An Evaluation of Component Adaptation Techniques

George T. Heineman
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609, USA
heineman@cs.wpi.edu
WPI-CS-TR-99-04

Abstract:

One of the many difficulties in making Component-Based Software Engineering (CBSE) a reality is that software components may require adaptation when constructing applications from COTS components. We survey the literature to discover various approaches to component adaptation and evaluated these approaches against a set of requirements for component adaptation mechanisms. We also discuss differences between adaptation of software components and extension of object-oriented classes.

---

1. Introduction

The closing sentence of a recent report on the current state of CBSE states that the growing use of external components will demand improvements in how components are documented, assembled, adapted, and customized [3]. This position paper addresses the issue of adaptation.

We have argued in [8, 6, 7, 9] that a true component marketplace will only exist when application builders can *adapt* software components to work within their application. For this position paper, we surveyed the literature for different approaches to adapting software components. Our primary contribution is to show that component adaptation is a highly relevant problem to CBSE. Component adaptation is sufficiently different from software evolution that it requires new techniques and certainly new understanding to solve

its challenges.

We first motivate the need to adapt third-party COTS components after they have been designed, implemented, and made available for purchase. We then discuss the differences between adaptation of components and adaptation of object-oriented programs. We then evaluate various approaches to component adaptation against a set of requirements for adaptation mechanisms.

2. Motivation

An application builder has designed and partially implemented a software system using several reusable in-house software components. The builder finds an externally available third-party software component that satisfies some desired functionality or behavior. Because there are such difficulties in accurately specifying software, however, the builder is not totally sure that the component will completely perform all the desired tasks; in fact, the component may contain additional unneeded features that are incompatible with the original system. There is enough evidence, however, to install the component and try to use it, so the builder proceeds.

The application builder must then integrate the component into the original system; this task may be complicated by syntactic incompatibilities between the interfaces that need to communicate with one another. The builder can either a) modify the original system to overcome these incompatibilities; b) modify the component; or c) introduce a component adaptor [19] or some other *wrapper* between the system and the component. As Hölzle shows, however, there are complications when multiple components must communicate with each other while they are contained within some form of wrapper object [17].

Once all syntactic problems are overcome, however, there will likely still be situations where the functionality or behavior of the component needs to be modified according to the needs of the application builder. Component designers cannot, of course, foresee every possible use of their component, and they cannot respond to every modification request from their users. We need to create mechanisms, therefore, whereby application-builders can easily adapt third-party components without requiring knowledge of the source code.

As more and more third-party components are added to the application - or when an application is constructed entirely from such components - the only solution that will scale is one that minimizes the effort to make modifications to the original application and to adapt the software components.

2.1 Adaptation, Evolution, Customization

The players in this drama are the component designer and the application builder. We make the distinction between *software evolution*, where component designers modify the software component they designed, and *adaptation*, where an application builder adapts a third-party component for a (possibly radically) different use. If the component designer were requested to adapt a component, the designer would likely select a minimal set of changes because of direct knowledge of the component. The application builder does not have this advantage, nor will the builder be able to acquire this knowledge simply from the source code and documentation. The application builder, thus, needs help to successfully adapt components. We also differentiate adaptation from *customization*; an end-user customized a software component by choosing from a fixed set of options (such as OIA/D [11]). An end-user adapts a software component by writing new code to alter existing functionality or behavior.

2.2 Differences between adapting components and classes

Object-Oriented Design (OOD) embodies the principle of *design for change*, a design principle first stated by Parnas [15] that encourages Software Engineers to modularize code to minimize the impact of future changes. OOD has two mechanisms that serve this purpose. First by designing classes with a public interface and private implementation, a class supports *information hiding*. The class designer can insulate the clients of the class from the internal implementation, which usually changes more frequently than the interface definition. Second, *inheritance* is a mechanism by which an object acquires characteristics from one or more other objects [1]. Inheritance can be classified as *essential*, referring to the inheritance of behavior or an externally visible characteristic, or *incidental*, referring to the inheritance of part, or all, of an underlying implementation of a more general object. Object-oriented designers learn early on that incidental inheritance, done strictly for the purpose of reusing existing code, leads to poor design.

In the Software Architecture literature, inheritance is a modeling vehicle used by various Architectural Description Languages (ADLs), such as ACME [4] to specify when *interface inheritance* occurs (there are exceptions, notably the use of object-oriented typing as seen in [18]). We argue, however, that inheritance should not be used to create new components from parts of old components.

One of the major differences between CBSE and OO is that engineers wishing to adapt an existing object-oriented program must perform the difficult task of understanding (often complex) class hierarchies. There is a tacit assumption with object-oriented technology that the designer of the system and the maintainer/adapter are one and the same. If this is not the case, however, the adapter must determine the set of classes to modify to make the change such that the original integrity is not broken. Often, additional leaf classes are added to a class hierarchy to avoid changing the original class structure when it would have been better to make modifications to existing classes. We seek to find ways for an application builder to adapt a component with only knowledge of its documented interface.

3. Requirements for Component Adaptation Techniques

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 [2, 8, 10]. 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.

3.1 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* ([8], was *transparent* in [2]).
2. Conservative - aspects of *C* there were not adapted should be accessible without explicit effort by C_A (was included as *transparent* in [2]).
3. Ignorant - *C* should have no knowledge of its adaptations (was included as *transparent* in [2]).
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 [10].
5. Composable - C_A should itself be open to future adaptations; it should be straightforward to compose together a set of desired adaptations [2].

3.2 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*) [2].
7. Black-box - the adaptation technique should have no knowledge of the internal implementation of *C* [2, 10].
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 [6]; 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 [12], CORBA [5], and JavaBeans [13] components.

3.3 Adaptation mechanism

10. Embedded - the adaptation mechanism must exist within *C* before *C* can be adapted into C_A [8].
11. Language independent - the adaptation mechanism must not be dependent upon the language used to implement *C* [8]; this requirement also pertains to the adaptation technique.

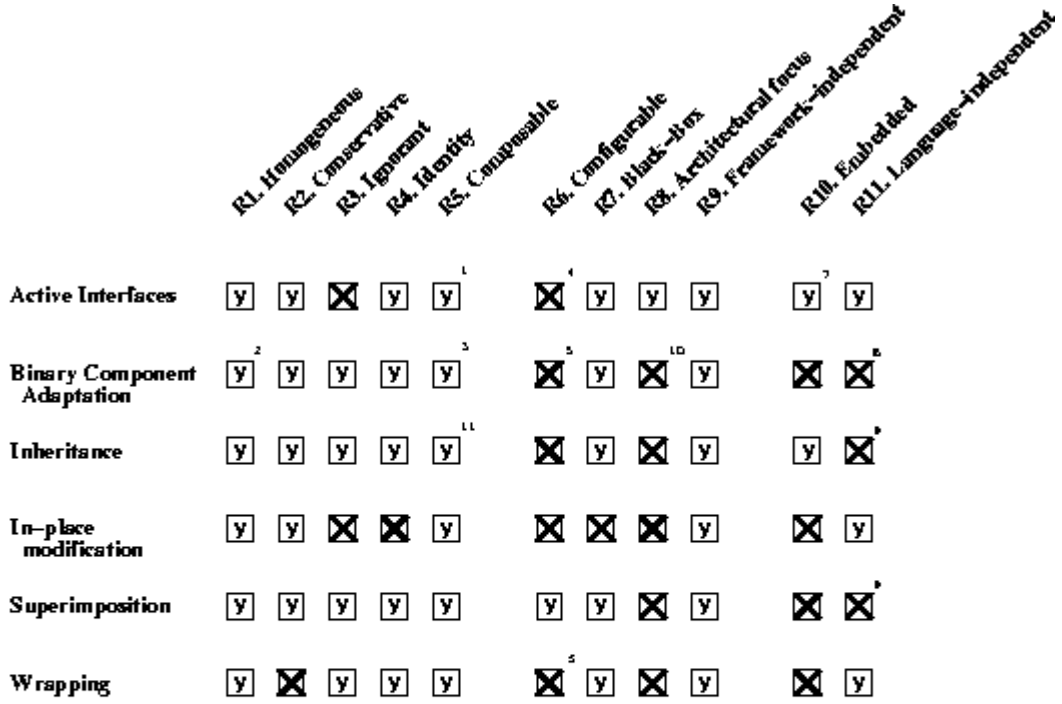
It may not be possible for an adaptation mechanism to satisfy each requirement, since these requirements are drawn from disparate sources. There is no clear indication on how to prioritize these requirements. Note that some of the requirements in Figure 1 are partly contradictory: **R3** and **R10**, for example. Others are strongly related, such as **R1-R3**. By evaluating component adaptation mechanisms against these requirements, we can determine those requirements that are the most useful.

3.4 Adaptation as a facet of Integration

Incorporating third-party software components will always require integration, but there is not enough emphasis on the necessary adaptation that must take place. Again, we differentiate adaptation from *customization* whereby the customer simply selects from a pre-determined set of options. Some have proposed wrapping or mediation as integration mechanisms, but these only partially satisfy the integration aspects, and do not solve the problems of adaptation.

3.5 Architectural evolution

Figure 1 lists only those approaches that adapt a software component to create a new component. There are several research efforts concerned with *Architectural Evolution*, namely the addition, removal, or replacement of components, connectors, or changes to the configuration of components and connectors. Some examples are ArchStudio [14] and Simplex [16]. There are also different efforts towards creating software systems whose architecture can change dynamically at run-time to adjust as needed to changing circumstances; these are dynamic versions of architectural evolution.



1. The callback methods can themselves be composed together
2. Must execute component within modified Java 1.1.8 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. It is possible to insert an active interface into certain components if the source code is unavailable[9]
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 1.1.8
9. Applicable only for components written in an object-oriented language
10. Can be integrated with architectural focus
11. Over time, may become impossible to further adapt a class through inheritance as class hierarchies become increasingly tangled

Figure 1: Comparison matrix

4. Discussion

The comparison matrix in Figure 1 reveals various correlations between the requirements and mechanisms. There is strong agreement that requirements **R1,R5,R9** are suitable for adaptation mechanisms. This reflects, perhaps, the fact that these requirements relate to structural issues. Requirements **R6** and **R8** only have one proponent each, namely Superimposition [2] and Active Interfaces [8], but this is simply a way these techniques differentiate themselves from others in the literature.

The sharpest division on these requirements (where at least two techniques vote positive and two vote negative) are embedded (**R10**) and language-independence (**R11**). Note that these requirements both refer to the type of adaptation mechanism employed by the adaptation technique. Most component designers select a programming language with only a passing attention to future adaptation needs. Object-oriented programming languages, however, automatically enable adaptation because inheritance is built-in. The Active Interface approach [8] only requires small hooks embedded into a component and thus is a viable adaptation technique. Currently, component implementation is driven by the selection of a particular component framework, such as JavaBeans [13], Component Object Model (COM) [12], or CORBA [5]. A component belonging to one of these frameworks must embed the appropriate mechanisms to belong to the framework, so in a sense embedding need not be a controversial topic.

When considering the adaptation techniques themselves, in-place modification clearly fails; with small differences, however, the other techniques are more related than one might realize at first. We have carried out a small experiment, described in [9], in evaluating the use of each of these techniques (except Superimposition) in solving an adaptation problem. We are currently designing more rigorous experiments to understand how to better aid application builders when they need to adaptation existing components.

5. Conclusion

This evaluation survey provides an overview of existing component adaptation techniques and provides a good starting point for discussing the nature of component adaptation mechanisms. This material belongs in various sections of the proposed Strawman outline for the workshop. Under the Technology supporting CBSE (Section 3), reusable components must be discussed within the framework of how application builders will adapt them. Integration technologies should not be limited to Run Time support; rather it should include such static mechanisms as discussed in this paper. Finally, from a philosophical perspective, it is important to differentiate software reuse (which traditionally has been a means of reusing functional code libraries or frameworks) from reusable components (which brings in the notion of adapting behavior).

References

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

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

3 Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37-46, September 1990.

4 David Garlan, Robert T. Monroe, and David Wile. ACME: An architectural description interchange language. In *1997 CASCON Conference*, pages 169-183, Toronto, Ontario, November 1997.

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

6 George T. Heineman. Adaptation and Software Architecture. In *3rd International Workshop on Software Architecture*, pages 61-64, Orlando, FL, November 1998.

7 George T. Heineman. Composing software systems from adaptable software components. In *DARPA/OMG Workshop on Compositional Software Architectures*, Monterey, CA, January 1998. `http://www.objs.com/workshops/ws9801/report.html`.

8 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.

9 George T. Heineman and Helgo Ohlenbusch. An Evaluation of Component Adaptation Techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, February 1999.

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

11 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.

12 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`).

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

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

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

16 L. Sha, R. Rajkumar, and M. Gagliardi. Evolving dependable real-time systems. In *IEEE Aerospace Applications Conference*, pages 335-346, New York, NY, 1996.

17 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.

18 Ian Welch and Robert Stroud. Adaptation of connectors in software architectures. In *Third International Workshop on Component-Oriented Programming (WCOP'98)*, Brussels, Belgium, July 1998.

19 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.