Piecemeal Migration of a Document Archive System with an Architectural Pattern Language

Michael Goedicke, **Uwe Zdun**
Specification of Software Systems
University of Essen, Germany
{goedicke|uzdun}@cs.uni-essen.de

Abstract

Large applications that have evolved over the years, are often well-functioning and reliable, but have severe problems regarding flexibility and reuse. It is hard to migrate such systems to a more flexible architecture or to new technologies. The document archive/retrieval system discussed in this paper is an example system, that has these problems. We will present a reengineering case study based to component technology based on an architectural pattern language. The patterns aim at the introduction of flexibility into black-box component architectures in a piecemeal way.

1 Document Archive/Retrieval System

The document archive/retrieval system in focus of this work, is a large C application that has evolved over the years. The system has a suitable large-scale architecture for its tasks and it is working well in several customer installations. But with growing complexity extensions and changes became more and more problematic. Moreover, deployment at the customer requires a certain amount of customizations. Moreover, the system suffers from minimal reuse of existing software components.

In general a document archive and retrieval system allows users to archive a large number of documents on several devices. The primary devices used in the application area are optical storage devices, but there are also other media types, such as hard drives or remote storage devices. Users can retrieve archived documents with different text- and GUI-based clients.

The system was originally designed and implemented in C on a Unix platform supporting only one database management system. It was ported to several Unix variants and finally to Windows NT. At some later time support for other database management systems was added. The archive and retrieval server were on Unix and Windows written in C, while Unix clients were C written and Windows clients were developed in C++ and Java.

The system consists of three distinct system layers. The *Client Layer* hosts the different customer clients. Moreover, there exists a number of system and administration clients. All clients access the system via a set of proprietary protocols and interfaces based on sockets. A *System Layer* hosts the system demon as a central access point managing a archive, retrieval and database query handler processes. A scheduler, a cache, and document handlers implement the actual archive/retrieval tasks. An administration demon implements the server-side of the administration tasks. Finally, the *Storage Device Layer* hosts several different storage device drivers, like optical disk jukebox driver, hard disk driver, network access protocols, etc. These are unified in this layer behind a RPC interface API.

The company faced several problems with the legacy application. It was hard to change how clients and servers communicated, because they used a proprietary protocol based on sockets. Only the access to storage devices was using an RPC standard protocol. Support of the system on several platforms with several versions became a severe maintenance problem. Moreover, similar problems occurred in maintenance of the database code, because the system used different DBMS products/versions that were accessed by different protocols and that had different SQL dialects.

Both problems led to a reduced understandability of the code, since it was full of pre-processor directives for different versions. For the database code, a dynamic change to the other database was impossible. Instead a complete re-compilation was necessary. Since many dependencies in the different parts of the code existed, it was hard to exchange implementations against other implementations. Thus a piecemeal migration seemed to be hard to accomplish.

The overall architecture of the system has proven to be reliable and the involved entities seem to match the modeled real world situation well. Therefore, the basic architecture of the system does not necessarily have to change. In a first idea of reengineering, a solution of migrating the whole code base to a rewritten solution in C++ or Java was envisioned. But this would have led to considerable costs in (useless) legacy migration. Moreover, it would require an additional concept for stepwise migration of the legacy parts. And presumably such a major reengineering step would have led to another architecture. This would require a considerable redesign effort, but the value of such an effort is questionable.

The reengineering case study presented in this paper presents a concept for piecemeal migration of the application to a solution that avoids the sketched problems, is based on reusable black-box components, and shows a significantly increased flexibility. There were the following basic concepts behind the restructuring:

- *Component-Oriented Structuring*: But even though the system is almost structured according to the component-oriented paradigm, there is no further support for component-orientation. Language support and accompanying services, as they can be found in component frameworks of scripting languages [13, 6] or in modern middleware architectures (see [15]), are missing. But these are a central reasons why the system is inflexible and hard to understand. The C subsystems have nearly no clear interfaces on which clients can rely. Thus they are not changeable without interference with their clients.
- *Multi-Paradigm Development*: A central observation for the case study is that most software systems of the presented application scale are structured with multiple paradigms. In general a paradigm is a set of patterns and rules for abstractions, partitioning, and modeling of a system. Even though the document archive and retrieval system is written entirely in the C language, we can observe system parts that are structured according to the procedural, object-oriented, component-oriented, functional, relational, etc. paradigms. But not all of these paradigms are explicitly supported by programming languages and used technologies.
- *An Architectural Pattern Language for Piecemeal Legacy Migration*: For piecemeal component migration we have used the pattern language from [8, 7]. The aim of the pattern language is to generate flexible black-box architectures in a piecemeal way. Here, for space reasons, we can only give thumbnails of the new patterns with problem and solution. The interdependencies of the introduced patterns among each other and with several popular object-oriented software patterns are introduced implicitly in the reengineering examples of the next section. It is important to note that the pattern language contains several smaller popular object-oriented software patterns, including WRAPPER FACADE [14], PROXY [4], DECORATOR [4], ADAPTER [4], and FACADE [4].

2 Component Integration

In a first step we use the distinct C written subsystems, that form the processes of the system, as components. Afterwards, we will refine these components in a piecemeal effort. Firstly, we have to choose an object system (in order to apply OBJECT SYSTEM LAYER) and have to wrap the components with this object system. We have the following choices: We can use C++ (or Java with JNI) and integrate the C components. Or we can build an object system as a library in C, as discussed in [7]. Or we can use an existing OBJECT SYSTEM LAYER implementation, as for instance the object-oriented scripting language XOTCL [12].

Here, we propose to use an object-oriented scripting language, like XOTCL, because it language supports several of the patterns: It is itself an OBJECT SYSTEM LAYER, it implements a DISPATCHER, and it has three language constructs implementing BEFORE/AFTER INTERCEPTORS. However, in this paper we present the reengineering with the pattern language independent of the used object system.

After choosing the object system, we provide all existing components with an export interface in the object system (which resemble mainly the existing header files). Now we connect the export interfaces with the existing implementations, as in the COMPONENT WRAPPER pattern (the wrappers firstly just forward to the implementations). Now we have a first class representation of all components in the object system, which defines their export part for the EXPLICIT EXPORT/IMPORT pattern. The COMPONENT WRAPPER can be used for changes, while the components are independent black-boxes.

These components are in a first step just plugged together again. A minimal implementation in the object system steers the components with the DISPATCHER. Now we want to extract the hot spots of the application, i.e., the parts which are likely to change often, stepwise into the object system. In this paper, we present two such hot spots and their migration: the communication and database subsystems.

3 Communication Subsystem Reengineering

For communication within the system and with the client layer a proprietary protocol based on sockets was used. For communication with the storage devices it was partially replaced by an RPC mechanism, that defines some interfaces. But all used communication protocols were hard-wired into the code. Therefore, the communication subsystem was hard to replace or change or extend.

Moreover, explicit interfaces were only partially given. Support for other programming languages (as for instance a Java client) had to be programmed by hand. There was no component- or object-model for communication and only

Name	*Problem*	*Solution*
COMPONENT WRAPPER	How can we solve the problem that black-box components are hard to customize beyond parameterization?	Provide a COMPONENT WRAPPER to the black-box as a central place for component access. The COMPONENT WRAPPER is a white-box for the component's client and enables to bring in changes and customizations, e.g. with DECORATORS and ADAPTERS.
DISPATCHER	How can we implement control tasks over the message flow, like interception, modifications of messages, or traces, in a programming language that does not support such techniques natively?	Build an explicit DISPATCHER instance. Let the calls in the subsystem and the calls to the subsystem pass the DISPATCHER, that redirects to the original receiver.
OBJECT SYSTEM LAYER [7]	How can we apply advanced object-oriented techniques in non-object-oriented languages or in object systems that are not powerful enough?	Build or use an object system as a language extension in the target language and then implement the design on top of this OBJECT SYSTEM LAYER.
EXPLICIT EXPORT/IMPORT	How can we ensure that internal changes in the exported component do not affect the exporting component and vice versa (especially when both sides of a component connection tend to change)?	Provide explicit interfaces for both, export and import of a component. Component describe the services they provide and that they require.
BEFORE/AFTER INTERCEPTOR	How can we avoid the problem that customizations through (class-based) ADAPTERS and DECORATORS split up one conceptual entity into several entities leading to a loss of transparency?	Provide a mechanism to introduce before/after interception. All decorations/adaptations can be performed transparently by this mechanism.

Table 1. Architectural Pattern Language for Piecemeal Reengineering: Pattern Thumbnails

a few basic services. Therefore, several service, as known from popular middleware systems (like messaging service or transaction service) were partially programmed and maintained by the development team. It was nearly impossible to exchange the communication subsystem of the system itself against another technology.

There were several possible solutions to these problems: The RPC mechanism could be used for the whole system and combined with a component model. Or one of the different middleware technologies could be used with its component model, services, etc. Or a web-based solution on top of HTTP could be used.

E.g., a middleware enables a company to have a single framework for development, integration, and extension of distributed applications. It provides interoperability through platform and programming language independence. Most middleware approaches come with a component-model and concepts for legacy integration. Moreover, they provide a comprehensive set of services, like messaging service, naming and directory service, transaction service, security service, etc.

The task for reengineering of the document archive and retrieval system's communication was to find an architecture, that lets the system make use of modern middleware technologies and benefit from their services. Furthermore, the resulting system should not depend on one technology. The communication subsystem itself should be exchangeable as a component. But since the existing system is well-functioning, we should find a way for piecemeal migration to the new communication technology.

Through the existing application we already know the required communication facilities of the system. Thus we can extract the calls to the communication subsystem, that are scattered over the code. And we can build an equal, generic import interface with EXPLICIT EXPORT/IMPORT. Firstly, the communication classes, like `AdminComm`, `ArchiveComm`, `RetrievalComm`, and `SQLComm`, are nearly identical with the structure of the proprietary communication protocol. Later on we will refine the implementations to more generic interfaces. E.g., the archive and retrieval communication classes are unified to one class, but the `ArchiveComm` and `RetrievalComm` classes are maintained as Adapters [4] to this class for piecemeal legacy migration.

We simply build for each type of communication function an abstract method in the interface classes. Afterwards we have an abstract view onto the communication of the system in the OBJECT SYSTEM LAYER. We can replace calls to the proprietary protocol stepwise with calls to the import interfaces of the archive/retrieval system's communication component. But we still have to realize the communication component with the middleware of choice (here: CORBA).

In order to obtain a piecemeal migration, we have to create an IDL interface and stub/skeleton classes in the CORBA implementation language (here: C++) several times until the reengineering effort is completed. At least for this time period import, export, and IDL interface carry the same information redundantly. Therefore, we propose to use an automatic generation of export interface and IDL from the import, as in shown Figure 1.

The import interface is written by hand. A generation tool produces corresponding export and IDL interfaces. With an introspective scripting language, as XOTcl, we can use the introspection facilities to retrieve the interface from the import interface implementation automatically. Otherwise we have to write a small parser. Then the C++ CORBA communication implementations are adapted to the changes, if necessary.

Finally, we have to integrate the sightly different object system of CORBA with the object system of the chosen Object System Layer. Such paradigm integration tasks can be fulfilled in the Component Wrapper object. Most often a set of Decorators and Adapters on a Component Wrapper, that only forwards messages, are sufficient. A more convenient way for such interface adaptations is to register a Before/After Interceptor on the Component Wrapper object. The export implementations of these components in the Object System Layer, therefore, are Component Wrappers to the C++ implementations of CORBA stubs/skeletons.

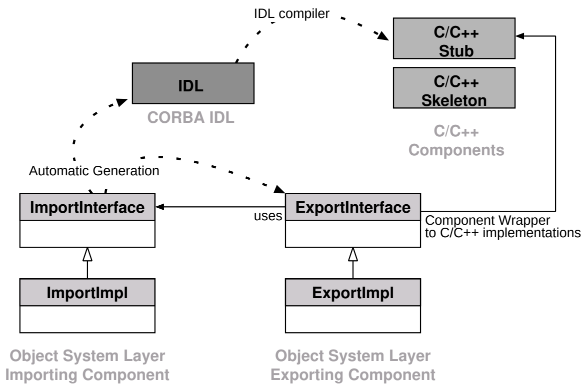

Figure 1. Automatic Generation of IDL from Interface Class

3.1 Database Interface Reengineering

The database management system integration faced similar problems as the communication subsystem. The code for DBMS access of the two used DBMSs was scattered over the code. A re-compilation of the system was necessary in order to exchange the used DBMS. Central adaptations, e.g. in the used SQL dialects, were nearly impossible, but had to be propagated through the code. Therefore, the system was not independent of implementation details of the DBMS.

In addition to these problems, it is explicitly required that different database products and especially different versions of the same products, have to be supported. Often customers already have a DBMS installation. Since the costs of DBMS are quite high in comparison to the document archive/retrieval system, it is undesirable that customers have to buy another version of the DBMS, just because the archive/retrieval system can not work with the existing version. That imposes the maintenance requirement, that new DBMS versions have to be rapidly adopted by the archive/retrieval system. Thus the connection to a database has to be extremely flexible.

The former solution consists mainly of `ifdef` preprocessor directives of the following style that were scattered through the code:

```
#ifdef INFSQL
  $SELECT clu_name
  ...
#endif    /* INFSQL */
#ifdef ANSISQL
  returnValue = archiveSelect(&sql_code, ...
#endif   /* ANSISQL */
```

This solution was inelegant, hard to read, and hard to change. Changes could not be made centrally, but had to be propagated through the code. Therefore, errors were hard to trace.

In Figure 2 we can see the architecture with the pattern language. Again we build a generic interface to database access. Then we build Component Wrappers to C/C++ implementations of different DBMS products/versions in the Object System Layer. Thereby we have simulated a small object-oriented database on top of the relational databases. The Component Wrappers seamlessly integrate and adapt the relational paradigm to the object-oriented paradigm. The different SQL styles of different DBMS products/versions can be handled on the Component Wrappers with Before/After Interceptors that modify the calls, if necessary. Thus we have to pass all calls through a Dispatcher that invokes the Before/After Interceptors.

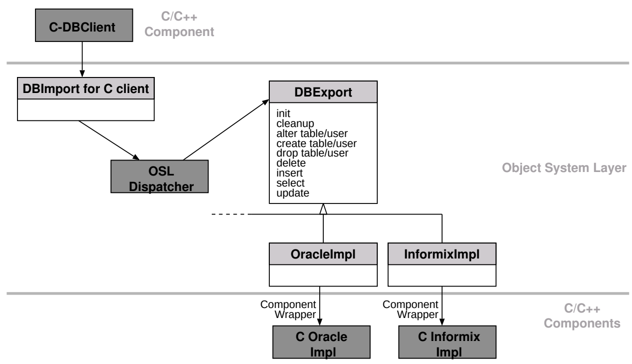

Figure 2. Database Interface With an Object System Layer

The database interface is mainly used by the C components. Therefore, we write import interfaces for the C clients, that are called through a C API (as if the Object

SYSTEM LAYER would be a native C library). The DISPATCHER maps these string-based calls to the export of the database interface. The clients can rely on these interfaces despite changes in the realization. BEFORE/AFTER INTERCEPTORS in the DISPATCHER can adapt the calls to the changes.

4 Related Work

Here, we sketch some approaches that are known uses of the pattern language. There are several patterns, which implement object systems in other languages. Thus they build partial OBJECT SYSTEM LAYERS. The Type Object Pattern [9] documents a general approach of enhancing an object system with a foreign object system.

In [15] the (mainly black-box) component models of current standards, like CORBA, COM, or Java Beans are discussed. As discussed before, these approaches offer the benefits of black-box component reuse, but have problems, when the internals of a component have to changed or adapted. All approaches have enhancements in the direction of the presented pattern language. The IDLs can be seen as a variant of EXPLICIT EXPORT/IMPORT. Stub and skeleton are a special form of COMPONENT WRAPPERS. Several approaches, like COM interceptors or Orbix Filters, implement BEFORE/AFTER INTERCEPTOR for distributed object systems.

A more general form of such object-oriented abstractions of the message passing mechanisms in distributed systems are composition filters [1]. Abstract communication types represent abstractions over the interaction of objects. Thus they are a variant of BEFORE/AFTER INTERCEPTOR, which may be used to implement COMPONENT WRAPPERS and/or EXPLICIT EXPORT/IMPORT. Roles as in [11], meta-object protocols [10], or meta-classes [3], and several similar approaches to express multiple concerns, impose meta-level behavior over an object. Therefore, these approaches can be used to implement BEFORE/AFTER INTERCEPTOR.

In [2] a component adaption technique based on layers is proposed, which is similar to the presented interceptors: it is also transparent, composable and reusable, but it is not introspective, not dynamic and a pure black-box approach. Layers are given in delegating compiler objects, that are statically defined before compilation time. This makes it hard to use the approach for expressing runtime dynamics in component composition, since changes in layer definitions require recompilations.

The architecture description language Π [5] offers support for EXPLICIT EXPORT/IMPORT. In a process of component configuration the imports can be mapped to exports. But in this concept configurations can not be changed at runtime.

5 Conclusion

We have presented an approach for piecemeal migration of large existing software systems to component technology and a more flexible architecture. An architectural pattern language was used to migrate the existing C implementations to components in an object system.

The existing C implementation is split in a piecemeal process to black-box components wrapped behind COMPONENT WRAPPERS. BEFORE/AFTER INTERCEPTORS let us bring in customizations decomposed and transparent to the component and its clients. An OBJECT SYSTEM LAYER with an explicit DISPATCHER provides a suitable way to implement and maintain the combination of COMPONENT WRAPPER and BEFORE/AFTER INTERCEPTORS. Through EXPLICIT EXPORT/IMPORT components define their required environment. Thus it becomes easy to exchange implementations without interference with clients or imported components. All changes induced by the patterns can be applied in a piecemeal process and existing implementations can be reused.

However, if we do not use an existing OBJECT SYSTEM LAYER implementation, like XOTCL [12], it has to be programmed by hand. This adds more complexity and higher maintenance efforts to the system. Performance can be decreased slightly. The OBJECT SYSTEM LAYERS conventions and interfaces have to learned by the developers.

The presented pattern language is language supported in the scripting language XOTCL. In this paper, we have presented a case study for the pattern language in language-neutral way.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. LCNS 791, Springer-Verlag, 1993.

[2] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41, 1999.

[3] I. R. Forman and S. H. Danforth. *Putting Metaclasses to Work – A new Dimension to Object-Oriented Programming*. Addison-Wesley, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] M. Goedicke, J. Cramer, W. Fey, and M. Groe-Rhode. Towards a formally based component description language a foundation for reuse. *Structured Programming*, 12(2), 1991.

[6] M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of 2nd International Symposium on Generative and*

Component-Based Software Engineering (GCSE'00), Erfurt, Germany, Oct 2000.

[7] M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.

[8] M. Goedicke, G. Neumann, and U. Zdun. A pattern language for introduction of flexiblity into black-box component architectures. to appear., 2000.

[9] R. Johnson and B. Woolf. Type object. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[10] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[11] B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2:143–160, 1996.

[12] G. Neumann and U. Zdun. XOTCL, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[13] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31, March 1998.

[14] D. C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report, SIGS*, 11(2), February 1999.

[15] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. ACM Press Books. Addison-Wesley, 1997.