Software Technology for Design System Integration

P. Bertok

Department of Computer Science, Royal Melbourne Institute of Technology, Australia
Em: pbertok@rmit.edu.au

J.P.T. Mo

CSIRO Manufacturing Science and Technology, Australia

Keywords CAD, Application integration

Abstract Even fairly recent software can become easily outdated due to the very fast development in software technology. It is very common that the business logic of the application is still up to scratch, but the user interface, or the development platform has become obsolete. Re-use of legacy components of legacy systems is preferred to redevelopment. An often-used solution is integrating the existing software into a new system via re-engineering. This paper proposes an integration platform based approach for integrating legacy and new components into CAD systems. It analyses different integration patterns and elaborates on their usability. The approach described was successfully applied to the development of the system Virtual CAD.

1 INTRODUCTION

New software techniques and technologies are introduced at a very fast pace; user interfaces, development platforms and other components are superseded before products using them reach the end of their lifetime. Hence, existing systems can become outdated due to external reasons, while their business logic is still performing well. It becomes necessary to reengineer the “old” system, but that may have some difficulties. Modifying the software to reflect the changes in the new environment is usually not feasible, as the documentation may be incomplete, the developers of the original system are already not around to help etc. and the effort needed may be difficult to justify.

Replacing whole applications just because of changes in the environment is also not feasible in many cases. Applications, or at least their most important features may have to be retained for other reasons as well, e.g. because too many users or other applications depend on it. Hence, new methods have become necessary, which reduce the effort of reengineering.

The emergence of application service providers (ASPs), who offer sophisticated applications to organizations, created additional demand for the integration of applications, as shown in [1].

This paper gives a brief overview of reengineering methods, and presents an example of reengineering and integrating different components into a new system called Virtual-CAD. The integration described here consists of two major steps: (1) connecting legacy systems to the new system, and (2) adding new services or utilities, such as access control, to the new system.

2. REENGINEERING AND SYSTEM INTEGRATION

There are two basic approaches to reengineering legacy systems. One approach requires a detailed understanding of the legacy system, all its internal architecture and operations. This is called *white-box* reengineering. The other method requires only a minimum understanding of the legacy system's interfaces, as it attempts to access legacy system functions via these interfaces, without any detailed understanding of the operation. This is also known as *black box* reengineering. The white-box approach provides an opportunity to make major changes to the legacy system, which can be beneficial in the long term. The work involved is substantial, and possibly includes some degree of reverse engineering the old system before making the changes. The black-box method, on the other hand, avoids any changes to the internal workings of the legacy system. The work required is minor compared to the white box approach, and interruptions to the operation of the system are also reduced.

2.1 Wrapping

The black-box method is usually implemented by using encapsulation. The legacy system is accessed via external interfaces only, the behaviour of the system need not be fully known. The essence of the legacy system is maintained in its original, untouched form. The black-box method with encapsulation is often referred to as *wrapping*, and distributed object technology, such as CORBA, is very often used for implementation.

2.2 Wrapping methods

There are three major wrapping strategies, as discussed in [2]. These are the Ad-Hoc, Legacy-First and Component-First methods. The Ad-Hoc method, as the name suggests, is a one-off solution, individually tailored to a particular application. The work is carried out without a standard methodology and without any general characteristics that can be ported from one case to another. This method is only a temporary fix: the result is a unique product that can later display symptoms similar to the original problem and substantial changes will become necessary. Its application can really be justified only if it can be performed with minimum effort.

The Legacy-First approach attempts to preserve the legacy system's functionality in the new environment. The aim is to preserve the system or its functionality in the new environment, and no effort is made to update, modify or improve the legacy system. This solution can extend the life of the legacy system in its current form, which is very often necessary with widely used, popular software. In many cases the user may not even notice any difference between using the old and new systems. In practice, this method builds a shell around the legacy system, and interfaces the system functions with the new environment. The new system is inherently constrained by the legacy system, even though the shell itself can manipulate the data if required, and there can be a limited opportunity to make some improvements. All development efforts are concentrated on building this shell, which means the software development process can easily be kept under control.

The Component-First approach puts the requirements of the new system first, while using the related functions of the legacy system. During specification of the new system the legacy system's services are examined, modifications are considered and a revised set of services is defined. In this process the reusable functions are identified and the necessary changes are elaborated. The new system can have additional components, e.g. can have a new type of user interface, and eventually the user may not notice that the business logic and other internals of the system remained the same. The Component-First method still aims at minimising the changeover effort though, as individual components of the legacy system are the building blocks of the new system. But the approach gives more control over the system by allowing a reassessment of functionality.

3. SOFTWARE DESIGN PATTERNS AND REENGINEERING

Using design patterns for recurring problems in software development is becoming increasingly popular. While each task is a unique case with specific problems, solutions to similar problems offer great help and can significantly reduce the development effort. Each pattern describes a problem scenario together with the description of the solution, and this is presented in a general form not relating to any particular application. There are many handbooks documenting dozens of patterns, e.g. [3], [4], and having a particular problem a related pattern can easily be searched for. In the context of reengineering, common approaches such as black box, Component-First, etc. present recurring scenarios and using patterns is an obvious choice. As each case has its own individual characteristics, patterns need to be adapted, and there can be more than one applicable pattern. Finding the most suitable one needs a good understanding of the problem at hand as well as a good knowledge of design patterns.

In the following sections several black-box reengineering wrapper patterns are analysed, and a pattern for component access control is also described.

3.1 Wrapping Patterns

3.1.1 Proxy Pattern

In the Proxy pattern an object is represented by another, proxy object [3]. The proxy object provides access to the represented object, but hides all the implementation details. This is a very frequently used pattern, remote proxies represent objects in remote address spaces, virtual proxies store partial information only and create/load expensive parts of objects on demand, smart proxies perform additional functions before or after accessing the represented object. The proxy can operate without knowing the type of the real object: accessing the represented object via an abstract interface allows uniform treatment of objects.

In the Proxy pattern one object is represented by another object, i.e. it implements a one-to-one mapping. This can be useful e.g. in Legacy-First type wrapping applications.

3.1.2 Adapter Pattern

In many cases a specific service component needs to be accessed by a client who has an incompatible interface. An adapter can convert one

interface into another, and can also perform functions not provided by the server [3].

Figure 1 The Proxy Pattern

An adapter can implement a one-to-one or a one-to-many mapping, i.e. an object can represent one or more adaptee objects. This feature can be useful for Legacy-First or Component-First based wrapping.

Figure 2 Adapter Pattern

3.1.3 Façade Pattern

To provide a common interface to a group of classes, components or subsystems the Façade pattern can be used. Classes encapsulated by a Façade can also be accessed individually. The main advantage of a Façade is that classes' and other components' collaborations and dependencies can be simplified and decoupled from clients by the Façade.

The Façade pattern provides one-to-many mapping.

Figure 3 Facade Pattern

3.1.4 Wrapper Façade Pattern

The Wrapper Façade is a version of the Façade pattern, but it is more focused towards reengineering. It implements a simpler structure and can

encapsulate data structures and functions to provide conveniently usable and maintainable interfaces [5]. It can also use dynamic assignment of methods, but this requires a great deal of expertise at implementation.

The Wrapper Façade provides one-to-many mapping. In the Virtual-CAD system this pattern was used to interface legacy systems, as it offered a simply manageable mapping between legacy system functions and new requirements.

Figure 4 Wrapper Facade Pattern

3.2 A Pattern for controlling access to design components

Users of a system want to access different data in the system, and each user may have different access rights. Access control is provided across the whole system and is encapsulated in the system. It is the job of the system administrator to define access policies of individual objects.

While access rights are connected to users, a user can play several roles during interacting with a system. Hence, access rights are better represented by being connected to roles than by being connected to users. User roles can be modelled with interface objects, and in this model an actor represents a particular role performed by a user [6]. A system can offer different ways or uses to the user, and the set of all these uses form the total functionality of the system. A particular usage of the system can be described in terms of use-cases, i.e. where each case can contain one or more uses [7] [8] [9]. An example of use cases of a design system is given in Figure 5, each oval representing a use case. Before starting to use the system the user is authenticated; this is represented by the *Login* use case. Then the user may want to view a design component; this will involve the *ReadData* use case. Finishing the session will involve the *Logout* use case.

Figure 5 Use Case Model

Access control is reflected in software architecture as well, as illustrated in the object-oriented class diagram in Figure 6. When a user logs into the system, the *SecurityManager* will validate the user's name and password. Then, during a session a new object can be created the following way. The *FactoryManager* creates a *Factory* object and returns a reference of the generated object to the user, who can then use it to access the object. Security related information remains within the system: access rights are assigned by the *FactoryManager* at creation, and they are not passed on to the user.

Figure 6 Object-Oriented Diagram of Access Control

Once created, objects can be accessed only via a proxy that enforces access rights and prevents unauthorized access.

4. CONCLUSION

Design systems may need to incorporate different components, some of which may be legacy systems with established usage. Reengineering of some sort is a common way of bringing legacy systems in line with new requirements. Wrapping is a straightforward reengineering method that can

be implemented with reasonable effort. Using design patterns is a common way of software development, and several patterns can be used for wrapping.

It was found that

- Legacy-First wrapping can be implemented by using the Proxy pattern. If some enhancements are needed, the Adapter pattern can be more suitable.
- Component-First wrapping may need a more complex pattern, like Façade, that allows new functionality be built on top of the legacy system. The pattern can hide the legacy system and satisfy new requirements while still using the old system's components.

Patterns can also be used to implement additional functionality over integrated components, and an example of enforcing security and access rights illustrated the usage of component access control.

Design system integration with the help of patterns as described here was successfully tested and applied to the development of the Virtual CAD system described in [10].

5. REFERENCES

- [1] Computerised Facility Integration, <http://www.gocfi.com>
- [2] Morin T. (1999). Migrating Legacy Systems to CORBA, http://developer.netscape.com:80/viewsource/morin_corba/morin_corba.html, 1999
- [3] Gamma E. Helm R. Johnson R. Vlissides J. (1995). Design Patterns, Addison-Wesley, 1995
- [4] Rising L. (2000). The Pattern Almanach, Addison-Wesley, 2000
- [5] Schmidt D. (1999). Wrapper Façade, C++ Report, February 1999
- [6] Jacobson I. (1995) Object-Oriented Software Engineering – A Use Case Driven Approach, Addison-Wesley, 1995
- [7] Sigfried S. (1996) Understanding Object-Oriented Software Engineering, IEEE Computer Society Press, 1996
- [8] Vadaparty, K. (2000) Use Cases - Basics, Journal of Object-Oriented Programming, Feb. 2000
- [9] Vadaparty, K. (2000) Refining the Use-Case Model, Journal of Object-Oriented Programming, May 2000
- [10] P. Bertok, J.P.T. Mo, Woodman S. (2000) Collaboration and Application Integration: Distributed Design with Virtual CAD, in this volume