Pattern Usage for Legacy System Integration in Distributed Environments

Seng Thean Ooi[1], Peter Bertok[1], Liping Zhao[1], Harvey Jiang[2], John P.T. Mo[2]

[1]Department of Computer Science
Royal Melbourne Institute of Technology
P.O.Box 71 Bundoora, Vic. 3083
Australia

[2]CSIRO Manufacturing Science and Technology
Locked Bag No.9, Preston, Vic. 3072
Australia

{sooi, pbertok, liping }@cs.rmit.edu.au
{Harvey.Jiang, John.Mo}@cmst.csiro.au

Abstract

Many systems have been developed on platforms or with interfaces that can become easily outdated, while the application logic and code are still up to scratch. Integrating an existing system into a new product can present similar problems. Re-developing existing and working products just to accommodate a new platform or new environment is not a feasible solution, as it would most likely be expensive and error prone; re-engineering the existing product is the preferred alternative. There have been two strategies proposed for re-engineering existing software: one that involves 'deep understanding' of the software, and one that relies on 'shallow understanding' only.

This paper describes the shallow-understanding strategy in a distributed object oriented environment, and proposes the use of design patterns to facilitate the re-engineering work.

1. Introduction

The pace at which new technologies are introduced in software development is becoming a problem for the industry. One of the most critical issues is that new technologies are not always compatible with the old ones, and because of this, existing systems can become outdated or aging rapidly, so called "legacy systems". Often, systems that become obsolete are not due to their internal computational logic and functions, but simply due to the incompatibility of the new hardware platform, new programming languages, or new infrastructure. Modifications to the legacy systems are not easy or straightforward for various reasons. For example, the developers of the legacy system may not be around any more, or the system documentation is incomplete. The internal structure of the system is usually unknown or difficult to find out. Winsberg summed up the dilemma in the following words: "Legacy code is like Chernobyl: too messy to clean up but too dangerous to ignore" [7].

The general availability of computers as a tool in everyday activities has changed the way software is developed and used. Applications, or at least their most important features, have to be retained in a rapidly changing environment, because too many users or other applications may depend on it. The issue in this case is how to interface new software with legacy systems. Replacing or redeveloping operational systems simply for the reason of advancement in the surrounding technology can be difficult to justify. Re-engineering old systems has become more prevalent than ever. The purpose of re-engineering is to use the services of an existing legacy software component in a new environment with new requirements.

2. Re-engineering Legacy Applications

2.1 Approaches

Currently there are two basic approaches to re-engineering: one requiring deep understanding (*white-box*) and the other requiring shallow understanding (*black-box*). The white-box approach requires a detailed understanding of the legacy system and involves substantial software re-development. However, the white-box approach can be beneficial for long-term reuse and provide opportunities for system improvement. The black-box approach, on the other hand, avoids understanding the internal structure of the legacy system, and attempts to access it via external interfaces. This approach keeps changes to the system at minimum, which also reduces the disruption of operating services.

Distributed object technology (DOT) has been widely accepted as enabling technology for distributed software improvement, maintenance and re-use [4]. DOT has been used to create middleware, a middle layer software that sits between software components such as clients and servers. Middleware offers an integration platform for diverse applications, including legacy systems. DOT also uses encapsulation technique to support the black-box approach by preserving the original essence of the legacy system and providing an external interface. This can be an important aspect when dealing with legacy components whose behaviour may not be fully known. The black-box approach with encapsulation has become a widely accepted method and is known as *wrapping*.

This paper discusses some wrapping techniques from a software engineering point of view. It also discusses the use of patterns for wrapping, and presents the case of wrapping a legacy CAD system in the context of a larger development project.

2.2 Wrapping Methods

Morin [5] discussed three major wrapping strategies: Ad-Hoc, Legacy-First and Component-First.

The Ad-Hoc method is an opportunistic one unique to each particular application. As the name suggests, there is no standard methodology involved. System development depends on the nature of the individual application and there are no general characteristics that can be ported from one case to another. As a result, this method introduces many problems in the design, implementation, and maintenance phases.

The Legacy-First approach is aimed at maintaining and offering the existing services of a legacy system in a new environment. No new or modified features, e.g. functionality or user interface are required. In many cases the user should not be able to notice any difference. The objective of this approach is to define the necessary interfaces and extract the functionality of the legacy system into a new "shell" where the information can be further manipulated as required by the new system. The consequence is that each interface is constrained by the capabilities of the legacy system. Considerable work is needed to make the interfacing process easier.

The Component-First approach makes use of the change in the environment. Instead of exporting the application to a new platform, requirements in the new environment are examined and a detailed component design is performed. Any need for functions provided by legacy systems are then identified and introduced into the new system design. Hence, it provides considerably more leverage over the system. The application can have a new interface, modified functionality, and a different set of requirements presented by the new environment. Even so, it is only the interface part that is re-modeled; the execution logic of the legacy application will not change in this process. In other words, Component-First wrapping will produce a new user interface to the old system while leaving system internals intact.

The Component-First approach can be implemented by software patterns. Implementing re-engineering methods based on patterns on a distributed platform has the advantages of modularity and extensibility; components can be varied or added to the system while leaving unaffected parts of the system intact. Encapsulation is readily offered by DOT, and CORBA facilitates wrapping even further by its Interface Definition Language (IDL) separating interface from implementation.

3. Design Patterns

The use of design patterns to solve recurring problems and implement re-usable software has been thriving. Although each pattern was designed to address a particular problem, their ability to be re-used in other applications makes them valuable. Many of these patterns are well documented, and available in "catalogs" [3]. In the process of re-engineering, the different approaches, such as white-box or black-box, present recurring scenarios, and an obvious solution is to use well-proven patterns to support these approaches. Each re-engineering approach has its own distinctive features, and there can be more than one pattern suitable for it. Finding the best pattern for a particular application needs careful a analysis of the problem at hand and good understanding of related patterns.

3.1 Relevant Patterns

In the following we discuss four architectural patterns that we found applicable to wrapping.

3.1.1 Proxy Pattern

The Proxy pattern is an object that represents another object. A proxy object provides an interface to the client for the represented object while hiding the implementation details. There can be remote proxies representing objects from a different address space, virtual proxies that create expensive objects on demand, protection proxies providing access control and smart proxies that perform additional actions when an object is accessed (Figure 1). An important feature of the proxy is that it does not always have to know the type of the real subject. If it can manage its subject via an abstract interface only and do not have to instantiate real subjects, then it can deal with all real objects uniformly.

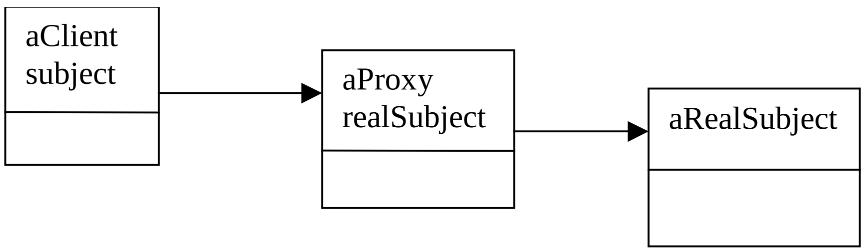

Figure 1 Proxy Pattern

3.1.2 Adapter Pattern

The Adapter pattern converts one interface into another (Figure 2). The adapter can be responsible for functionality that the adapted class does not provide. Its main advantages include the ability to cooperate with unrelated classes that may have incompatible interfaces.

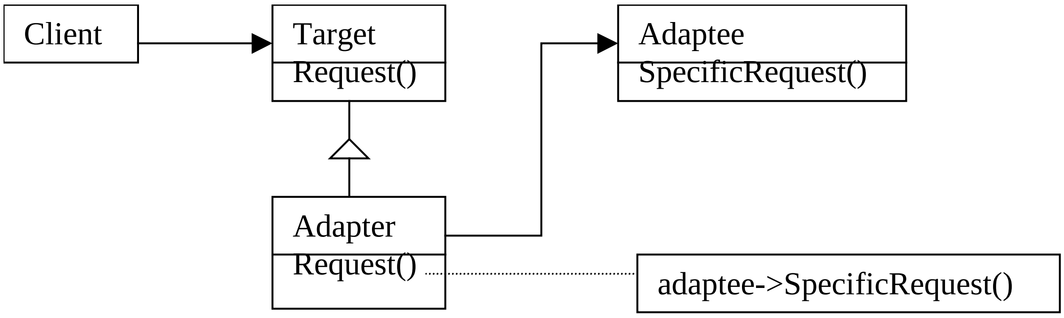

Figure 2 Adapter Pattern

3.1.3 Facade Pattern

The Facade pattern provides a common interface to a group of classes, components, or subsystems (Figure 3). Classes under the façade can collaborate with each and yet can be individually accessible. A facade can decouple a system from its clients or other subsystems and simplify the many dependencies between the different components of the system.

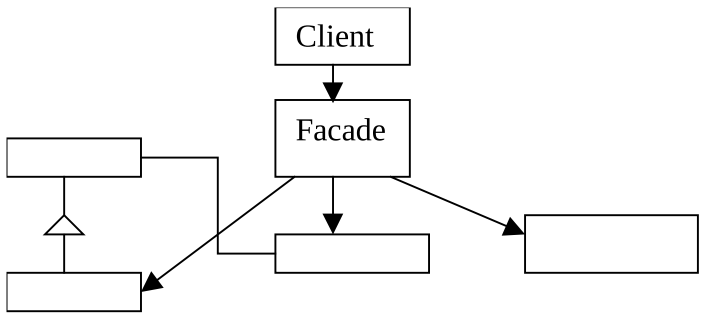

Figure 3 Facade Pattern

3.1.4 Wrapper Facade Pattern

The Wrapper Facade pattern is similar to the Facade pattern, but it is more focused (Figure 4). It is to encapsulate data structures and functions to provide conveniently usable and maintainable interfaces [6]. The Wrapper Facade can be implemented with the Bridge pattern if the abstraction methods in the Bridge pattern can be dynamically assigned. This can be a very flexible solution, but the implementation may need a great deal of expertise.

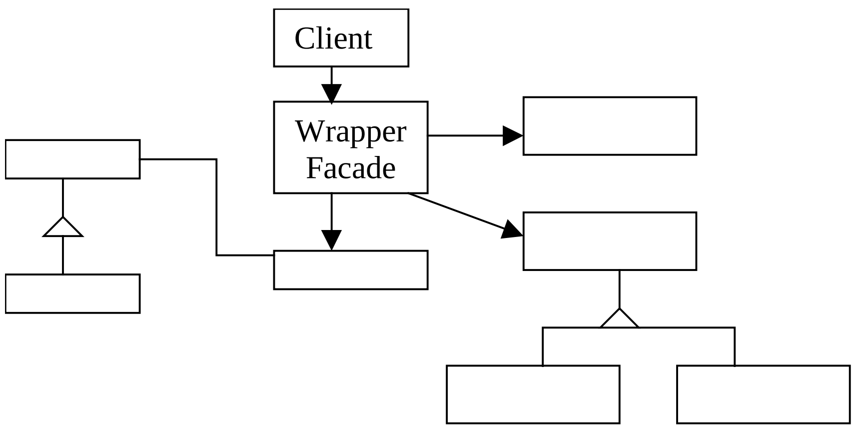

Figure 4 Wrapper Facade Pattern

3.2 Applicability

The four patterns mentioned above can all be used for re-engineering, but their suitability depends on the approach. For Ad-Hoc type wrapping it can be difficult to choose a pattern, as it is individually tailored to the application. The Legacy-First approach can be easily implemented with the Proxy pattern, where each function or data structure has its proxy on the other platform. It can be a straight function invocation or data transfer. The parameters or data can be passed directly as in a command line or via files. If some enhancement of the legacy application is also required then the Adapter pattern can be used, which is still based on an almost one-to-one correspondence between the old and new interfaces.

Component-First wrapping needs patterns that can provide more than simple translation from one interface to another. The Facade pattern allows the implementation of new functionality on top of the legacy system. The pattern can hide the legacy system's structure so that the new interface satisfies the requirements of the new system, but it still utilizes the components of the old system. The Wrapper Facade pattern can be more suitable in some cases, as it is more specific and less complex. It provides a translation of the new interface into the old one by encapsulating functions and data structures. The Wrapper Façade pattern has been the primary focus in this research because of its applicability to a variety of applications, while still maintaining a relatively simple architecture.

4. The Virtual CAD System

4.1 Platform and Pattern Used

The primary objective of the "Virtual CAD" project is to develop a system that allows remote users around the world to access CAD models the same way as local users do. There are many CAD system vendors, and it would be difficult for any one to capture a lion share of the market [2]. Hence, in large scale global projects involving several design teams around the world, it is common that different groups use different CAD systems. The common client interface that can access different CAD systems and can use their native CAD model data was developed to facilitate unambiguous information flow in engineering projects.

Selecting a platform, or middleware, for integration is the first step, as it can have a serious impact on the emerging new system. This decision was based on user requirements rather than on architectural or other, software related matters.

To keep the system simple, a web-based solution would have sufficed, as most browsers can display different types of data by using plug-ins. The Virtual CAD system, however, is intended to be used as an active front-end in a design system, i.e. one that can both display CAD data and perform some modifications on them. However, implementing functions that modify CAD data would have been a great challenge with a Web browser and its plug-ins.

When the shortcomings of the Web-based solution became clear, the next level was CORBA, DCE or a similar platform. CORBA was preferred over DCE and other systems mainly because of convenience: it was designed as object oriented from the start, and it had implementations (Orbix) under the operating systems available in the user environment. CORBA proved to be the right choice, it allowed all required user functions to be implemented without any major problem.

The task described in this paper was part of the larger Virtual CAD project, and consisted of integrating a commonly used CAD system, AutoCAD, into a larger, CORBA based system, as illustrated in Figure 5. Other CAD systems, such as Pro-Engineer, SolidWorks, IDEAS, are also being considered for future integration. The clients can be distributed around the world, and each of them is running the same code, albeit with possibly slightly different local configuration data.

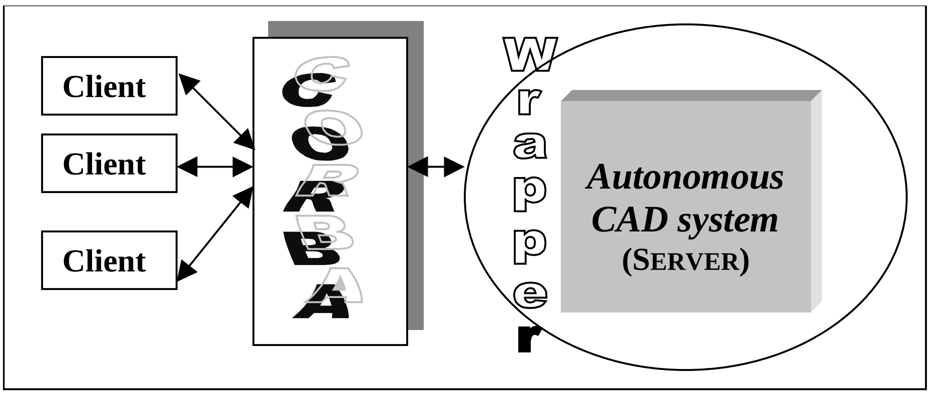

Figure 5 Wrapper Facade Pattern Applied to Virtual CAD

Component-First based wrapping approach was used, as different clients need access to different CAD functions and data. The work involved a feasibility study that also examined the appropriateness of messages and data transfer. The aim was to make best use of the legacy CAD system's capabilities and of the available bandwidth in the communication network (Internet or Intranet). There was no clear one-to-one correspondence between the services/data provided by the legacy system and those required via the new client interface. However, since developing the client was also part of the project, it was possible to find a quasi-optimal mapping of objects between the old and new systems, which did not require too complex conversions of functions and data. The result was a many-to-many type mapping, which really restricted the choice to the Facade and Wrapper Facade patterns. As the Proxy and Adapter patterns suit one-to-one mappings, implementing the wrapper in these patterns would have meant implementing multiple wrappers, which was not a desirable solution.

The extendibility of the wrapper was also considered, as modifications, such as including additional functions in the new interface or including other CAD systems, may need to be incorporated at a later stage. The Facade and Wrapper Facade patterns offer relatively easy extension of the wrapper, as opposed to the Adapter and Proxy patterns, which present a harder case for extension.

When comparing the Facade and Wrapper Facade patterns, simplicity of the wrapper was considered. While the Facade pattern can implement complex relationships, this feature of the pattern was not required in the application, because developing the client was also part of the project as explained above.

4.2 Implementation

The wrapper was subsequently developed by using the Wrapper Façade pattern. The implementation was divided into two parts, the first one laying down the class foundations and the second one addressing error handling and exceptions.

4.2.1 Clustering functions and data structures into classes

Within the wrapper class several classes were defined to reflect the new user interface and functionality. In many cases, the client was specified to ease the burden of interface conversion, and the local server functions are replicated at the client side. In these cases simple message passing between the two sides was sufficient. Great help was the Java 3D based client interface that could emulate many of the CAD server's 3D-manipulation capabilities [1]. Classes representing these functions simply forward their method invocations to the underlying low-level server functions.

An important feature of the Virtual CAD system is its ability to manipulate CAD primitives and modify CAD models. This is illustrated in Figure 6. In this figure, the class `BasicPrimitive` defines common interfaces required by a CAD primitive. It associates with the `TransformGroup` and `BranchGroup` classes defined in the Java 3D library. It defines basic interfaces to transform the geometry of a primitive. In addition, this class defines two methods (post-order and pre-order) to traverse this structure. The class `LeafPrimitive` can define a primitive such as standard shapes and CAD models represented in neutral formats. The class `CompositePrimitive` can define a primitive that contains other primitives (defined as children) such as `CompositePrimitive` or `LeafPrimitive` objects. The class `CompositePrimitive` provides interfaces to add and delete children.

These basic primitives support the fundamental user operations in the Virtual CAD environment. For more complex operations, such as assembling a few components in the CAD server, additional levels of indirection are required. A possible way to implement additional indirection is dynamical dispatching of wrapper facade method implementations, which can also increase extensibility. Aggregated functions need to be developed and wrapped in the server by using languages associated with the CAD system.

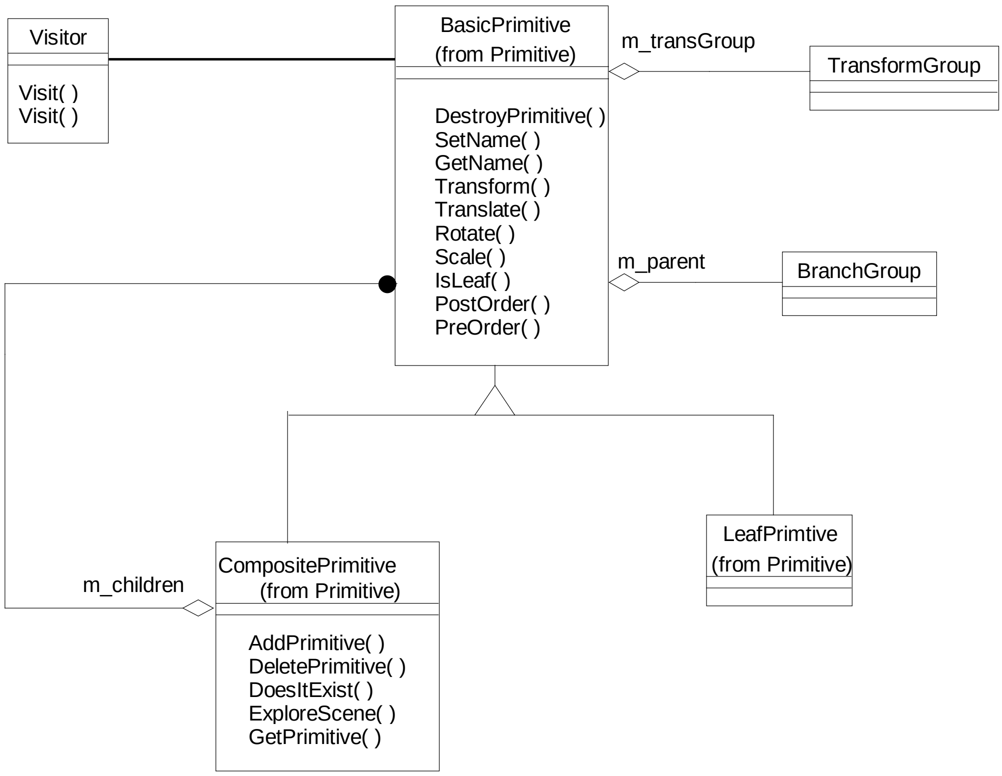

Figure 6   System Structure with CAD Object Manipulating Primitives

4.2.2 Establishing error reporting mechanisms

Handling errors is essential in any application. It can be kept simple by reporting back to the caller only, or can be quite complex when corrective actions may need to be initiated, and the pattern used may need to accommodate that.

In this project it was kept fairly simple. The options were the following. Errors could be communicated back to caller via standard error codes, as done on many conventional platforms. Using exception-handling mechanisms provided by CORBA was another option, but it was more complex and needed to be justified. Currently the exception handling facilities in Java are used, which provide a comprehensive set of functions for contingencies. In future the error may be communicated back to server. The error handling capabilities of the server will also have to be considered, as the server may be coded in a language different to the client.

If the error reporting mechanisms become too complex, however, that can have an effect on design pattern selection, and may justify the use of the Facade pattern in this application.

5. Conclusion

This paper presented a case of legacy system integration based on design patterns. The approach chosen was a black-box method with encapsulation: a wrapper was built around the legacy application. Several patterns were examined for suitability, and the simplest one that satisfies the requirements was selected.

The research produced the following findings.

- User requirements preferred CORBA as the implementation the platform.
- All four design patterns: Proxy, Adapter, Facade and Wrapper Facade could be used in re-engineering the application.

- The Wrapper Facade suited the application best and was extensible to cater for future inclusion of other legacy objects or new functionality.
- A straightforward application of the pattern resulted in a simple and coherent system implementation.
- Error reporting mechanisms more complex than the current Java methods can justify the use of the Facade pattern.

6. Future of Re-Engineering

Integrating legacy applications will become more and more complex. On one hand, legacy systems are increasing the level of software complexity. On the other, wrapping requirements are becoming more and more challenging; they can include load balancing, multi-threading, connection pooling, etc. One of the main thrusts is increasing demand for graphical user interfaces (GUIs) that can transform many legacy systems into up-to-date web-connectable applications. A new class of middleware based on Java is already emerging on the horizon that can integrate legacy systems while satisfying the performance requirements of many applications. While these will facilitate the re-engineering work significantly, they can not replace human design and experience encapsulated in design patterns.

References

[1] Chan P. and Lee R. The Java Class Libraries. Second Edition, Volume 2, Addison-Wesley, 1998

[2] Fowler S. and Karinthi R. Remote access to CAD databases using an information sharing system, Computers in Industry, Vol.29, 1996, pp.117-122.

[3] Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison-Wesley, 1995

[4] Mowbray T.J. and Zahavi R. The Essential CORBA-Systems Integration Using Distributed Objects, John Wiley & Sons Inc. 1995

[5] Morin T. Migrating Legacy Systems to CORBA, http://developer.netscape.com:80/viewsource/morin_corba/morin_corba.html, 1999

[6] Schmidt D. Wrapper Facade, C++ Report, February 1999

[7] Winsberg P. Software Legacy Code: Don't Bag It, Wrap It, http://www.datamation.com/apdev/05bsw100.html, 1999