Composing Software Systems from Adaptable Software Components
WPI-CS-TR-97-9
George T. Heineman
Worcester Polytechnic Institute
Worcester, MA 01609

`http://www.cs.wpi.edu/~heineman`

Abstract

The construction of software systems from pre-existing, independently developed software components will only occur when application builders can adapt software components to suit their needs. Our ADAPT framework [Hein97] supports both component designers in creating components that can easily be adapted, and application builders in adapting software components. We propose that software components provide two interfaces -- one for behavior and one for adapting that behavior as needed. In this position paper, we outline some requirements for composing software systems from components and suggest that adaptation be recognized as a significant factor.

1 Introduction

The goal of constructing software applications from reusable software components is proving to be very challenging. We believe that adapting software components for use by a particular application is a key enabling technology towards realizing this goal. Using a software component in a different manner than for which it was designed, however, is challenging because the new context may be inconsistent with implicit assumptions made by the component. Techniques such as component adaptors [Yell97] that overcome syntactic incompatibilities between components do not address the need to adapt software components.

Our focus is on supporting both component designers and application builders: the designers will be aided in creating components that can easily be adapted (thus increasing reuse), and for the first time, application builders will have mechanisms for adapting software components. By design, black box components often only allow minimal customization and are reusable if they exactly match a particular need in an application. However, the use of the component is also heavily dependent on the ability for application builders to adapt the component for use in different applications.

We make a distinction between software *evolution*, where the software component is modified by the component designer, and *adaptation*, where an application builder adapts the component for a different use. We also differentiate adaption from *customization*; an end-user customizes a software component by choosing from a fixed set of options (such as OIA/D [Kicz97]). An end-user adapts a software component by writing new code to alter existing functionality.

1.1 ADAPT

The goal of the ADAPT framework is to increase the feasibility of component-based software development by showing how to design adaptable software components. The main idea is that component designers must provide mechanisms that allow application builders to incorporate and adapt these components into their application. Two research directions for the ADAPT project that are relevant to the discussion of compositional software architecture are:

- A Component specification language for specifying the interface of a component and how it is adapted.
- Active interfaces

1.1.1 Component Specification Language

```
component Spreadsheet {
  implements Serializable,
             SpreadsheetListener;

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

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

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

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

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

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

  // SpreadsheetListener Interface
  void handleSpreadsheetEvent(SpreadsheetEventObject);
}
```

The ADAPT (Architectural Description of adAPTable components) language is used to describe the interface for a component and its adaptations. If the component is written in a reflective language, such as Java, then the specification for a component can initially be generated directly from the component. The above figure describes one such specification generated for a Spreadsheet Bean. This Bean supports a set of spreadsheet services, allowing clients to set individual cell values (through the Value property), and allows external client Beans to become listeners for refresh requests as new values are added and computed. An ADAPT description provides a convenient place to specify adaptations to the component, as we now describe.

1.1.2 Active Interfaces

The interface must play a greater role in helping application builders adapt the component. The component interface is more than a syntactic description of the method invocations accepted by the component. As defined in [Abow95], components are active computational entities whose interfaces defines methods to invoke, events to receives and/or send, or complex access protocols. An *active interface* decides whether to take action when a method is called, an event is announced, or a protocol executes. There are two phases to all interface requests: the "before-phase" occurs before the component performs any steps towards executing the request; the "after-phase" occurs when the component has completed all execution steps for the request. These phases are similar to the Lisp advice facility described in [Rama97]. A standard way to alter the behavior of a component is to interpose an entity to intercept messages and/or events. Because such adaptation is likely to occur, the component should provide an interface for this purpose.

Suppose that the application builder wishes to modify the Spreadsheet Bean in Figure 1 so that it generates an event whenever the particular value of a cell changes (for example, because the cell contains a calculated formula), not just when the contents change; for example, changing a cell from *"(+ 2 3)"* to *"(* 5 1)"* changes the contents, but not the value. The application builder could modify the component directly (which we are trying to avoid) or filter out messages from the Spreadsheet (but this would require the client to store duplicate values to detect differences, and is space/time inefficient). We use an active interface to insert a *before-evaluate* function that has the Spreadsheet component record the value of the cell before its update and an *after-evaluate* function that compares the new value against the old; if the values are different, the *after-evaluate* function generates the appropriate notification. These functions would then be incorporated into the object *ss*, the instantiation of the Spreadsheet component. With this scheme, different component objects from the same class can be adapted in different ways, offering even more flexibility to the application builder.

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

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

Active interfaces are different from the pre-packaged implementation strategies of OIA/D [Kicz97]. OIA/D sketches a solution showing how the client can provide their own implementation strategy, but typically an entire method for a component is replaced. Our approach is more fine-grained, allowing adaptation to occur when needed. We do not violate the encapsulation of the component, since the methods invoked within the active interface do not directly access private information in the component. Thus the component designer has great flexibility, and can place the responsibility for correctness on the application builders that adapt the component.

1.2 Context

Our ADAPT framework [Hein97] is independent of the particular programming language and architectural style, and is thus widely applicable. For our initial prototype, we have chosen to use the JavaBeans [SUN97] software component model. A Java *Bean* is a reusable software component written in Java that can be manipulated visually in a design environment, such as the sample Bean Developers Kit (BDK) shipped with the initial release of JavaBeans. BDK allows application builders to instantiate a collection of Beans that communicate with each other using *events*. The JavaBeans event model provides a convenient mechanism for components to propagate state change notifications to one or more listeners. Each Bean contains a set of state properties (i.e., named attributes) and BDK allows application builders to customize a Bean by modifying its properties.

We are interested in extending the JavaBean component model to distributed applications. In particular, we are designing a message framework called SOWER on which a distributed application composed of beans can be built. SOWER is based on the Event-Based Software Integration (EBI) described in [Barr96]. The key features of SOWER are:

- It is integrated with the BDK.
- JavaBean components can be used as is without modification.
- Connector code is automatically generated as needed to deliver JavaBean events between remote components.
- The registration of the communication between Beans is separate from the actual delivery of Bean events.

In this way, we can experiment with constructing distributed applications from components and investigate the evolvability of such software systems.

2 Requirements

For this position paper, we identify the following requirements for compositional software architectures. We discuss these requirements in the context of JavaBeans, but they apply equally regardless of programming language.

2.1 Core Requirements

1. *Separation of* ***physical*** *composition from* ***logical*** *composition*. JavaBeans provides a convenient abstraction for communication between two components, and the distribution mechanism must maintain this separation.
2. *Using JavaBeans without design modification*. The JavaBeans standard is gathering support in industry, so any approach to building distributed applications must be compatible with the basic Bean model.
3. *Support for dynamic configuration*. The framework enabling communication between distributed Beans must allow the application builder to deploy and move Beans between each *site*. The virtual communication between Beans should be flexible enough to be reconfigured when Beans move.

2.2 Advanced Requirements

As part of our efforts towards building adaptable software components, we feel the following requirements are essential for realizing the goal of building distributed applications from software components.

1. *In Situ Adaptation*. When adapting the behavior of a class *C*, object-oriented design methodologies suggest that a new subclass *SC* be created that extends *C* to create the new, desired functionality. This approach does not work when we wish to adapt the behavior of a component. First, even though a component may syntactically be equivalent to a class (as in JavaBeans), inheritance is ill-suited for such adaptation. Consider a component constructed using the Facade design pattern [Gamm95]; creating a subclass of the *Facade* class complicates the design. Second, once a

component is deployed, the application builder should be able to adapt the component by supplying new code to be integrated into the component; inheritance is strictly a compile-time mechanism.

3. Conclusions

In this paper, we outlined some requirements for compositional software architectures. We are focused on creating design methods and implementation mechanisms that allow application builders to adapt software components in their applications. We showed that component models must provide some mechanism for adaptation, and we introduced active interfaces for this purposes. We sketched the ADAPT and SOWER frameworks, and described their usefulness in constructing distributed applications from software components.

References

1. **[Abow95]** *Formalizing Style to Understand Descriptions of Software Architecture*. ACM Transactions on Software Engineering and Methodology, 4(4):319-364, October 1995.
2. **[Barr96]** *A Framework for Event-Based Software Integration*. Daniel Barrett, Lori Clarke, Peri Tarr, Alexander Wise. ACM Transactions on Software Engineering and Methodology, 5(4):378-421, October 1996.
3. **[Gamm95]** *Design Patterns: Elements of Reusable Object-Oriented Software*. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley, 1995.
4. **[Hein97]** *A Model for Designing Adaptable Software Components*. George Heineman. Submitted for Publication.
5. **[Kicz97]** *Open Implementation Design Guidelines*. Gregor Kiczales, et al. 19th International Conference on Software Engineering, pages 481-490, May 1997.
6. **[Rama97]** *A Emacspeak: A Speech-Enabling Interface*. Dr. Dobb's Journal, 22(1):18-23, September 1997.
7. **[SUN97]** *JavaBeans 1.0 API Specification*. Sun Microsystems, Inc. December 4, 1996.
8. **[Yell97]** *Protocol Specification and Component Adaptors*. Daniel Yellin and Robert Strom. ACM Transactions on Programming Languages and Systems, 19(2):292-333, March 1997.