An Experiment in Component Adaptation*

George T. Heineman
Helgo O. Ohlenbusch
WPI
Computer Science Department
100 Institute Road
Worcester, MA 01609 USA
(508) 831-5502
{heineman,helgo}@cs.wpi.edu
WPI-CS-TR-99-34

ABSTRACT
Adapting existing code to include additional functionality or behavior is a common theme in software engineering. As component-based software engineering achieve greater widespread use, there will be a distinct need to support such third-party adaptation. This paper describes several component adaptation techniques from 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 and suggests principles for component designers and adapters.

Keywords
Software Components, Component Adaptation

1 INTRODUCTION

A driving force behind component-based software engineering 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. One reason for this lack of success is the difficulty in integrating independently developed black-box components into a target software application. Often, new features need to be integrated with the component, resulting in extraordinary effort in wrapping or working around the fixed component. To reduce the integration and adaptation costs, we need to find strategies for adapting existing code.

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. Even after overcoming architectural mismatch [8], there is often a large problem in simply integrating the component into an application. For this paper, we assume that an application builder has somehow located a component developed by a third party and is ready to integrate the component into a target application.

We believe component-based software will only become widespread when third-party application builders can adapt components to integrate them into a target application. Consider the following definition:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. [29]

The component interface provides no insight into adaptation, and the definition does not even admit to components being subject to such adaptation. A component can be released with a special source code license allowing code modifications. The Hot Java Component from `javasoft.com`, 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.* [16]

This is hardly adequate support for adapting this component. Designing for change is an established concept in software engineering that requires the designer to consider future extensions when designing a component [27]. However, there is an understanding that the original design team will be extending the component. We suggest that a component be deployed with a rich

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

Draft of paper submitted to ICSE 2000.

description of its behavior and a mechanism that shows how to incorporate new code with the component during adaptation. In this paper we do not consider the description language further, although there are many Architectural Description Languages available for such purpose [2, 20, 21]. We focus here on several candidate adaptation techniques that may prove indispensable for component adaptation. Designing for adaptation suggests that the designer provide extra features to enable third-party application builders to adapt the component.

Adaptation, Evolution, and Customization

There is a 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 the component. It is assumed that the software engineers can freely modify the source code of the component and then the evolved component will become available for purchase and reuse. In contrast, adaptation occurs when an application builder acquires a third-party component, C, and creates a new component C_A to use within a target application. Adapted components are generally not intended for public use, and reuse of C_A will 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 each 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.

There is also a distinction between *adaptation* and *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 the arrow. The distance between two points in the figure is proportional to the difference between the components represented by those points.

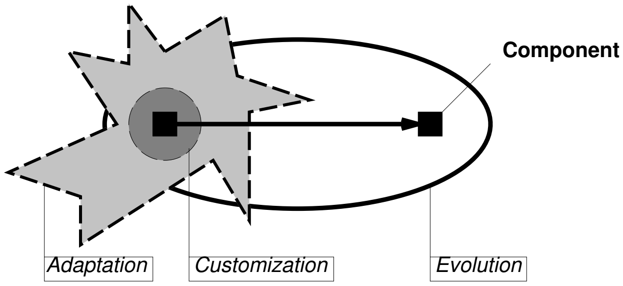

Figure 1: Perspective on Adaptation, Customization, and Evolution

The component has a pre-packaged set of options that enables customization, as represented by the small dark-gray circle; the distance 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. To maximize the reuse potential for a component, we need to understand the types of possible adaptations.

Software Architecture

Software architecture is commonly defined as a level of design that specifies the overall system structure of a software application [9]. 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 itself. An Architectural Description Language (ADL) should be used to create specifications that reflect both such aspects.

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 fully addressed by the software architecture community.

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 [23, 26]. Adaptation techniques are different since they focus on creating an adapted component C_A from

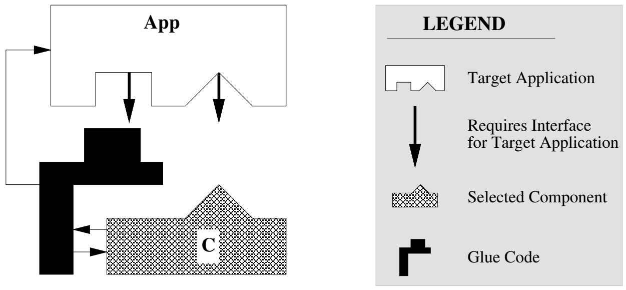

Figure 2: Adaption context

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.

Overview

In Section 2 we present a set of adaptation techniques drawn from the literature. We then describe in Section 3 the results of an experiment that applies these techniques to add two new features to an existing software component. Finally, we evaluate the adaptation techniques and the results of the experiment against a set of requirements consolidated from the individual requirements for each technique. The discussion in Section 4 is based in part on the success (and failure) documented in the experiment.

There are several contributions of this paper. First, we describe the results of an experiment that shows the use of the adaptation techniques side-by-side. Second, we describe *active interfaces* [12], a specific adaptation technique that increases the ability to adapt and integrate software components. Third, we identify a set of principles to guide designers and component adapters in software component adaptation.

2 COMPONENT ADAPTATION TECHNIQUES

The primary function of adaptation is to adapt the behavior of a component C to integrate it within a target application *app*. Consider figure 2. The target application, **App**, has an interface it expects C to support. The identified component may provide most of the expected behavior, but not enough; in the figure, there is glue code written which, in conjunction with C, provides the necessary adapted behavior. We consider the adapted component C_A to be the glue code together with the original component. Following the type framework of Medvidovic *et al.* [23], we seek a solution that creates a new component that has the same interface and implementation, but a different behavior (as seen by the target application). Once all techniques are introduced, we will proceed to the experiment.

Active Interfaces

A component interface is defined by the set of interfaces that it implements; 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. There are two phases to a method 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. 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 because no code can be written to access the private attributes or methods.

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* interface. 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 [7] 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 reveal key 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].

Referring back to Figure 2, we observe that the active interface technique builds a mechanism directly into C such that the glue code can be tightly integrated with C without requiring the application builder to directly modify C. In this fashion, one benefit of this technique is its ability to make direct behavioral changes to C without modifying its source code.

Subclassing

Inheritance is a mechanism that allows an object to acquire characteristics from one or more objects [5]. *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. In effect, the glue code in Figure 2 is the extra methods written in the various subclasses of the classes from which C was composed.

Open Source

Open Source 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. Increasingly, however, more software systems are being developed and deployed using this basic technique. Regarding the adaptation context in Figure 2, there is no need to construct glue code since component C is directly modified to create component C_A.

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 [6]. Hölzle argues that wrapping leads to poor performance as well as an excessive amount of adaptation code [30]. The Adapter and Decorator patterns from [7] 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.

3 DESCRIPTION OF THE EXPERIMENT

Experiment objectives

The experiment was designed to test the following hypotheses:

- **H1**: Can one adapt an independently-developed component to change its behavior to perform within an application.
- **SH1**: Is it possible to adapt a component without directly accessing or modifying its source code.
- **H2**: Is the *Active Interface* technique a more effective adaptation mechanism than subclassing or wrapping

Design

When designing this experiment, we had to choose between a controlled experiment vs. a field study. As presented in [28], field studies are beneficial since they generate results from the real application of software engineering techniques but the variability in the environment may make it hard to generalize the results. Controlled experiments, alternatively, enable one to create reproducible results, but then there may be difficulty in applying these results back to a real software engineering environment.

The experiment was part of an introductory graduate course in the design of software systems. There were twenty-one student subjects (5 part-time graduate students, 3 undergraduate seniors, and 13 full-time graduate students). On a five point scale (5 being the highest) the average language proficiency of the subjects was 2.7 (C++ and Java) and 4 (C); they had on average 4.1 years of C coding experience but less than 1 year of Java experience. It is safe to say that most of this experience was based on their academic experience.

The subjects were each allowed to volunteer to employ one of the adaptation techniques presented earlier. This created groups of size nine (Active Interfaces), four (Open Source), four (Subclassing), four (Wrapping). The experiment was performed in November 1999 over a seven-day period.

There are several threats to the validity of the experiment. First, there may be varying skills between the individual groups; we observed that the average C coding experience (in years) for the four groups was (4.5, 5, 4, 3.3). We decided to carry out the experiment using a component implemented in Java in part because most of the students had little experience using Java. In this way, they would be forced to focus on the design of the experimental task before jumping into the implementation. Second, the results may not generalize because of factors specific to the component used and/or the tasks carried out. We believe the results generalize because the experimental task is a common one in

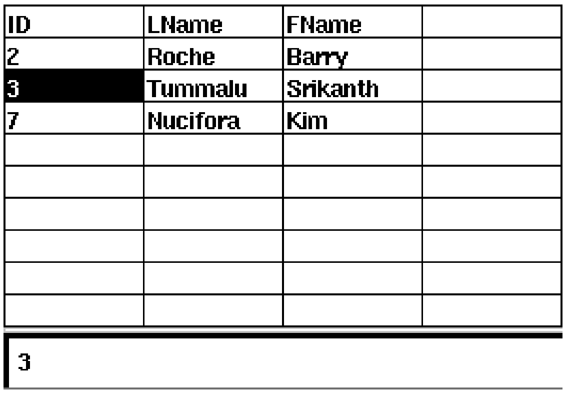

Figure 3: TableBean component

software engineering, namely, adding a new behavioral feature to an existing code unit. The component used in the experiment was composed of eight Java classes and two interfaces totaling 2778 lines of code. The subjects had used the component in an earlier assignment and so would have been familiar in its usage, though not its implementation.

Tasks
Figure 3 shows TableBean, a Java component that displays String values in a two-dimensional matrix. Using the interface `setTableValue (int c, int r, String s)`, TableBean can be programmed to display a string in the table cell at column c and row r. Figure 3 contains a sample applet, *app*, that allowed a user to enter text into a selected cell in the TableBean; the target cell is selected using the mouse.

The TableBean component was presented to the subjects as a collection of objects and classes encapsulated in a single Java package. The task of the subjects was to create a new component (that could be instantiated as tb_A) with the following additional features:

> **Problem A (Formatter)**: Adapt TableBean to allow formatting information to be associated with individual table cells. Each *format string* is of the form "$n.d$" where n specifies the count of digits to show from the integer portion of a number and d specifies the count of decimal digits to show (if more decimal digits exist they are to be truncated). If the contents of the table cell is a number, it should be displayed according to its corresponding format. For example, if the format for location $(2, 3)$ is "3.3" and the string value "3.14159" is entered for that cell, the TableBean should show "␣␣`3.141`".
>
> **Problem B (Background)**: Adapt TableBean to provide an interface for specifying a background color for a table cell if the value of the string stored in that cell is less than zero.

Adaptation Technique	Use-Case	Design	Coding
Active Interfaces	1.48	1.77	13.12
Open Source	1.25	1.44	8.63
Wrapping	1.00	1.46	11.25
Subclassing	0.77	1.75	3.00

Adaptation Technique	Success F1	Success F2
Active Interfaces	6-3	6-3
Open Source	2-2	3-1
Wrapping	2-2	1-3
Subclassing	0-4	0-4

Figure 4: Table of Results

Both problems require additional behavior to be integrated into the component.

Results
The columns in the first table of Figure 4 refer to the time spent (in hours) while performing Use-case analysis, Pseudo-code design, and then implementation. There was wide variance in the coding numbers, with some subjects giving up after only two hours while others struggled "up to 40 hours" without success. The second table shows the success ratio, with active interface being the most likely to succeed and subclassing the least likely.

4 EVALUATION

Before we evaluate the results of the experiment, we return to the adaptation techniques used and focus on the requirements proposed individually for these techniques.

Requirements
We compiled a list of requirements from the literature [6, 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_A and C, requirements on the adaptation technique, and requirements on the adaptation mechanism.

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 [6]).
2. Conservative – aspects of C there were not adapted should be accessible without explicit effort by C_A (was included as *transparent* in [6]).
3. Ignorant – C should have no knowledge of its adaptations (was included as *transparent* in [6]).
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 [6].

Adaptation technique

6. Black-box – the adaptation technique should have no knowledge of the internal implementation of C [6, 17].
7. Architectural focus – The adaptation technique should create a global description of the architecture of the target application together with a modified specification of C_A [11].
8. Framework & language 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 CORBA [10] and JavaBeans [22] components.

Adaptation mechanism

9. Configurable – the adaptation mechanism should be parameterizable; that is, it should be possible to apply a particular adaptation (the *generic part*) to many different components (the *specific part*) [6].
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].

Review

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 [4]) 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.

There were some requirements identified that we discarded for this paper. For example, Bosch required that a component technique should be reusable such that a generic adaptation to be reused, or a specific modification can be applied to multiple components [6]. 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.

In general these requirements are focused entirely on the mechanisms and the techniques and have little to do with how the subjects employed the mechanisms. As we review the experimental results, we have the following observations:

- Subclassing – The subjects that selected subclassing were frustrated in their attempts by the implementation of TableBean. One subject showed a successful solution but then admitted that he had modified TableBean to reclassify a `private` method as `protected`. The key lesson to be learned from using this technique is that component developers must ensure that `protected` methods exist, otherwise the subclass will act as a foreign class. We expect that the subjects tried so hard to get the subclass technique to work, they failed to observe that they could have implemented the subclass as a wrapper to successfully solve the adaptation.

  The result of the subclass group was surprising because this is one of the first techniques suggested when new behavior needs to be added to object-oriented software. Clearly, the component designer needs to be aware of future use of the component and must take steps to correctly classify `private` methods vs. `protected` ones. Some of the subjects complained that it was risky to override methods sight-unseen with their own implementation.

- Wrapping – The subjects that selected wrapping also had a difficult time in successfully carrying out the adaptation. In the comments, they admitted that at times they looked at the source code to TableBean so they could "understand how it functioned to better adapt it." This impulse would be reduced through better documentation and/or specification. Since Wrapping is so commonly used as the technique for integrating independent components, the results question this blind assumption. The subject that succeeded in using Wrapping to implement feature F2, for example, essentially had to re-implement the inner `paint` method for the TableBean to produce the altered behavior. This is likely to be a common situation for Wrapping when an adaptation requires only a minor tweak to existing functionality.

- Open Source – the subjects that selected open source felt they were very successful in adapting the component to include the new features. At the same time, however, they felt intimidated by the large body of code that they had to read through and understand as they made the individual changes. Several of the subjects pointed out that such direct code-level changes would require a sophisticated configuration control system to manage the multiple versions created with each successive adaptation. The Open Source subjects exhibited the least time spent in Use Cases and Design, no doubt because they felt obligated to start right in with the existing code. In their comments, the subjects stated that they would prefer an adaptation technique that did not require such deep involvement with the source code of the component.
- Active Interface – the subjects that selected active interface spent the longest time in programming the assignment, yet they had the highest success ratio on both features. This occurred even though feature F2 was more difficult to introduce into TableBean than feature F1.

  The comments revealed some of the reason for their success. One subject appreciated the ability to debug the component by inserting `printing` callbacks at key methods; in this way, he was able to better understand the interaction between the various internal methods. One subject liked the incremental behavior of adding methods one by one to the adapted component. In general, they felt there were many aspects in common with subclassing and wrapping.

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 [25]. 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.

DRADEL [23] shows how a system evolves to substitute new components that support new requirements. However, it doesn't show whether the new component C' is derived from C or simply written from scratch. DRADEL can be used to ensure that an adaptation can be used within the given system and thus provides us with an instant type-checking regiment for the adaptation techniques proposed in this paper so long as there is an architectural focus.

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 [31] 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 retooled 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 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 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.

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 [27]; 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.

The experiment revealed certain behavior among those adapting software components. First, developers do not want to access the source code directly – this runs counter to the increasing trend towards open source software projects. Second, developers need more detailed descriptions of the interface for a component than simply the API. Often the largest time spent during the adaptation was in trying to figure out exactly what the code did.

Thus when thinking about the definition of a software component, we clearly need more documentation that describes how to use the component; and, by extension, how to adapt the component. By analogy consider the difference between high-precision machine manufacturing vs. building a product (i.e., book shelves) at home. With modern manufacturing, parts are milled to within one-thousandth of an inch and then precisely placed by expensive, specially built factories of machines. The average adult can assemble furniture at home without specialized tools because (1) only standard tools are needed for the assembly otherwise they are provided; and (2) the way in which the parts are assembled has built-in degrees of freedom to allow construction without costly precision.

If we are to achieve component-based software engineering, it must be easy to construct systems from components, and we believe focusing on component adaptation will increase the use and reuse of software components.

The authors would like to acknowledge the hard work of the students in CS509 as they participated in the experiment.

REFERENCES

[1] G. D. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[2] R. J. Allen, D. Garlan, and J. 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] J. Andersson. Reactive Dynamic Architectures. In *3rd International Workshop on Software Architecture*, pages 1–3, Orlando, FL, November 1998.

[4] 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.

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

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

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software.* Addison-Wesley, Reading, MA, 1995.

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

[9] D. Garlan and M. Shaw. *An Introduction to Software Architecture*, volume I of *Advances in Software Engineering and Knowledge Engineering.* World Scientific Publishing Company, New Jersey, 1993.

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

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

[12] G. 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] G. T. Heineman and G. 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] C. B. 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] R. Keller and Urs Hölzle. Binary Component Adaptation. Technical Report TRCS97-20, Department of Computer Science, University of California, Santa Barbara, December 1997.

[18] G. Kiczales, J. Lamping, C. Lopes, C. Maeda, A. Mendherkar, and G. Murphy. Open Implementation Design Guidelines. In *19th International Conference on Software Engineering*, pages 481–490, May 1997.

[19] K. 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] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference*, Barcelona, Spain, 1995.

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

[23] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *21st International Conference on Software Engineering*, pages 44–53, May 1999.

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

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

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

[28] L. Prechelt and B. Unger. A Series of Controlled Experiments on Design Patterns: Methodology and Results. In *Proceedings Softwaretechnik '98 (Softwaretechnik-Trends)*, pages 53–60, September 1998. `http://wwwipd.ira.uka.de/~prechelt/Biblio/patseries_st98.ps,g% z`.

[29] C. Szyperksi and C. Pfister. Workshop on Component-Oriented Programming, Summary. in mühlhäuser M. (ed.) *special issues in object-oriented programming - ecoop'96 workshop reader.*

[30] 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.

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