CUSTOMIZABLE ADAPTERS FOR BLACKBOX COMPONENTS

B. Küçük, M. N. Alpdemir and R. N. Zobel
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, U.K.
Tel: +44 161 275 6210/6189
Fax: +44 161 275 6204
E-mail: {kucukb, alpdemim, rzobel}@cs.man.ac.uk

Abstract

An approach to arrangement of component responsibilities in a component-based software system is proposed, in which black-box components constitute the major building blocks and perform the main part of computation while whitebox *adapter components* enhance their interoperability. We recommend a technique in which components contribute to the construction of the adapters in a structured way. A component supplies one or more customizable classes to be used in implementation of the portion of the adapter associated with that component. Through various examples we investigate the advantages and feasibility of using customizable adapters that compensate for the inevitable gaps between interacting components' interfaces. Their usage allows a black-box component to be designed as concise as possible, i.e. providing a (minimal) interface that is confined to its functionality, and assuming the most convenient interface from the components it makes use of. An important outcome of this conciseness is improved reusability.

1 Introduction

The idea of building complex software systems from ready-made components can apparently lead to substantial reduction in the engineering time and effort — hence the great amount of research devoted to this goal [10, 7]. The ultimate aim is to make software construction almost a process of connecting components through their plugs. The co-operation of components is, ideally, achieved by their adherence to some standard protocols.

The idea is elegant, and has proven successful in various engineering disciplines. However, due to the limited spectrum of available components, in a component-based industry the designs are driven not only by the requirements, but also by availability. This has a particularly adverse effect in the field of software where the structure of the information to be exchanged between components, and the exchange mechanism can take exceptionally complex forms: It is not feasible to produce binary off-the-shelf components to suit the requirements of every possible application in an optimum manner, and similarly it is impossible to develop protocols to describe every type of component-set co-operation. Consequently, extra coding is usually necessary to compensate for the interface mismatches. A straightforward solution is to access an incompatible component through another component, called an *adapter* which converts its interface into a form desirable by its client [3]. Several research efforts are devoted to devise a formal approach to adapter construction [1, 11, 6].

The *adapting* technique can be utilized to give the application builder a high degree of freedom, despite the finite variety of ready-made components. This benefit, like any other, comes at some cost, that is a small amount of programming. In this paper, we accept that adaptation is inevitable in most cases [4, 1], and to try to do it in a way that maximizes efficiency and reusability. An important issue is determining what type of functionality should be implemented by components and what aspects should be left for implementation in their adapters, to gain maximum reusability. Another issue is the development of adapters. The degree to which the suppliers of ready-made components can facilitate this task is discussed.

2 Adapters for Software Components

Engineering systems are almost always connected through some form of adapter such as gauges, digital to analog or analog to digital converters, amplifiers, rectifiers, buffers, etc. In the computer hardware industry, for a specific example, a graphics adapter is used to isolate a monitor from the main board that drives it. The monitor's interface does not make assumptions specific to a particular type of main board. Instead, all the architecture-specific properties are placed in the adapter. Thus it becomes possible to connect a monitor to many different computer architectures; all that has to be done is to use the appropriate adapter. Similarly adapters make it possible to connect various types of monitor to a computer.

One difficulty that the developer of a novel computer architecture faces is to make an adapter for each peripheral device as the manufacturers of devices produce their adapters only for common architectures. It would have been very useful if the manufacturers were able to produce generic, highly customizable adapters. This is not easy, because the degree of flexibility that can be built into hardware is very limited due to it being *hard*. Fortunately the idea can be applied to the adapters of *soft*ware.

In general, as observed above, an adapter module is a convenient place to keep the client- or access-related aspects of a component. Thus, the component's characteristic properties and behaviour can be kept separate, in binary form, as a *black-box* [10] which is accessed only through its interface. The blackness feature is essential for complex components as it ensures that their implementation can be modified or enhanced without affecting their clients. The adapter, on the contrary, is — in this context — preferably something alterable/customizable by the people who use the associated component.

Providing an adapter in binary form as a dynamically configurable black-box component will not work in general, because the task of configuring an adapter usually cannot be reduced down to parameter adjustments; it is hard to avoid programming. Our suggestion instead, is that adapters be provided in source code. A whitebox, i.e. source-level adapter turns the rigid interface of a black-box component into an extremely flexible one. Although working with source code seems to contradict with the ideal solution of plugging, the level of skill and effort required for understanding and customizing an adapter program is reduced by the fact that the concern of an adapter is only interface conversion rather than the details of a specific application domain. So an adapter module will mostly be limited in terms of software complexity, and its size can be kept small enough to comprehend without a great amount of effort.

Since the engineers of a component can predict, to some degree, the form of customization the users might need, they can develop a model adapter to be completed by the user. A more detailed discussion of this idea is presented in Section 4. An alternative source of adapter for a component can be the third parties expert in a particular application domain who can develop multi-purpose libraries for adapter construction. To achieve this, software patterns [3] which are frequently used in adapting components can be identified and a multi-purpose class library based on these patterns can be constructed to serve the engineers of a particular domain of application.

A few significant usage areas for adapters are presented in the following section, and at the same time, the benefits of their being provided source-level are observed. The examples given below could be seen as a starting point to the identification of common adapter-related patterns.

3 Examples of Adapter Responsibilities

A component's interface can be altered in several ways. As the adaptee is a black-box component, the alteration is restricted to what can be done using its public operation set. A possible classification of types of adaptation is presented below, starting from the lowest level of complexity up.

3.1 Conversion of Parameters

Through an adapter a client can make use of more convenient data structures than the ones accepted by the adaptee's operations. The adapter in this case re-implements some operations using the client-defined data types as parameters. It has to perform the necessary type conversions before calling the corresponding original operations.

3.2 Refinement of Operations

Changing the operations' parameters can be coupled with improvement in their behaviour. The adapted version will make use of the original operation set, adding some extra features to them. For example, a very time-consuming function can be replaced with one that returns immediately after initiating a separate thread of execution to run the function concurrently. Then the client can learn whether the function has been completed by checking an adapter-defined flag before asking for the result. Alternatively, the adapter can return the output to the client by message passing or by triggering an event. Since neither concurrent execution nor event production is an inherent or indispensable feature of our core component, excluding them from the black-box implementation and offering them by means of an adapter prevents the risk of reducing reusability at the expense of extra features. Moreover, moving these highly platform-dependent facilities out of the component contributes to its portability.

As another example in this category, we can think of improving input management by an adapter. The core component's operations may be designed for ideal input; the adapter then deals with the possibility of erroneous input. The error handling mechanism can be customized according to the user's preferences. If, for instance, an out-of-expected- range value is given for a parameter, the adapted function can be made to issue a meaningful warning message.

3.3 Modification of Access Mechanism

An adapter can be used to switch to a more convenient access mechanism. Such an adapter takes the calls from its client through a particular kind of communication medium, does the necessary data structure conversions and calls the core component's functions through another medium. For example, a server that can only be accessed via the PVM [9] middleware can be made accessible to a COM [5] component through an adapter with a COM interface and using PVM inside. Thus, acting as a proxy [3] can be one of the responsibilities of an adapter component.

3.4 Extension of Functionality

Usually an adapter will not affect all the operations in an interface. In fact in some cases the aim may be to *extend* the functionality of an interface, rather than change it. A typical capability to add to a component is to visualize the drawable objects it contains. This is a capability that every visible object should possess in an object-oriented visualization or animation environment. However, adaptation of the component's visual representation into the environment is also important. There are various factors such as location, size, colour, level-of-detail, etc. which concern the environment as well as the component. These factors should not be hard-wired by determining them from within a black-box component. A much more flexible solution is to provide the possible visualization solutions for a component through a source-level adapter. An additional benefit from this approach is increased reusability, as the visualization techniques moved out of the black-box are highly platform-dependent.

Functionality can also be extended by adding functions which provide information they derive from the existing features, e.g. average values, totals, or more complex analysis results.

3.5 Modification of Overall Functionality

The range of conversion an adapter does can vary to the degree that sometimes an almost different interface may result. A typical reason for this much of adaptation is obtaining a higher level interface than that of the component. In order to increase the reusability of a component, its design and implementation should make as few assumptions about the characteristics of a client as possible, and should try to maximize flexibility. However, this may lead to a too generic and too detailed interface according to many of the clients. An adapter in this case can specialize it to a particular way of usage and reduce the complexity of functionality. A higher level function offered by the adapter will usually contain several calls to the core component's various methods.

For example, the component for simulation of an electric servo motor with many input parameters such as input and source voltages, output load, etc. can be encapsulated by an adapter that assumes constant source voltages and simulates various feedback control mechanisms (at a behavioral level). The result is a very different interface which excludes source voltages and accepts small, possibly digital, control-signals as inputs. The adapter being whitebox, the user is given the ability to examine and modify the control algorithms.

Substantial change in an interface can also result when a component is being customized to be used in an environment of an alien software architecture. Introducing a component with explicitly called methods into a system with an implicit invocation mechanism [8] is an example for this case.

4 Facilitation of Adaptor Construction by Reuse

Traditionally a binary component is coupled with an interface definition file and a set of examples illustrating the usage of the component through the interface. The adaptation of the component is usually done in an ad hoc manner through copying and pasting of these examples. As mentioned earlier, several research efforts aim at formalizing the adapter construction task. Yellin and Strom [11] and Konstantas [6] investigate the issues of automatically generating adapters from a high-level description language. Although these approaches are capable of providing solutions for interface translation and protocol conversion, in case of various other highly complicated adaptation types such as the ones exemplified above, they do not seem to be adequate yet. Bosch lists several types of component adaptation and introduces superimposition, a novel, powerful black-box adaptation technique that allows pre-defined, configurable types of functionality to be imposed on a reusable component [1, 2]. Superimposition is implemented as a language construct in an extended object model called LayOM, through the notion of layers.

We suggest a simpler approach which does not require a specific environment or formalism, and:

- enables adapter development with a considerably small amount of effort,
- eliminates ad-hoc reuse by introducing a structure,
- is powerful enough to facilitate a wide range of adaptation types.

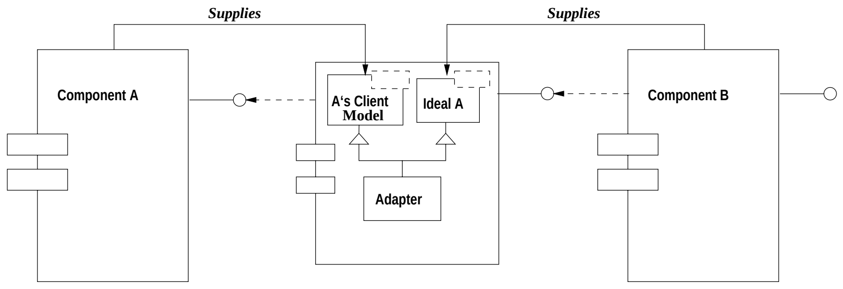

Figure 1: Adaptation by contribution from the adaptees

The main objective of this approach is to let the component developers contribute to adapter development as much as possible. A challenge is that a component's developers cannot know the exact specifications of the adapters required to connect it to other components. However, if the component is in a client position, the developers know what functionality it needs and can predict to a certain extent what aspects of this functionality are suitable for implementation by an adapter; similarly, if the component is in a server position, the developers will know what it provides and what other functionality could be needed by its clients that may be conveniently performed by an adapter. Therefore, component developers are able to produce customizable classes that constitute a partial implementation for an adapter. Figure 1 illustrates this approach.

We have applied the concept in the development of an integrated CAM application for the clothing industry. Interoperation of three major components of the environment, `Coordinator`, `Cloth_Cutter` and `CAD_Integrator`, is depicted in Figure 2. The `Coordinator` manages the overall activities including importing the geometry of the pieces to be cut from a CAD file via the `CAD_Integrator`, processing the geometry data, and generating cutting information based on a group of cuttable primitives (features), for the `Cloth_Cutter`. The `Cloth_Cutter` takes a list of commands interpretable by the hardware controller card and drives the card to perform the cutting operation.

There are various reasons for using adapters to connect these components. In this example we elaborate on the adaptation between the `Coordinator` and

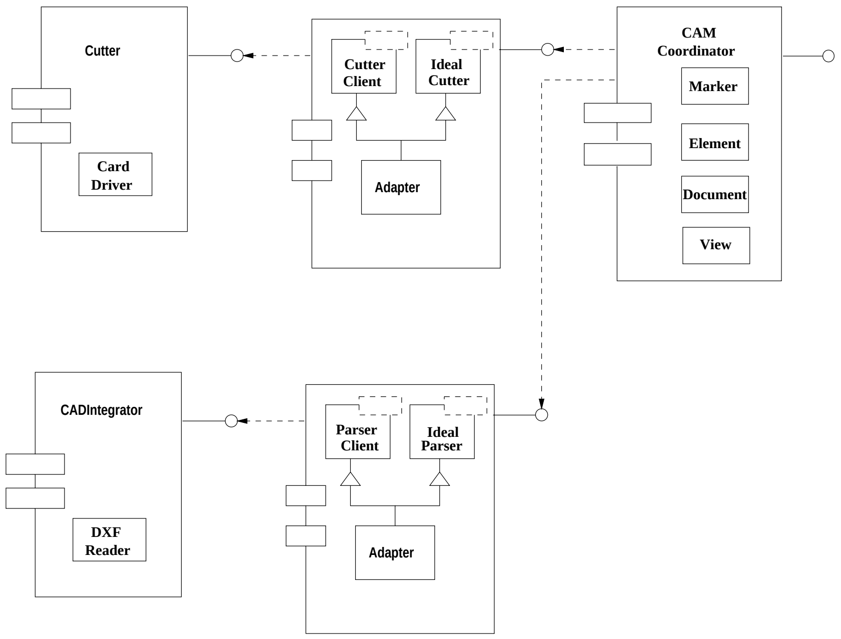

Figure 2: Adaptation of Cutter and Parser components to a CAM Coordinator

`Cloth_Cutter`. Firstly, there is a large difference between the data structures used by them. Secondly, multi-threading has to be introduced since the cut() operation of the `Cloth_Cutter` blocks its caller for a long time, i.e. throughout the whole procedure. Thirdly, the `Cloth_Cutter` does not provide event-based notification which is necessary for a real-time observation of the cutter's activity. The `Coordinator` is designed to work with an idealized cutter component, capable of multi-threading and event production. Considering that these features are not among the cutter's main responsibilities, the `Coordinator`'s developers embed them in a model class called **Ideal Cutter**, as shown in Figure 2. The `Cloth_Cutter` on the other hand, places the code that implements the protocol for its correct usage including procedures such as buffer management and error checking in the class **Cutter Client** as part of the adapter.

5 Conclusions

Customizable adapters could not only provide a solution to the problem of difficulty in resolving interface incompatibilities, but also could reduce the environment-dependence of black-box components. An important outcome of reduced environment-dependence is an improvement in reusability.

The degree of an adapter's customization can be maximized by making it white-box, but this requires component assembler to work at the source-level. Nevertheless, the manufacturers of components can support the users by providing template adapters, since they can predict, to some degree, the form of customization the users might need.

We describe an approach in which the model classes supplied by the components provide a skeleton solution for the most complicated types of adaptation that are hard to be achieved manually or automatically. Moreover, the overall structure is open to eclectic contributions in the sense that straightforward adaptation types such as parameter conversion and interface matching can be performed by automatic adapter generation techniques and integrated into the adapter construction process.

References

[1] J. Bosch. Adapting object-oriented components. In W. Weck and J. Bosch, editors, *Proc. 2nd International Workshop on Component Oriented Programming*, pages 13–21. Turku Centre for Computer Science, September 1997.

[2] J. Bosch. Superimposition: A component adaptation technique. *http://www.ide.hk-r.se/˜bosch*, 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, pages 17–26, November 1995.

[5] R. Grimes. *Professional DCOM Programming.* Wrox Press, 1997.

[6] D. Konstantas. Interoperation of object-oriented applications. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 3, pages 69–95. Prentice-Hall, 1995.

[7] O. Nierstrasz and L. Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 1, pages 3–28. Prentice-Hall, 1995.

[8] M. Shaw and D. Garlan. *Software Architecture.* Prentice-Hall, 1996.

[9] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[10] C. Szyperski. *Component Software — Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[11] D.M. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.