based software development is the idea of "plug-and-play" programming. Components, it appears, combine the best features of object-oriented technology and reusable software. We must admit, however, that the promise of building software systems from highly-reusable software components has not yet been achieved. The primary difficulty for this lack of success is the inherent conflict between designing a fixed reusable component and the bottom-up construction of software systems from pre-built components. Some might argue that no code should be written before a software system is designed but there are many reasons why this occurs in practice. First, if a system is designed to incorporate pre-existing components, the system builder might have to modify or work around an existing component. Second, systems are often divided into individual subproblems that typically can be implemented independently (for example, using a Recursive Parallel life cycle [6]) and implementation may occur in stages. Third, requirements can change after significant parts of a software system have been implemented. Thus, software engineering practice forces us to find strategies for adapting existing code.

Even before component adaptation, however, there are many obstacles to simply reusing independently developed software components. It is often difficult to locate a component with the specified functionality; then, once a component is found that (perhaps only closely) matches the desired need, there may be incompatible interfaces. Finally, it is a technical challenge to use a software component in a different manner than for which it was designed and documented. For this paper, we assume that an application builder has somehow located a component developed by a third party.

We believe component-based software will only become widespread when third-party application builders can adapt components as needed. Most components are released with a documented Application Programming Interface (API). This interface, however, only describes the functionality of

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

the component and provides no insights for adapting the component. Sometimes, a component is released with a special source code license allowing code modifications and the application builder is responsible for compiling the component. The Hot Java Component from `javasoft.com` [16], for example, is released with this intriguing message:

> *A source code license allows developers to view and modify the source code. You might want this extra flexibility to custom-fit the HTML Component to very small devices or to add or integrate functionality to the product.*

This is hardly adequate support for adapting this component. We suggest that components be deployed with a specification describing its composition and behavior. Then, when an application builder specifies a desired adaptation, the component shows how to incorporate the new code. Designing for change is an established concept in software engineering that requires the designer to consider future extensions when designing a component. However, there is an understanding that the original design team will be extending the component. Designing for adaptation suggests that the designer should provide extra mechanisms so that the component can be adapted by third-party application builders.

When an application builder adapts a component, the goal is to integrate the adapted component into a working system that satisfies some system requirements. When designers evolve their own software, they seek to change the code so as to maintain the integrity of the original design and minimize costs of future maintenance. These differing goals, revealing the gap between designers and adapters, show the two perspectives we must consider when considering adaptation techniques.

1.1 Adaptation, Evolution, and Customization

We make the distinction between software *evolution* and *adaptation*. Evolution occurs when a software component is modified by the original component design team or by programmers hired to maintain and extend the component. It is assumed that the software engineers can freely modify the source code of the component. Another feature of evolution is that the newly evolved component will become available for purchase and reuse. In contrast, adaptation occurs when an application builder acquires a third-party component and creates a new component C_A to use within the target application. Adapted components, as a rule, will not be released for public use, and reuse of C_A will typically occur only within the company that adapted component C.

To further emphasize the difference between evolution and adaptation, assume that the source code is available and that the component design team and the application builder wish to extend the component with the exact same behavioral change. When the design team performs the extension, they typically have a full understanding of the component's design and will likely select the optimal changes to make. The application builder, in contrast, does not have the time to comprehensively understand the source code and seeks to learn just enough to make the desired changes. The application builder may be unable to overcome the many obstacles to component adaptation without a suitable adaptation technique.

We also need to differentiate *adaptation* from *customization*. End-users customize a software component by choosing from a fixed set of options that are already pre-packaged inside the software component. End-users adapt a software component for a new use by writing new code to alter existing functionality; customization, thus, has a limited range.

Figure 1 presents our perspective on component adaptation. Given a software component (represented by a small black square), the large oval represents the space of possible evolution paths for a component, one of which is shown by an arrow. The distance between the two components is proportional to the difference between the components. The component has a pre-packaged set of options that enables customization, as represented by the small dark-gray circle; the apparent difference between a customized component and its original is very small. The oddly shaped light-gray region represents the possible adaptations that can be performed by an application builder. The area for each region is proportional to the situations in which the component can be reused. We

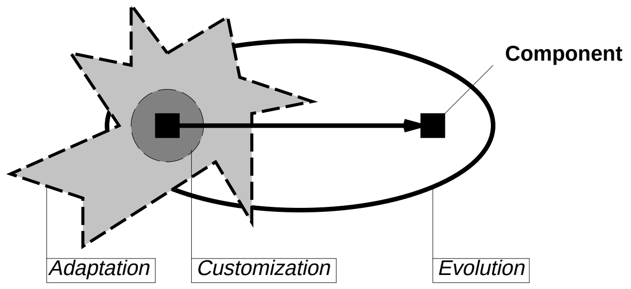

Figure 1: Perspective on Adaptation, Customization, and Evolution

need to understand the types of allowable adaptations to better understand the relationship between these three areas.

1.2 The Role of Software Architecture

We view the research and results in software architecture as essential to any techniques for component adaptation. Software architecture is commonly defined as a level of design that specifies the overall system structure of a software application [9]. These structural issues include system organization, global control structure, communication protocols, and composition of design elements. Because a component is adapted to operate within the context of a larger application, there needs to be a global understanding of the interaction between the component and the application as well as a detailed understanding of the adaptations to the component. An Architectural Description Language (ADL) should be used for both purposes.

Early work in Software Architecture focused on categorizing different *architectural styles*, sets of design rules for composing an application from inter-connected components [1]. Many ADLs have been proposed that can describe, model, and analyze the specific architecture for a software systems [2, 20, 21]. Implicitly, however, the target audience for an ADL specification has been the designers and developers of the original system. We believe that an ADL specification for a component must describe the fixed and extensible features of a component and provide a guide for its adaptation. This is a responsibility that has not yet been addressed by the software architecture community. The active interface technique is a step in this direction.

Component adaptation is strongly related to *Architectural Evolution*, a research area concerned with the addition, removal, or replacement of components or connectors that comprise a component-based application [26]. The adaptation techniques in this paper, however, are focused on creating an adapted component C_A from an existing component C. Whether dynamic [3, 26] or static [15], architectural evolution is not a competing technology, but one that should be used in conjunction with component adaptation techniques.

In Section 2 we discuss and analyze various adaptation techniques from the literature. Section 3 describes our success at applying five of these techniques to solve a component adaptation problem. We close the paper with a summary of lessons learned and comparison with related work. There are several contributions of this paper. First, we compile together and evaluate various requirements from papers discussing component adaptation techniques. Second, we describe *active interfaces* [12], a specific adaptation technique that increases the reusability of software components. Third, we compare and evaluate various techniques for component adaptation in the literature.

2 Component Adaptation Techniques

We first evaluate various requirements for component adaptation drawn from articles in the literature. In Section 2.2 we briefly describe six component adaptation techniques we selected for evaluation. Section 2.3 presents a comparison matrix summarizing the adaptation techniques and how they compare with our consolidated set of requirements. Some of the conclusions shown in this table are based on our experience in using four of these adaptation techniques to adapt a simple component-based application as described in Section 3.

2.1 Requirements

To set the context for our comparison, consider an application builder that acquires a component C from a third-party. The application builder employs an adaptation technique to construct a new component C_A from the original component C. The technique may rely only on ad-hoc solutions or it may provide some specific adaptation mechanism. C_A is then used as a component within the target application. If C already exists as a component in an application, we classify the situation as *adaptive evolution*. Contrast this with a standard integration problem where the application builder must modify the application so that component C can be used as is.

We compiled a list of requirements from [7, 12, 17]. We considered three additional requirements for this paper and have consolidated the total list to a set of eleven possible requirements which we have divided into requirements on C and C_A, requirements on the adaptation technique, and requirements on the adaptation mechanism. In Section 2.1.4 we evaluate these requirements to determine inconsistencies and compatibilities.

2.1.1 Adapted component C_A and original component C

1. Homogeneous – the code that uses C_A should use C_A in the same manner as it would have used C ([12], was *transparent* in [7]).
2. Conservative – aspects of C there were not adapted should be accessible without explicit effort by C_A (was included as *transparent* in [7]).
3. Ignorant – C should have no knowledge of its adaptations (was included as *transparent* in [7]).
4. Identity – C should continue to retain its own identity as a separate entity; this eases the way in which future updates of the component will be handled [17].
5. Composable – C_A should itself be open to future adaptations; it should be straightforward to compose together a set of desired adaptations [7].

2.1.2 Adaptation technique

6. Configurable – the adaptation technique should be able to parameterize and apply a particular adaptation (the *generic part*) to many different components (the *specific part*) [7].
7. Black-box – the adaptation technique should have no knowledge of the internal implementation of C [7, 17].
8. Architectural focus – There should be a global description of the architecture of the target application together with a specification of C and a modified description of C_A [11]; the specifications of C and C_A must be different. This will enable the application builder to specify the adaptation(s) at an architectural level.
9. Framework independent – the adaptation technique must not be dependent upon the component framework to which C belongs. For example, the technique must function equally well on COM [22], CORBA [10], and JavaBeans [23] components.

2.1.3 Adaptation mechanism

10. Embedded – the adaptation mechanism must exist within C before C can be adapted into C_A [12].
11. Language independent – the adaptation mechanism must not be dependent upon the language used to implement C [12]; this requirement also pertains to the adaptation technique.

2.1.4 Evaluation

As a general rule, these requirements help to decrease coupling. For example, if a component is not ignorant of its adaptations, then coupling increases between the original component C and its adaptations. If an adaptation mechanism is dependent upon a particular language, there is an increased coupling between the component and the mechanism. Other requirements ensure that the basic properties of components are retained, namely that the adapted component continues to be composable and reusable. It is not necessary for a particular adaptation mechanism to satisfy all of these requirements.

Some adaptation mechanisms require a component to be designed in a specific way for adaptation to occur (consider customizable black-box adapters [5]) and are thus inapplicable in most cases. We feel it is still reasonable for an adaptation mechanism to suggest minor extensions to the implementation of a component.

We considered and discarded some requirements for this paper. A component technique is reusable if either a generic adaptation can be reused, or a specific modification can be applied to multiple components [7]. We feel that this is simply an extension of being configurable. We also considered that a technique should be *reversible*, that is, it should be possible to always revert to an earlier adaptation, or in fact, to the original component instance itself. We decided that this should be supported not by the adaptation technique, but by a suitable configuration control mechanism.

2.2 Adaptation Techniques

This section briefly describes six adaptation techniques we evaluated for this paper. We were unable to effectively evaluate certain adaptation techniques such as Superimposition [7] because the corresponding adaptation mechanisms was unavailable for download. We chose not to pursue the in-place modification because this was clearly the least desirable of all the adaptation techniques.

2.2.1 Active Interfaces

A component interface is defined by a set of *ports*; in [12] we argue that this interface must play a greater role in helping application builders adapt the component. An *active interface* for a component can be programmed to take action when a method is invoked. A *port* is associated with a set of methods, so each method request is a port request as well. 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[1]. We also consider the internal component interface consisting of private and protected methods. Although private to the component, these internal methods are able to support an active interface and can have their own *before-phase* and *after-phase*. Revealing the internal interface of a component in this way does not reveal its implementation.

An active interface allows user-defined *callback* methods to be invoked at each phase for a method and thus may augment, replace, or even deny a method request. Briefly, each component has an associated *component arbitrator* that maintains the callback methods installed for the active interface. The arbitrator and the component communicate through a special *Adaptable* port. An adaptation to a component is specified at an architectural level and is translated into lower level adaptations. This approach is more general than the standard means of interposing proxies or wrappers [8] between components to intercept method requests.

The active interface mechanism, as described, is limited to adapting the behavior of a component at the standard interface boundaries. In general, a component designer can create special ports that allow policy decisions of the component to be adapted. In this way, the interface for the component is augmented, as in Open Implementation [18], to enable key decisions to be adapted. The adaptation technique of active interfaces is supported by the internal adaptation mechanism of a component arbitrator. Such an arbitrator can easily be integrated into any component as shown in [13].

[1]For this paper we limit discussions to method ports; see [25] for further discussion of other port types.

2.2.2 Binary Component Adaptation

Binary Component Adaptation (BCA) is an adaptation technique that applies adaptations to component binaries without requiring any source code access [17]. Component adaptation occurs after the component has been deployed and the internal structure of the component is modified in place. The BCA system is currently implemented to work with Java [4], an object-oriented language that is compiled into bytecode binaries that are executed within a Java Virtual Machine (JVM). An application builder wishing to adapt a Java component constructs a *delta file specification* containing information about the desired changes to a class; this includes adding or renaming an interface, method, field, or method reference. One can even alter the superclass for a component. A Delta File Compiler (DFC) creates a binary *delta file* containing the necessary bytecode adjustments to the component being adapted.

Once a component is adapted, other classes that refer to the adapted component must be recompiled using a modified `javac` Java compiler. The ClassLoader for `javac` merges bytecode streams from the original `Component.class` file and the extra bytecode stored in the binary delta file. The newly adapted component must then execute within a JVM (version 1.1.5) that has similarly been modified to include the extended ClassLoader; see [17] for further details. The BCA adaptation technique is supported by such adaptation mechanisms as a modified ClassLoader and the DFC.

2.2.3 Inheritance

Inheritance is a mechanism that allows an object to acquire characteristics from one or more objects [6]. *Essential* inheritance relates to the inheritance of behavior and other externally visible characteristics of an object while *incidental* inheritance emphasizes the inheritance of part or all of the underlying implementation of a general object. Essential inheritance is a way of mapping real-world relationships into classes and is used mostly during the analysis and design phase of an object-oriented project. Incidental inheritance often is a vehicle for simply reusing or sharing code that already exists within another class.

Inheritance is both an adaptation technique and mechanism. It is automatically built-in to any component written using an object-oriented language like Java or C++. Inheritance has the benefit that newly created subclasses are separate from the original component being adapted. However, component adaptation through inheritance often reverts to incidental inheritance since the adapter must have detailed understanding of the internal behavior and functionality of a superclass to implement a successful change.

2.2.4 In-place modification

In-place modification occurs when the application builder applies the necessary changes directly to the source code for a component. Naturally, such an approach is possible only if the source code is available and if the application builder is capable of understanding the component's code well enough to make the desired changes. There are no supporting mechanisms for this technique, and we include this technique solely as a baseline for comparison.

2.2.5 Superimposition

Superimposition is an adaptation technique that allows an application builder to adapt a component using predefined and configurable *adaptation types* [7]. These adaptation types are much more expressive than BCA and are thus more complex. The principle behind superimposition is that a component and the functionality adapting the component should be decoupled from each other. Superimposition has been implemented using a layered object model (LayOM) which was unavailable for download; please refer to [7] for further details.

	Homogeneous	Conservative	Ignorant	Identity	Composable	Configurable	Black-Box	Architectural focus	Framework-independent	Embedded	Language-independent
Active Interfaces	y	y	☒	y	y [1]	☒ [4]	y	y	y	y [7]	y
Binary Component Adaptation	y [2]	y	y	y	y [3]	☒ [5]	y	☒ [10]	y	☒	☒ [8]
Inheritance	y	y	y	y	y [11]	☒	y	☒	y	y	☒ [9]
In-place modification	y	y	☒	☒	y	☒	☒	☒	y	☒	y
Superimposition	y	y	y	y	y	y	y	☒	y	☒	☒ [9]
Wrapping	y	☒	y	y	y	☒ [6]	y	☒	y	☒	y

1. The callback methods can themselves be composed together
2. Must execute component within modified Java 1.1.5 virtual machine
3. One can extend a delta class file appropriately
4. Since active interface changes are made in the specification, one could design a separate layer that can configure the same adaptation to multiple components
5. One could design a pre-processing layer that applies a particular change to multiple delta files
6. One could design a flexible wrapper generator that generates unique wrappers for use with multiple components
7. We show in Section 3.3.4 how to insert an active interface into certain components if the source code is unavailable
8. BCA theoretically can be applied to object code from any high-level language, but there are serious obstacles to such efforts; the current system operates only with JDK 1.1.5
9. Applicable only for components written in an object-oriented language
10. Can be integrated with architectural focus as shown in Section 3.3.4
11. Over time, may become impossible to further adapt a class through inheritance as class hierarchies become increasingly tangled

Figure 2: Comparison matrix

2.2.6 Wrapping

As an adaptation technique, wrapping can be used to alter the behavior of an existing component C. A *wrapper* is a container object that wholly encapsulates C and provides an interface that can augment or extend C's functionality. Bosch separates wrapping, whereby the behavior of C is adapted, from aggregation where new functionality is composed from existing components [7]. Hölzle argues that wrapping leads to poor performance as well as an excessive amount of adaptation code [31]. The Adapter and Decorator patterns from [8] are useful ways in which to coordinate the controlled extension of classes, but it is typically very hard to impose a design pattern onto an existing class hierarchy. The Wrapping technique typically has no supporting adaptation mechanism.

2.3 Comparison Matrix

As seen in Figure 2 there is complete agreement on the homogeneous, composable, and framework-independent requirements. This is likely because component technology supports these core features. There are some requirements (configurable, architectural-focus) that only one technique satisfies. This is likely because the adaptation technique considers the particular requirement as a discriminating factor when comparing itself against other adaptation techniques. Consider architectural-focus and active interfaces: there is no reason why the other approaches cannot incorporate an architectural focus into their technique. Similarly, if configurable adaptation is important, as superimposition believes, then the other approaches can rapidly improve to meet this new requirement.

3 Description of the Evaluation

A suitable evaluation of these adaptation techniques would compare their effectiveness at solving a real situation; for a fair evaluation, we tried to minimize the variability. We thus chose to apply all techniques to components written in Java, and we decided to apply the same adaptation to a single application. The C2 architectural style provided such a demonstration component [29]. We rate the amount of effort needed (low, medium, or high) according to three measures: how challenging was the actual programming task? how much knowledge of the class hierarchy was needed? how much knowledge of C2 was needed?

We plan to carry out controlled experiments to extend the early findings presented in this paper. Our current results should be viewed as a fact-finding mission to determine the scope of future experiments.

3.1 Sample Application

Figure 3b contains the architecture of the *StackVisualization* application (SV)[2]. The building blocks of the C2 architectural style are components (white boxes) and connectors (thin gray rectangles) [30]. An application is constructed from a layer of components and each component is unaware of the components that reside "beneath" it at a lower layer. Messages sent "up" the layered hierarchy are requests while messages sent "down" are notifications. The top (bottom) of a component can be welded to the bottom (top) of only one connector. A connector can have more than one component welded to its top and bottom. Messages are sent and received in first-in/first-out fashion.

We model the C2 architectural style using our Component Specification Language (CSL) [12]. C2-components have four port types: `RequestOut`, `NotifyOut` (sub-typed from `OutgoingMethod`) and `RequestIn`, `NotifyIn` (sub-typed from `IncomingMethod`). These port types can be combined to create four new port types: `RequestInRequestOut`, `RequestInNotifyOut`, `NotifyInNotifyOut`, `NotifyInRequestOut`. For example, a `NotifyInRequestOut` port type generates a request in response to receiving a notification. It is invalid to have a port type `RequestInNotifyIn` in C2 because notification and request messages are sent in opposite directions. It appears rare in C2 to have a component that only generates requests without receiving notifications, or a component that spontaneously generates notifications without a previous request. The `Java` implementation of C2 relies upon C2-port objects that are associated with C2-components. In C2, the abstract design elements of ports are instantiated and become part of the implementation; thus there is an object **tp** (**bp**) for the top (bottom) port of a C2-component. Contrast this, for example, with the implicit nature of JavaBeans [23].

Figure 3a contains a sample screenshot of SV in action. Using the buttons, a user can `push` (`pop`) an integer on (off) a stack; `top` places the topmost element of the stack into a textfield and `quit` exits. SV is constructed from three components: `StackADT` maintains the stack state, `StackArtist` visualizes the stack in an abstract "viewport", and `GraphicsBinding` realizes the viewport using Java's Abstract Windowing Toolkit (AWT). `MainBus` and `BindingBus` are connectors that transmit requests and notifications through the component hierarchy.

3.2 Methodology

We first selected a sample adaptation to apply to a particular component that would change both its behavior and functionality. We chose to extend SV so that pushing an n-ary operator onto the stack applies the operator to the n topmost elements of the stack; for example, in Figure 3a, pushing "*" should result in a stack of two elements: { 1683, 18 }. We decided against modifying `StackADT` since this would require either (1) the basic stack type to allow non-integer elements; or (2) new methods in the interface to process elements on the stack. Both of these choices reduce the cohesion of the `StackADT` component. We chose instead to adapt `StackArtist`.

[2]SV is distributed with the C2 Java distribution [29].

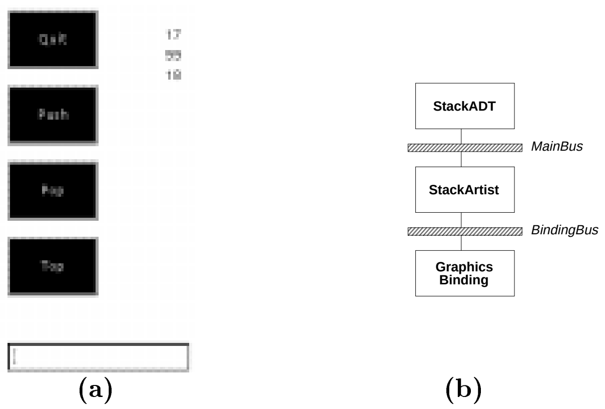

Figure 3: StackVisualization application with conceptual C2 architecture

```
port type NotifyOut extends OutgoingMethod {
    void send (Notification n);
}
port type RequestOut extends OutgoingMethod {
    void send (Request r);
}
port type NotifyIn extends IncomingMethod {
    void handle (Notification n);
}
port type RequestIn extends IncomingMethod {
    void handle (Request r);
}
port type NotifyInRequestOut {
    NotifyIn      Incoming;
    RequestOut    Outgoing;
}
port type RequestInNotifyOut {
    RequestIn     Incoming;
    NotifyOut     Outgoing;
}
component type C2-Component {
    port Top extends NotifyInRequestOut {
        C2-port topPort (); // Returns tp
    }
    port Bottom extends RequestInNotifyOut {
        C2-port bottomPort (); Returns bp
    }
}
component type C2-Connector {
    port Top extends NotifyInRequestOut {
        void      addTopPort (C2-port p);
        void      removeTopPort (C2-port p);
        Vector    topPorts ();
        C2-port   topPortAt (int n);
    }
    port Bottom extends RequestInNotifyOut {
        void      addBottomPort (C2-port p);
        void      removeBottomPort (C2-port p);
        Vector    bottomPorts ();
        C2-port   bottomPortAt (int n);
    }
}
```

```
component StackADT extends C2-Component {
    // push (value), pop, top, get_state
} implemented by {
    StackADT    ST;
    C2-Port     bp;
    Map (Bottom, bp);
}
// BindingBus is identical to this description
component MainBus extends C2-Connector implemented by {
    C2-Port t, b;
    Map (Top, t);
    Map (Bottom, b);
}
component StackArtist extends C2-Component {
    // AcceptEvent (button), AcceptEvent (text_field)
    port Bottom.Incoming {
        void handleUserInput (Request r); // helper function
    }
    // pushed (value), popped (value), top (value), state (stack), empty
} implemented by {
    StackArtist   SA;
    C2-Port       tp, bp;
    Map (Top, tp);
    Map (Bottom, bp);
}
component GraphicsBinding extends C2-Component {
    // ViewportCreated, ApplicationTerminated, ViewportDestroyed,
    // ViewportCleared, PanelAdded, PanelCleared, ButtonAdded,
    // TextFieldAdded, TextFieldSet, TextFieldCleared, c2Grectangle
    // c2Gline, c2Garc, c2Goval, c2Gtext
} implemented by {
    GraphicsBinding    GB;
    C2-Port            tp;
    Map (Top, tp);
}
```

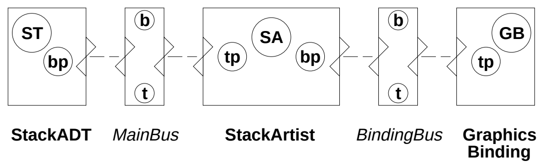

Figure 4: C2 definition in CSL

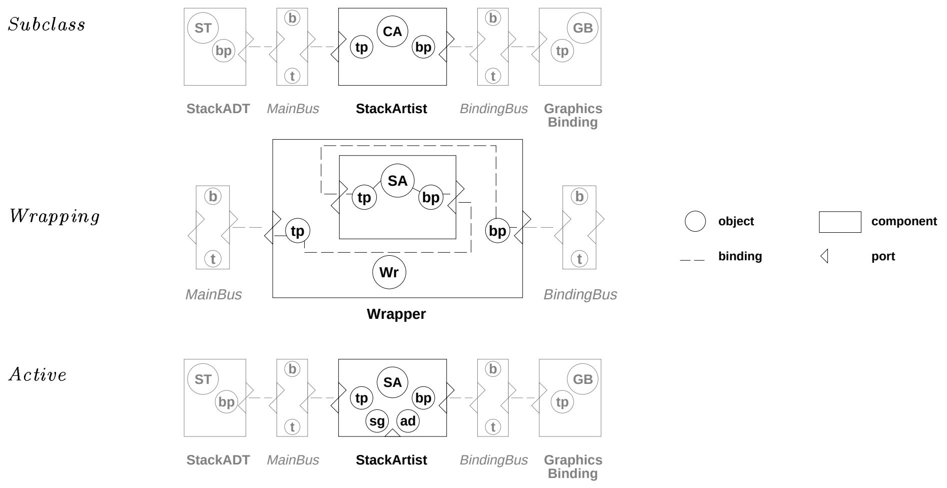

Figure 5: Modified architectures

The adaptations were carried out by one of the authors over a four day period in the following order: Inheritance, Wrapping, and Active Interfaces. The BCA adaptation was added after a few weeks and was completed in several hours. We do not consider this a robust experiment; nonetheless, this experience has proven useful in qualitatively comparing the different adaptation techniques.

The CSL definitions for C2 and the SV application are shown in Figure 4. The details in the upcoming sections are relevant since they show the level of effort required by various adaptation techniques; certain conclusions in the comparison matrix shown in Figure 2 are based on this experience. We include modified CSL specifications as they occur.

3.3 Results

We now apply the alternatives from Figure 2 to adapt the `StackArtist` component. The adapted software architectures for various adaptation techniques are shown in Figure 5.

3.3.1 Inheritance

When using inheritance as the adaptation technique, we define a new subclass `CalculatorArtist` from the existing class `StackArtist` and thus replace the internal object **SA** with a new object **CA** instantiated from `CalculatorArtist`; note that inheritance operates within the component being adapted. **CA** implements the same interface as `StackArtist`, invoking original methods with `super.handle (Request)` and `super.handle (Notification)` when the behavior is unchanged. To implement the stack-based arithmetic, **CA** intercepts `push` messages (from `GraphicsBinding`) that contain n-ary operators, enters calculator mode, and sends n pop requests up to `StackADT`. **CA** then receives these pop requests, calculates the function, issues a push request with the new value, and exits calculator mode. Note that the CSL specification for this adapted system is unchanged because C2 hides internal implementation details behind a standardized component interface; this makes specification of the adaptation difficult.

3.3.2 Wrapping

Wrapping results in a more complicated component because C2-components are constructed to communicate only with C2-connectors. To wrap `StackArtist`, we create a component `Wrapper` that is inserted into the architecture where `StackArtist` used to be. When `Wrapper` receives original notifications (or requests) needed by `StackArtist`, `Wrapper` passes them in, using the `handle`

```
component StackArtist extends C2-Component {
    port Adapt {
        void setAdapter (ComponentAdapter ca);
        ComponentAdapter getAdapter ();
    }
    port Bottom {
        port Incoming {
            void handleUserInput (Request r) {
                before sg.beforeRequestIn
                after sg.afterRequestIn
            };
        }
        port Outgoing {
            void send (Notification n) {
                after sg.afterNotifyOut
            };
        }
    }
    port Top {
        port Incoming {
            void handle (Notification n) {
                before sg.beforeNotifyIn
                after sg.afterNotifyIn
            };
        }
    }
} implemented by {
    StackArtist             SA;
    StackArtistGlue         sg;
    C2-Port                 tp, bp;
    ComponentAdapter        ad;
    Map (Top, tp);
    Map (Bottom, bp);
}
```

Figure 6: CSL specification for application and adapted component

`(Notification)` (or `handle (Request)`) methods provided by `StackArtist`. `Wrapper` implements stack-based arithmetic in the same manner as above. One tricky business was processing the notifications and requests coming out of the original `StackArtist`. Since C2 does not allow `StackArtist` to be connected to multiple C2-connectors, we use the functionality provided by C2-port objects and we link **StackArtist.tp** to deliver messages to **Wrapper.bp** (and similarly **StackArtist.bp** to **Wrapper.tp**). Now, when internal `StackArtist` sends requests up, they are received by the bottom port of `Wrapper` and sent up to higher components; the reverse occurs for notifications. These details show the impractical side of using wrapping as an adaptation technique.

3.3.3 Active interface

Integrating an active interface into a component results in an architecture that preserves the original integrity of the `StackArtist` class and extends the component specification to incorporate the new functionality. `StackArtist` includes a new port that associates a component adapter **ad** with the `StackArtist` component. Figure 6 shows the modified CSL of the `StackArtist` component.

Active interfaces are realized by a small set of helper objects that manage the adaptations for a component. **ad** allows before- and after- callback functions to be inserted for the methods that the designer of `StackArtist` has designated to be adaptable. This is accomplished by altering the CSL description for `StackArtist` to insert these callback functions with the appropriate methods. Once active interfaces are installed in a component C, a third-party can simply insert new code to be invoked at the selected phases; in our example, this code is placed into a `StackArtistGlue` object **sg**. Because the callback functions are associated with the component being adapted, two components of the same type can have different adaptations. Alternatively, the same callback method can be used to

	(1)	(2)	(3)
active interfaces	low	low	medium
BCA	low	low	medium
in-place adaptation	low	low	medium
inheritance	low	medium	medium
wrapping	medium	low	high

Table 1: Comparison of results (lower is better)

adapt multiple components. We had to manually construct the initiation code that instantiated the `StackArtist` component and installed the adaptations; we are currently working on a pre-processor to automatically generate such code.

3.3.4 Binary Component Adaptation

After completing the active interface adaptation, we observed that BCA and active interfaces are supporting technologies. In particular, we found that we could insert an active interface onto an existing Java component using BCA. We reused the `StackArtistGlue` class created for the active interface adaptation and in less than one hour had completed the BCA adaptation. This partnership was an unexpected benefit of carrying out this evaluation.

4 Summary

We successfully adapted the SV application to become a stack-based calculator using the four adaptation techniques described in previous sections. In Table 1, we compare the results by rating the following: (1) difficulty in programming; (2) difficulty in understanding class hierarchy; (3) difficulty in understanding component model.

Although in-place adaptation requires low effort, it is clearly not the preferred technique. One important reason is that it will be impossible to incorporate new versions of the component if the adaptations are made directly to the component's source code. Also, if all adaptations are embedded within the original component, there may be no way to restore the original component (unless version control is applied). Lastly, multiple adaptations made to the same component will quickly interfere with each other unless the adapter is aware of the difference between original code and added code.

BCA and active interfaces are preferred next since they are both supported by adaptation mechanisms. Although they differ when compared by the requirements in Figure 2, they can be used in conjunction to overcome each other's weaknesses. For example, if source code is unavailable, BCA can be used to instrument active interfaces onto a Java component; also, if one requires an architectural focus, active interfaces can be specified using CSL and this specification can be converted into the appropriate delta files for use by BCA.

The *inheritance* option is preferred next because it follows good design practice; its difficulty naturally arises from having to thoroughly understand the object-oriented class hierarchy. Lastly, and somewhat surprising, we determined that although *wrapping* is a simple concept, its realization can be complicated. Composability within a component framework is useful, but not sufficient, for adaptation to occur. When combining the results of [31] with our results, it is clear that wrapping is not sufficient for component adaptation.

From the perspective of the application builder, *wrapping* and *active interfaces* are most easily expressed in CSL. The CSL specifications reveals the micro-architecture of the components that are used to construct the final software system, thus increasing opportunities for adaptation. BCA, however, can easily be extended to include more of an architectural focus.

5 Related work

This paper presents a framework for comparing adaptation techniques for software components. This work is closely related to several areas of prior research. The first area is the software architecture community. There are many ADLs defined (such as [20, 21, 2]) and they have been used to describe and analyze specific software architectures to detect race conditions and deadlock situations. Our work is perhaps the first in the community to target the use of ADLs as a vehicle for specifying and instrumenting adaptations for software components; by doing so, we will be able to take advantage of the powerful analyses offered by the community. Recent work proposed by Medvidovic and Rosenblum [24] identifies various domains of concern in software architecture to better understand the requirements for future ADLs. Component adaptation is directly related to the domain of architectural evolution, as well as others in their framework, and not enough ADLs support it.

The second related area is research in software evolution in general. Much emphasis has been placed on the role that adaptive maintenance plays in increasing the functionality of existing systems [15]. The evolver of the system, however, has direct knowledge of how the system was originally designed and constructed. The closest related work is the research by Peyman on decentralized software evolution [27]. Peyman analyzes the different ways in which software can be evolved "post-deployment" by a third-party, but the focus has been on adding components into an existing architecture, not on adapting existing components. Ben-Shaul has defined a framework for increasing the functionality of mobile code through dynamic update reflection [14]. This project defines both a component model and a powerful mechanism for adding or replacing existing functionality in a component. We are currently investigating how to use our CSL approach to help define and specify these dynamic adaptations.

Lieberherr's Demeter project [19] promotes *adaptive programming* as a technique for increasing the evolvability of a program by creating flexible interactions among objects. It is not specifically targeted towards adapting third-party components, but it is clear that components developed using Demeter would have a greater chance of being adapted. This further supports our argument that the adaptation mechanism must be built into a component for application builders to adapt the component. Techniques such as component adaptors [32] that overcome syntactic incompatibilities between components, however, do not address the need to adapt software components.

Lastly, we distinguish our work from the many efforts in defining component frameworks. Component frameworks offer a standardized platform in which components can communicate and interoperate, seemingly "plug-and-play". However, these frameworks require all components to adhere to a strict standard and set of assumptions, requiring existing components to be re-tooled to the standard. Also, there will continue to be a need for application builders to adapt components to work. A good component framework offers flexibility and tailorability, but this in no way satisfies the need to adapt existing components to meet additional requirements.

6 Conclusion

This paper has compared various approaches to adapting software components. We believe this area of research needs much investigation since current state-of-the-practice of component-based software engineering is unable to achieve its promised goals. To summarize, we have shown that third-party application builders will benefit by having ADL-level specifications of reusable software components. But more importantly, the application builder needs mechanisms that will help adapt software components for their own special needs.

We surveyed various approaches for component adaptation and collected together a set of requirements by which we compared the techniques. We carried out an evaluation of several techniques by adapting an existing component within a sample application. We plan to carry out more controlled experiments to further judge and compare the various adaptation techniques.

We showed how *active interfaces* mechanism should help increase the reusability of any software component, regardless of the underlying programming language. We showed how to combine active interfaces together with Binary Component Adaptation (BCA), to produce a powerful technique that satisfies many requirements for adaptation techniques.

Component designers should be aware that they cannot hope to produce software components that satisfy all needs, so they should find ways in which their components can be adapted as needed. Parnas observed that software should be designed to be easily extended and contracted [28]; the difficulty, of course, lies in foreseeing exactly what features will be adapted. The insight to active interfaces is that a component can be flexible enough to handle unforeseen situations. Our work is a step towards realizing the goal of having a marketplace of software components with supporting technologies aiding both application builders and component designers.

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 J. Allen, David Garlan, and James Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In *Sixth International Symposium on the Foundations of Software Engineering*, November 1998. to appear.

[3] Jesper Andersson. Reactive Dynamic Architectures. In *3rd International Workshop on Software Architecture*, pages 1–3, Orlando, FL, November 1998.

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

[5] B. Kücük and M. N. Alpdemir and R. N. Zobel. Customizable Adapters for Blackbox Components. In O. Nierstrasz, editor, *Third International Workshop on Component-Oriented Programming (WCOP'98)*, Brussels, Belgium, July 1998.

[6] Edward V. Berard. *Essays on Object-Oriented Software Engineering*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[7] Jan Bosch. Superimposition: A component adaptation technique. Technical Report TR, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, September 1997.

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

[9] 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.

[10] Object Management Group. CORBA standard. Internet site (`http://www.omg.org`).

[11] George T. Heineman. Adaptation and Software Architecture. In *3rd International Workshop on Software Architecture*, pages 61–64, Orlando, FL, November 1998.

[12] George T. Heineman. A Model for Designing Adaptable Software Components. In *22nd Annual International Computer Software and Applications Conference*, pages 121–127, Vienna, Austria, August 1998.

[13] 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.

[14] O. Holder and I. Ben-Shaul. A Reflective Model for Mobile Software Objects. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS98)*, pages 339–346, Baltimore, Maryland, May 1997.

[15] Catherine Blake Jaktman. Understanding the Evolution/Maintenance Relationship in Software Architectures. In *International Workshop on Empirical Studies of Software Maintenance WESS'97*, Bari, Italy, October 1997.

[16] Javasoft. `http://java.sun.com/products/hotjava/bean/index.html`.

[17] Ralph Keller and Urs Hölzle. Binary Component Adaptation. Technical Report TRCS97-20, Department of Computer Science, University of California, Santa Barbara, December 1997.

[18] 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.

[19] Karl Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[20] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture using Rapide. *IEEE Transactions on Software Engineering*, April 1995.

[21] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference*, Barcelona, Spain, 1995.

[22] Microsoft Corporation and Digital Equipment Corporation. The Component Object Model Specification: Draft Version 0.9, October 24, 1995.
Internet publication (`http://www.microsoft.com/oledev/olecom/title.htm`).

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

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

[25] Helgo M. Ohlenbusch and George T. Heineman. Composition and Interfaces within Software Architecture. In *1998 CASCON Conference*, Toronto, Ontario, November 1998. CD media.

[26] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[27] Peyman Oreizy. Decentralized Software Evolution. In *Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1)*, Kyoto, Japan, April 1998.

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

[29] Richard Taylor *et al.* `http://www.ics.uci.edu/pub/arch/c2.html`.

[30] 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.

[31] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. In O. Nierstrasz, editor, *ECOOP '93 Conference Proceedings*, LNCS 707, pages 36–56, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[32] 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.