Adaptation of Software Components

George T. Heineman
Computer Science Department
WPI
Worcester, MA 01609

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 the adaptation of software components that may be required when constructing applications from COTS components. We surveyed 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 [5,6,7,8] 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 [18] 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 [9].

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). 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 [17]). We argue that inheritance should not be used to create new components from parts of old components.

However, 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. In particular, 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 the hierarchy to avoid changing the original class structure when it would have been better to make modifications to existing classes. There is, thus, a tacit assumption with Object-Oriented technology that the designer of the system and the maintainer/adapter are one and the same. 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

Figure 1 contains the requirements for component adaptation techniques that we compiled from various articles [8,2,10]. 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: **R2** and **R4**, for example. Others are strongly related, such as **R5** and **R7**. By evaluating component adaptation mechanisms against these requirements, we can determine those requirements that are the most useful.

1. *Black Box* - The person adapting the component should only need to understand the interface to the component.
2. *Transparent* - the client of the adapted component, as well as the component itself, should be unaware of

the adaptation between them.

3. *Flexible* - it should be possible to induce a wide range of adaptations (functional as well as behavioral).

4. *Embedded* - The adaptation mechanism should be built into the component.

5. *Language Independence* - the adaptation mechanism should not specifically depend on any one programming language.

6. *Composable* - the adapted component should continue to be composable with other components; the actual adaptation should be composable with other adaptations.

7. *Reusable* - one should be able to reuse the code written to adapt a component.

8. *Architecturally Aware* - A component-based application should have some global concept of architecture, and the specification and/or implementation of the adaptation should be visible at this architectural level.

9. *Configurable* - An adaptation mechanism should be capable of applying the same particular adaptation (a generic part) to a particular set of target characteristics (the specific parts). Synonymous with *Repeatable* or *Template-drive.*

Figure 1a: Requirements for component adaptation mechanisms

Adaptation Mechanism	**R1**	**R2**	**R3**	**R4**		**R5**	**R6**	**R7**	**R8**	**R9**
Metaobject Protocols [17]	Yes	Yes	Yes	Yes[1]		No[1]	Yes	Yes	Yes	Yes
Active Interfaces [5]	Yes	Yes[6]	Yes	Yes		Yes	No[5]	Yes	Yes	No
Superimposition [2]	Yes	Yes	Yes	Yes		No[4]	Yes	Yes	No	Yes
Binary Component Adaptation [10]	Yes	Yes[9]	Yes	No		No[3]	Yes[8]	No	No	No
Inheritance[2]	Yes	Yes	Yes	Yes		No[4]	No	No	No	No
Open Implementation [11]	Yes	No	No	Yes		Yes	No[7]	No	No	No
Copy-Paste[2]	No	Yes	Yes	No		Yes	No	No	No	No
Wrapping[2]	Yes	No	No	No		Yes	Yes	No	No	No

[1] Reflection is required.

[2] These basic techniques are carefully analyzed in [2]. See [8] for supporting evidence.

[3] BCA theoretically will work on object code compiled from any high-level language, but there are serious obstacles to such efforts; the current prototype works with Java.

[4] The language must be object-oriented.

[5] Active interfaces can be composable if the individual adaptations themselves are programmed as such.

[6] Once this mechanism is built into the component, the component is unaware of its workings.

[7] All adaptations are chosen from pre-selected implementation strategy code; there is an opportunity for a component to replace such strategy code as part of the adaptation, but the authors admit the difficulty of such a task [11].

[8] Although not discussed as such in [10], it should be straightforward to apply BCA to a previously adapted component.

9

[9] The Java compiler was modified to transparently resolve client references to adapted code.

Figure 1b: Comparison Matrix

3.1 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.

3.2 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.

4. Discussion

The comparison matrix in Figure 1 reveals various correlations between the requirements and mechanisms. There is strong agreement that requirements **R1-R4** are suitable for adaptation mechanisms (as shown in the upper left quadrant). This reflects, perhaps, the fact that requirements **R1** (Black-Box), **R2** (Transparent), and **R4** (Embedded) relate to structural issues. **R3** (Flexibility) is perhaps a poor measure since it is subjective (for example, the Wrapping mechanism is marked as not being flexible based on its inability to adapt the internal behavior of the component being wrapped).

The lower right quadrant of the matrix strongly agrees that the various mechanisms (Inheritance, Wrapping, and Copy-Paste) are not satisfactory. This is not surprising, considering that these have been used by [8,2] as a *control* to compare adaptation mechanisms. Open Implementation was developed to help the designer of a component create a highly customizable component, rather than one that would be easy to adapt.

The upper right quadrant contains mixed results, and instead of being used to globally select which adaptation mechanism is "best", these results should be used to guide component designers to select the adaptation mechanism best suited to their own concerns. For example, Language Independence (**R5**) is not an agreed upon requirement, but this should not affect component designers since they will select adaptation mechanisms best suited for the language in which they implement their component. It is unlikely (although still probable) that a component designer would select a programming language for the ease in which a third-party application builder would be able to adapt the component.

There is a direct correlation between **R2** (Transparent) and **R3** (Flexibility) that is certainly not evident simply from the list of requirements. One conclusion to be drawn is that mechanisms are most flexible when they do not introduce additional coupling between the adapted component and the target application. A strong correlation exists between **R7** (Reusable) and **R8** (Architecturally aware). This is interesting since it suggests that adaptation mechanisms that focus on the global architecture are able to reuse local adaptations in powerful ways. A negative correlation exists between **R5** (Language Independence) and **R9** (Configurable) which suggests more research needs to be performed to find ways to make adaptation mechanisms more generic.

5. Conclusion

This evaluation survey provides an interesting overview of the state-of-the-art in component adaptation and provides a good starting point for discussions on 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 bring in the notion of adapting behavior).

Acknowledgements

This work is sponsored in part by National Science Foundation grant CCR-9733660.

References

1. Edward V. Berard, Essays on Object-Oriented software engineering, Volume I, Prentice Hall, 1993.
2. Jan Bosch, Superimposition: A component adaptation technique, submitted for publication, http://bilbo.ide.hk-r.se:8080/~bosch/papers/compadap.ps
3. Alan W. Brown and Kurt C. Wallnau, The current state of CBSE, *IEEE Software*, Vol. 15, No. 5, September/October 1998.
4. David Garlan, Robert T. Monroe, and David Wile, ACME: An architecture description interchange language, In *Proceedings of IBM Centre for Advanced Studies Conference (CASCON'97),* pp. 169-183, Ontario, Canada, November 1997.
5. George. T. Heineman, A model for designing adaptable software components*, Twenty-second International Conference on Computer Software and Applications Conference (COMPSAC)*, pp. 121-127, Vienna, Austria, August 1998.
6. George T. Heineman, Adaptation and software architecture, *Third International Workshop on Software Architecture*, Orlando, Florida, November 1998.
7. George. T. Heineman, Composing software systems from adaptable software components, *DARPA/OMG workshop on Compositional Software Architectures*, Monterey, California, January 1998.
8. George T. Heineman and Helgo Ohlenbusch, Towards a theory of component adaptation, Technical report WPI-CS-TR-98-20, Worcester Polytechnic Institute, *submitted for publication*.
9. Urs Hölze, Integrating independently-developed components in object-oriented languages. In O. Nierstrasz, editor, *Proceedings ECOOP'93, LNCS 707,* pp. 36-56, Kaiserslautern, Germany, Springer-Verlag, July 1993.
10. Ralph Keller and Urs Hölze, Binary component adaptation, Technical report TRCS97-20, University of California, Santa Barbara, December 1997.
11. Gregor Kiczales *et al.* Open implementation design guidelines, in *19th International Conference on Software Engineering,* pp. 481-490, May 1997.
12. Nenad Medvidovic, Peyman Oreizy, J. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. SIGSOFT?96. pp 24-32. San Francisco, CA, October 1996.
13. Peyman Oreizy, Decentralized software evolution. In *Proceedings of the International Conference on the Principles of Software Evolution (IWSPE 1),* Kyoto, Japan, April 1998.
14. Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor, Architecture-based runtime software evolution. The Proceedings of *The International Conference on Software Engineering 1998 (ICSE'98)*. Kyoto, Japan, April 19-25, 1998.
15. David L. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, Vol. 5, No. 12, pp. 1053-1058, December 1972.
16. L. Sha, R. Rajkumar, and M. Gagliardi, Evolving dependable real-time systems, *IEEE Aerospace Applications Conference*. New York, NY, pp. 335-346, 1996.
17. 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.
18. 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.