JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. 2001; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2001/03/16 v1.01]*

Piecemeal Legacy Migrating with an Architectural Pattern Language: A Case Study

M. Goedicke, U. Zdun*,†

Specification of Software Systems, University of Essen, Germany

KEY WORDS: Software Pattern; Pattern Language; Reengineering; Component Architecture

SUMMARY

Numerous large applications that have evolved over many years are well-functioning and reliable, but have severe problems regarding flexibility and reuse. Due to many fixes that were applied in a system's lifetime, it is often hard to customize, change, or exchange system parts. Therefore, it is problematic to migrate such systems to a more flexible architecture or to new technologies. The document archive/retrieval system, discussed in this article, is an example large C system that had such problems. As a solution, we will sketch an architectural pattern language that involves patterns well-suited for a piecemeal migration process. The patterns aim at building and composing highly flexible black-box component architectures with an object-oriented glueing layer. We present a reengineering case study for the document archive/retrieval system based on these patterns. The patterns are used to wrap the existing C implementations and integrate

Contract/grant sponsor: Part of the work has been supported by the Eureka/ITEA CAFE project; contract/grant number: ITEA project ip00004
*Correspondence to: Specification of Software Systems, Institute for Computer Science, University of Essen, Altendorferstrae 97-101, D-45143 Essen, Germany
†E-mail: {goedicke|uzdun}@cs.uni-essen.de

Received 31 January 2000
Revised 12 May 2000

them with an object system. Moreover, the patterns introduce flexibility hooks into the hot spots of the architecture and let components define their required environment. This enables an easier future evolution of the system. The case study demonstrates a pattern language as an approach for piecemeal legacy migration apart from implementation details.

1. INTRODUCTION

The article deals with the problem of large system evolution. It focuses on a particular document archive/retrieval system as an example large C application that has evolved over 16 years. In the current practice of software engineering, it is often a hard task to maintain or reengineer large legacy systems. Usually there is a constant and relentless pressure for changes: bugs have to be fixed, new requirements have to incorporated, design flaws have to be eliminated, etc. Ideally, the developers would be able to evolve a given architecture and implementation in a stepwise manner. That is, the architecture would be incrementally evolved to a higher degree of coherence and required changes would be seamlessly integrated.

However, often the reality is different. In many systems we can observe that with growing complexity more and more architectural and implementational changes are added that violate the purity of initial designs. Step by step the coherence of the architecture degrades. Eventually even simple changes become problematic. Martin calls this process rotting design [20]. Often, in response to such problems, reengineering projects are launched. But such reengineering projects seldomly succeed because monolithic reengineering approaches are usually quite complex. Moreover, in many cases, the developers have to build the new design in parallel to the old one. The old system still evolves and the new system has to keep up with these changes. Thus the new design starts to rot even before the first release.

In this article we present a case study of such a reengineering project for a large document archive/retrieval system. At first, we will give an overview of the system and discuss some problems that have led to the reengineering project. We consider the problems of this system typical for many large legacy systems that have evolved over many years in a business critical setup. Thus, in the first part of the article, we will motivate the typical problems tackled by our case study as well. Moreover, the overview of the system architecture should ease understanding of the concrete reengineering steps discussed in Sections 7–9.

Secondly, in Sections 4–5, we will present a pattern language for stepwise migration of given architectures to a flexible black-box component architecture. In this article, we will not focus on the patterns but sketch them very thoroughly (see [10, 11] for more details on two individual patterns). We will concentrate on the application of the pattern language in a larger industrial context. As we will discuss in Section 11.2 it is one of only a few industrial pattern case studies available so far. The patterns presented are widely applied in the area of scripting language's component glueing concepts, but can be found in many system language projects as well. The pattern language concentrates on the essential fragments of object-oriented component glueing architectures. Thus, the patterns do not require a full-fledged scripting engine to be implemented. However, all pattern implementation's introduce a certain complexity and require a certain maintenance effort. Here is clearly a tradeoff between the additional complexity introduced by the pattern on the one hand and the higher flexibility and cleaner architecture on the other hand. In other words, such liabilities have to be contrasted to the reduction of complexity and higher flexibility in the reengineered legacy system.

Thirdly, in Section 6, we will discuss the concrete considerations of how to apply the pattern language to the document archive/retrieval system. We will see that a stepwise evolution can be achieved without sacrificing the architecture of the given system. Thus one reengineering problem can be tackled at a time, instead of having to change several orthogonal system parts at once. For this reason, we claim that the pattern language leads to incremental evolution of large legacy systems. In our experience, the required implementation effort for the patterns is rather limited compared to the size of a typical large industrial system, as the example document archive/retrieval system. Of course, an existing implementation can be reused, such that there is no direct implementation effort required (but the APIs of the implementation have to be learned by the developers).

Fourthly, in Sections 7–9, we will present three concrete subsystems as prototypical examples of the steps to be performed for reengineering legacy architectures incrementally: the communication subsystem, the DBMS subsystem, and the configuration/customization framework. These were the primary problem domains identified by the company in the beginning of the reengineering project. The company had quite concrete ideas for the resulting solution: a middleware standard should be used for communication, DBMS interfaces should use as a unified form of database access, and the configuration/customization framework should be more flexible and easier to use. Obviously, such requirements for a reengineering project affect completely orthogonal and very complex system parts. Therefore, it is essential for a success of the reengineering effort to deal with one problem domain at a time, instead of changing the whole system at once. We will see that this goal can be achieved for the discussed subsystems.

Finally, we will discuss known uses of the patterns as well as important related work. Since the number of available pattern case studies is currently quite limited, we will summarize the available ones and briefly compare them to our project.

2. DOCUMENT ARCHIVE/RETRIEVAL SYSTEM

The document archive/retrieval system is a large C application that has evolved over many years. The system has a large-scale architecture suitable for its task, and it is working well in numerous customer installations. Note that the authors were not involved in the original development of the archive/retrieval system. They were only involved in the reengineering project presented in this paper. The project was launched because of several flexibility and reuse problems, similar to the ones sketched in the beginning of the previous section.

In the system targeted, flexibility and reuse problems appeared in different forms. Extensions and changes, such as introduction of a new communication technology, integration with object- and component-oriented concepts, usage of reusable service implementations, integration of customization callbacks, and usage of a generic database interface, became more and more problematic. Already used technologies were hard to replace with other implementations or versions. Moreover, deployment to the customer required a certain amount of costly customizations. With the given system implementation these efforts simply took too much time. Besides these severe flexibility problems, the system suffered from minimal reuse of existing software components.

An explicit requirement of the project was knowledge transfer so that the developers of the system could participate in the reengineering effort and apply the concepts on their own. The primary task was not to implement certain changes to the system, but to find a practically

Prepared using smrauth.cls

applicable way to achieve a more flexible architecture and to introduce new technologies in various orthogonal system parts. Thus the example subsystems presented are only prototypical instances. We consider the pattern form as a good way to transfer technical knowledge, because the patterns are rather independent from concrete technology without being too vague to discuss technical details. Sequential applications of the pattern language, that have proven to be successful in practice, can be used to exemplify the application of the pattern language as a whole. The reengineering examples of these three subsystems can be seen as such sequence examples.

In this section, we firstly introduce the existing architecture of the system. Afterwards, we describe the problems tackled by this work in more detail.

2.1. Architectural Overview

A document archive and retrieval system allows users to archive a large number of documents on several sorts of devices. The primary devices used in the application area are optical storage devices, but such systems are also able to access other media types, such as hard drives or remote storage devices in a network. Users can retrieve archived documents with different text- and GUI-based clients. These clients need to be customizable for different customer requirements. Searches are delimited by the client through several search criteria. Such information are stored in an associated database for each record archived by the system.

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. Later on, support for other database management systems was added. The system used optical storage devices with proprietary interfaces. 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.

In Figure 1 we can see a principal overview of the archive and retrieval system architecture. The system consists of three distinct layers. Each of the layers consists of several interacting processes.

- *Client Layer*: *Clients* are used by the customer to archive and retrieve documents. *Loaders* are applications that enable automated loading of information into the archive and retrieval system. An *Administration Client* enables client access to the

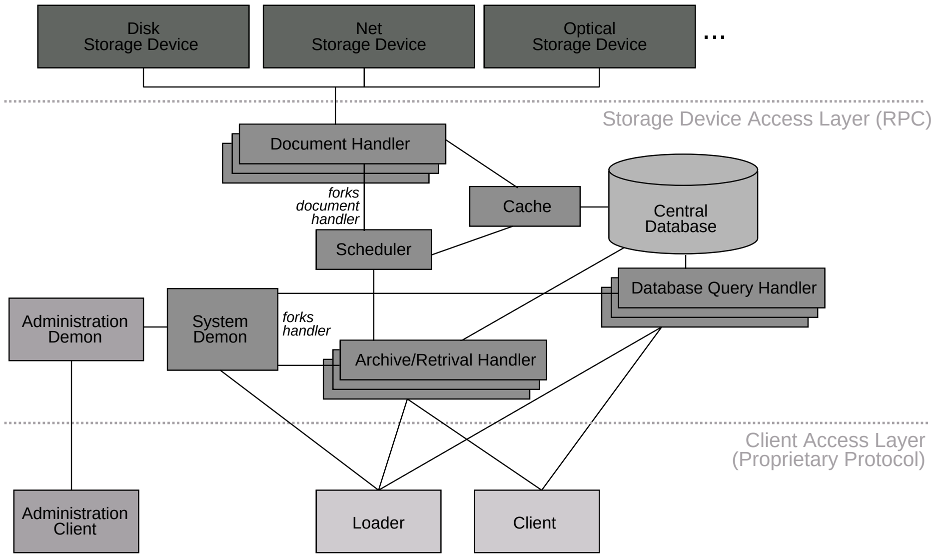

Figure 1. Document Archive/Retrieval System – Architectural Overview

administration of the running system processes, to load plug-ins, to define users and groups, to monitor logs, etc. All clients access the system via a set of proprietary protocols and interfaces (for retrieval, archiving, administration, and SQL access) based on sockets.

- *System Layer*: The *System Demon* is a permanently running demon that forks handlers for client requests. These are either: *Archive/Retrieval Handlers* to access the document management system, or *Database Query Handler* to access the central database. The *Central Database* stores meta-data about the documents for searches. Moreover, it maintains changes to the documents on the optical storage devices, which must be assumed to be read-only after they are written once. A *Scheduler*, a *Cache*, and *Document Handlers* implement the actual archive/retrieval tasks. The *Administration Demon* is a permanently running demon that implements the server-side of the administration tasks.
- *Storage Device Layer*: Several different storage device drivers, such as optical disk jukebox driver, hard disk driver, network access protocols, etc. are unified in this layer behind an API based on a proprietary RPC (remote procedure call) implementation.

2.2. Reengineering Task Synopsis

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 a standard RPC protocol. The used protocols did not contain implementations of more sophisticated services, as can be found in modern middleware approaches. Therefore, the system developers had to implement several such services on their own.

Support of the system on several platforms with several versions became a severe maintenance problem. Moreover, similar problems occurred in maintaining 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 pre-processor directives for different versions were scattered across the entire database code. This is hard to modify centrally and can only be changed by re-compiling the system, thus excluding on-the-fly change of databases.

Configurations of the system for different customer requirements were problematic, since it had to be tightly integrated with the customer's IS infrastructure and with the nature of the archived documents. No two installations of the document management system are exactly the same. The adaption of the system was programmed by hand during deployment using low-level C APIs and shell scripts.

Since many dependencies between different parts of the code existed, it was hard to exchange existing implementations with other implementations. Thus a piecemeal migration seemed to be hard to accomplish. For instance, it was hard to replace cache management strategies for archiving/retrieval, because they were not encapsulated in distinct components with well-defined interfaces. Moreover, some parts of the system, such as the communication subsystem or database access were even harder to exchange, because they were not implemented with distinct APIs.

Still the existing software was working in an efficient and reliable way. Thus, as far as possible, existing (and well-working) system parts should be reused in a new solution. The system installations were crucial parts of the customers' IS infrastructure. In general, it is not trivial to migrate a running installation with all documents to a new technology. Several customers are even suspicious about "new" and – from their viewpoint – unproven technologies because document archiving is an important and integral part of their business. Therefore, the new technologies should have been wrapped in the appearance of the well-known and reliable technology.

The overall architecture of the system, as sketched in Figure 1, 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 be changed completely.

Initially, a reengineering strategy 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 a different architecture and potentially new errors. This would require a considerable redesign effort, but the value of such an effort is questionable.

3. MULTI-PARADIGM DEVELOPMENT AND COMPONENT-ORIENTED STRUCTURING

A central observation for our case study is that most software systems of larger application scale are structured with multiple paradigms. In general a software paradigm is a set of patterns and rules for abstractions, partitioning, and modeling of a system. There are some design time approaches for multi-paradigm integration proposed, such as [30] and [5]. However, they do not (yet) focus on the technical integration of the paradigms that are (in part only implicitly) used in existing systems.

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. Some usages of foreign paradigms are not even recognized by the developers themselves.

E.g., the component-oriented paradigm seeks for reusable, black-box building blocks as the primary commonality aspect. In [27] an object-oriented white-box framework is characterized by primarily using inheritance. But inheritance results in strong coupling between components. Making a new class involves programming and potentially also re-programming existing parts. It may become difficult to understand a system's composition in complex white-box frameworks. In many languages inheritance is static what hinders runtime changes.

A black-box framework composes components at runtime. Variations are primarily achieved by means of parameterization. Thus it may be hard to cope with required changes in the black-box and to flexibly adapt interfaces. If we consider the system, as presented in the previous section, we can already identify several C components that satisfy such criteria. These components are nearly identical to process boundaries. Thus, it is generally possible to factor out existing components and support them with a component model.

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 [25, 9] or in modern middleware architectures (see [31]), are missing. In this article we will present a pattern language that helps in incrementally factoring out components from a given system and for incrementally building a context- and domain-specific model for component integration.

Based on our experience, the missing component integration is a central reason 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 interfering with their client's implementation. There is no concept for runtime loading of components, and introspection of loaded components is missing as well. Therefore, loaded components are not traceable at runtime. For these reasons components are hard to exchange with other implementations. There is no way to introduce adaptations at a central place, say, when interfaces or requirements change gradually.

Therefore, the system should be enhanced with a component model that let us define explicit interfaces. It serves as a central indirection layer that handles issues, such as dynamic component loading, component adaptation, paradigm integration, etc. Nevertheless, the existing C implementations should be (re-)usable as core implementations.

4. PATTERNS FOR FLEXIBLE, COMPONENT-ORIENTED ARCHITECTURES

To migrate the system to a new component-based architecture we use an architectural pattern language (see [10, 11] for a more detailed discussion of some of the patterns). The aim of the pattern language is to generate flexible black-box architectures in a piecemeal way. Here, for space reasons, we can only sketch the patterns with context, problem, solution, architectural overview, and consequences. The interdependencies of the introduced patterns among each other and with several popular object-oriented software patterns are sketched in the next section. The reengineering case study in the remainder of this article will present examples for the pattern usages and possible usage sequences.

Pattern I: Component Wrapper

Context Black-box components leverage reuse and ease understandability of a system to a higher degree than white-box frameworks because the components can be reused without intimate knowledge of the component's internals. The component's internal implementation becomes replaceable. But black-box components are hard to customize, since the black-box property means that parameterizations are the primary variation technique.

Problem How can we customize and adapt a component to a degree that goes beyond parameterization without sacrificing the black-box properties?

Solution Let accesses to an integrated system component be represented by a first-class object of the programming language. Indirection by a Component Wrapper object gives the application a central, white-box access point to the component. Here, the component access can be customized without interfering with the client or the component implementation.

When we want to integrate a foreign paradigm component into object-orientation, we apply the Wrapper Facade [28] pattern. For object-oriented components we can use Proxies [7] to get placeholders for the component's objects on the client side. Easily Decorators [7], Adapters [7], and other kinds of extensions/changes/customizations can be added to the Component Wrapper.

Architecture Component Wrappers occur in different styles (see Figure 2). *Wrapper Facade [28] Component Wrapper* integrates foreign paradigm/language components into an

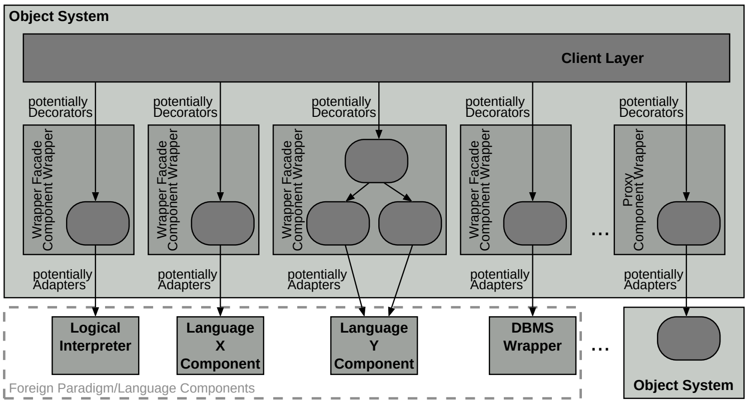

Figure 2. COMPONENT WRAPPER – Architecture

Object System. Components written in the object system itself can be integrated into the application by a *Proxy [7] Component Wrapper* that exposes the integrated component interface as a first-class object to the application.

Component Wrapper Decorators enhance or customize the component from within the application. Thus, they are mostly used for customizations with respect to behavior. *Component Wrapper Adapters* allow us to access different component versions, internal implementations, or to fulfill interface adaptations. Both, DECORATORS and ADAPTERS, are optional in a COMPONENT WRAPPER architecture (they often represent an architectural evolution).

Often the component wrapper object cannot yield the desired language functionalities because the base language does not support them natively. Here, OBJECT SYSTEM LAYER can be used to implement a customized "little" language on top of the base language. BEFORE/AFTER INTERCEPTOR is an example of a functionality, not supported by many system languages, which can be applied on COMPONENT WRAPPERS by using OBJECT SYSTEM LAYER as an integrating architecture for several COMPONENT WRAPPERS.

Consequences

+ Customizability of black-box components is enhanced.

+ Different styles of wrapping are conceptually integrated.
+ Application developers can flexibly adapt and change interfaces to components at a central access point.
+ An external component is used as an internal, first-class entity of the object system.
+ The component use is decoupled from the internal realization. Therefore, the component itself is exchangeable with another implementation.
– The indirections to the COMPONENT WRAPPER reduce efficiency slightly.
– In the class-based design the conceptual entity "component" may be split up into several entities (wrappers, DECORATORS, ADAPTERS, etc.).
– More classes and flexibility hooks can result in a higher descriptive complexity of the application and thus reduce understandability.

Pattern II: MESSAGE REDIRECTOR

Context When black-box components are glued together, one can only rely on the interfaces provided by the components. As systems evolve, interfaces of black-boxes tend to change. Thus clients have to cope with such changes. COMPONENT WRAPPERS are a central access point to one component, where we can apply specific changes. However, changes that affect more than one component, more than one component version, other subsystem objects than the COMPONENT WRAPPER, etc. have to be propagated through the code. This is because dealing with such concerns, that are cutting across various objects or components, requires to have a control over the message flow to all objects and components affected by a change.

Problem How can we gain control over the message flow in an object-oriented (sub-)system, so that we at least can trace and modify the messages and their results?

Solution Build an explicit MESSAGE REDIRECTOR instance that is called with a symbolic call, every time a message should be sent to a certain (sub-)system. The message itself is given as an argument to a call of the MESSAGE REDIRECTOR. The MESSAGE REDIRECTOR maps the symbolic calls to actual implementation objects/methods.

Architecture A MESSAGE REDIRECTOR handles message calls for a set of *Interacting Objects* that form a *Subsystem* (see Figure 3). Note that the objects might be objects of the base language, but they might also be newly constructed (e.g. in the context of OBJECT SYSTEM LAYER). None of these objects communicate directly with each other. Instead a

Prepared using smrauth.cls

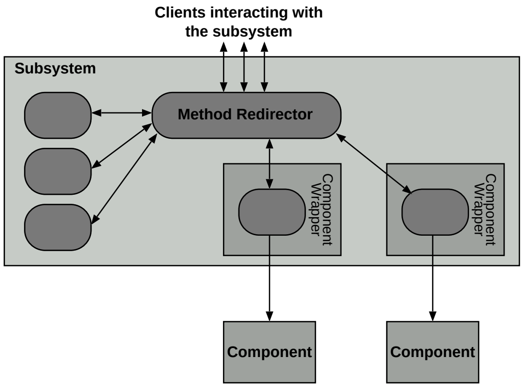

Figure 3. MESSAGE REDIRECTOR – Architecture

central *Message Redirector Object* dispatches the object calls through simple dispatch scheme $Message \rightarrow MessageImplementation$. Usually, all object interactions in the subsystem are handled via the MESSAGE REDIRECTOR.

The symbolic messages can, for instance, be strings of the form "*object method arguments*." The MESSAGE REDIRECTOR redirects the symbolic call to the implementation object, on which the symbolic method name is looked up. Thus, each MESSAGE REDIRECTOR contains abstractions for objects and methods, sometimes for classes as well. On top of this architecture we can implement object-oriented abstractions, such as classes, delegation, aggregation, inheritance, etc. These "higher-level" abstractions distinguish MESSAGE REDIRECTOR from the COMMAND pattern [7], which only performs an indirection by means of subclassing in the base language, and COMMAND PROCESSOR [3] that adds an additional processor to COMMAND (e.g. to support undo and redo functionality on the commands).

Furthermore MESSAGE REDIRECTORS usually provide the possibility of registering callback methods that are called when certain criteria are satisfied. Those can be used for interface adaptations and similar tasks. Thus, the MESSAGE REDIRECTOR can, for instance, act as an ADAPTER [7] for changing interfaces.

Consequences

+ All outgoing calls from a (set of) client(s) are bundled. They can be centrally modified and adapted.
+ Simple MESSAGE REDIRECTORS can be implemented very efficiently.
+ MESSAGE REDIRECTOR implementations usually do not require more complicated client code.
+ Adaptations and extensions in the MESSAGE REDIRECTOR are transparent to the client.
+ Used as a FACADE, the MESSAGE REDIRECTOR shields a subsystem and provides an uniform way of access.
+ The MESSAGE REDIRECTOR can be used to provide flexibility through highly programmable interfaces.
– All MESSAGE REDIRECTORS consume additional computation time.
– If a MESSAGE REDIRECTOR is used in parallel with ordinary method calls, two different styles of method calls are present in a system.
– Without language support it is hard to enforce that a MESSAGE REDIRECTOR is not bypassed by ordinary method calls.

Pattern III: OBJECT SYSTEM LAYER

Context Often object-oriented approaches can be used throughout the design, but (parts of) the implementation may be realized in a language that does not support object-orientation, such as C or Cobol. Or the implementation may be realized in an object system that does not support desired object-oriented techniques as, for instance, message interception or reflection, such as C++ or Object Cobol. Often the use of such languages is imposed by business decisions.

Problem How can we use high-level object-oriented functionalities when we are faced with a target programming language that is non-object-oriented, or with legacy systems that cannot quickly be rewritten, or with target object systems that are not powerful enough or not properly integrated with other used object systems (e.g. when COM or CORBA is used)?

Solution 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. Use COMPONENT WRAPPERS to provide well-defined interfaces to non-object-oriented components or components written in other object systems. Use MESSAGE REDIRECTOR to map symbolic messages to the OBJECT SYSTEM LAYER objects.

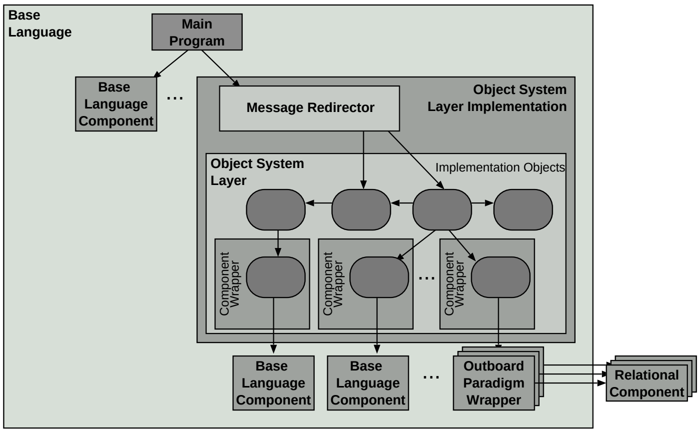

Figure 4. Object System Layer – Architecture

Architecture Object System Layers extend a base language, such as C or C++, with a dynamic object system (see Figure 4). The *Object System Layer* integrates other base language components wrapped by Component Wrappers, and it is itself a base language component. It internally uses a Message Redirector to map calls to the implementation objects.

This architecture is primarily chosen for flexibility reasons. It let us define a "little" component glueing language on top of a given base language. Thus we can customize the component integration relationships to our concrete domain. By using Message Redirector and Component Wrappers the Object System Layer architecture usually integrates well with the base language. The architecture mimics the architecture of an object-oriented scripting language implementation in a very simple fashion. However, a simple Object System Layer is far from a full-fledged scripting engine. If such functionality is required, it may be a good option to embed an existing scripting language implementation and support the other patterns in this language, instead of implementing Object System Layer from scratch.

There are several patterns implementing object systems in other languages. Thus they build partial Object System Layers. The Type Object pattern [14] documents a general approach of enhancing an object system with a foreign (dynamical) object model.

Consequences

+ The flexibility of the application can be dramatically improved, since OBJECT SYSTEM LAYER allows us to introduce many flexibility hooks and to implement high-level language constructs, such as interception, reflection, and adaptation techniques.
+ Custom component relationships and other relationships that rely on runtime dynamics can be "language supported" by the pattern in languages that do not offer such functionality natively.
+ Classes can become runtime entities and can be manipulated with the same ease as objects.
+ The complexity of the application can be dramatically reduced, because techniques, such as dynamic classes, aspects, roles, mixins, filters, etc., often enable us to avoid unnecessary explicit class definitions. Thus we avoid a combinatorial explosion of subclasses.
– Complexity of the application can also rise, if the client has to maintain the OBJECT SYSTEM LAYER. Then issues such as garbage collection, object destruction, relationship dynamics, reflection, etc. have to be programmed by hand. But this problem can be avoided by using an existing OBJECT SYSTEM LAYER as a library.
– Performance can be decreased through the additional indirection and through flexibility hooks.
– The OBJECT SYSTEM LAYER's conventions and interfaces have to be learned by the developers. If the OBJECT SYSTEM LAYER is an entire scripting language, a new language has to learned.

Pattern IV: EXPLICIT EXPORT/IMPORT

Context Export and import are abstract views onto the realization of a component. These views define the required environment of a component. In many implementations the export/import information is (partly) lost. In the context of reengineering to a component-based architecture, we usually have to stepwise change interfaces and connect them properly again. Implicit interfaces often result in inconsistencies between interfaces. Those, in turn, often result in hard to find errors and inelegant workarounds. Documentation of changing implicit interfaces are usually hard to be kept consistent as well.

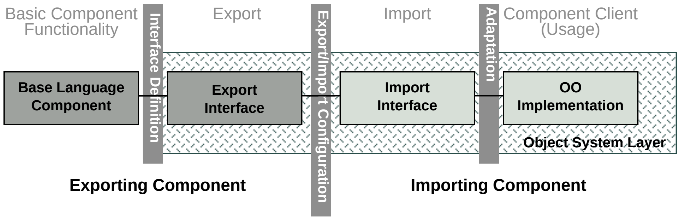

Figure 5. Three-Level Component Configuration with Explicit Export/Import

Problem How can we let a component define its required environment at runtime, so that we do not have to keep track of changes in used components and propagate changes in a component to all clients?

Solution Make export and import interfaces of a component accessible objects of the programming language. Then connections of export to import interfaces mean a runtime-traceable component connection that transparently glues two components together in an explicit way. Through explicit export and import interfaces and their respective connections, we can check the consistency of component configurations at runtime.

Architecture At runtime Explicit Export/Import relies on the concept of *component configuration* [8]. The pattern's solution often results in a three-level component configuration: a component is configured to an export interface, an import interface is configured to an importing component, and the export interface is configured to the import interface. In Figure 5 we can see the general architecture of connecting foreign language components to components written in an Object System Layer.

This architecture allows (a) for flexible component connections and (b) for coping with inconsistencies during evolution of the system. Often the Object System Layer supports introspection options to retrieve the current component configuration, so that consistency can be checked at runtime.

Consequences

+ Explicit Export/Import makes a component self-contained, and it eases the usage and deployment, say, in new component environments.

+ Explicit Export/Import can enhance understandability, since component dependencies are documented at a central place in the component's code.
+ Components can be exchanged with other components transparently.
– When import objects in Proxy/Adapter style are used, we have to split up a conceptual entity into several runtime entities. Besides this conceptual problem, the solution can reduce performance slightly.
– In some variants of the pattern the interfaces are not locally defined with the component, but have to be searched in so-called component connectors.

Pattern V: Before/After Interceptor

Context "Indirection of the messages" means that a message is redirected to a set of other receivers before/after it reaches the original receiver. Often we require such indirections for Component Wrappers and Explicit Export/Import, say, for checking interface consistency, for ensuring runtime contracts, for adapting to other interfaces, for enhancing a component's functionality dynamically, etc. There are several other patterns, such as Decorators or Chain of Responsibilities, and more complex relationships, such as relationship objects or runtime ensuring of contracts, that have similar kinds of indirections around messages.

Problem Indirections of messages, as in Component Wrapper, Explicit Export/Import, and similar patterns, are recurring in many contexts. But we have to program the before/after indirections of messages by hand for each occurrence of the patterns. Usually the implementations are scattered across the code. How can we ensure that a certain callback method is executed every time before a message is called and/or after a message is called on a certain object/component?

Solution Built a callback mechanism into the Message Redirector that triggers a certain callback whenever certain criteria for a symbolic call are satisfied. Provide both before and after callbacks. Those are implemented as objects of the Object System Layer and can be dynamically customized at runtime.

Architecture Before/After Interceptor need not come with an explicit architecture because often programming languages offer more or less convenient ways to implement Before/After Interceptors. Several languages offer language means to express callbacks, scoped objects, traces, etc., which can be used to express the indirection. If no appropriate constructs for the before/after indirections can be found, then the construct can be built with a MESSAGE REDIRECTOR (e.g., as a part of an OBJECT SYSTEM LAYER).

If a MESSAGE REDIRECTOR is used, the MESSAGE REDIRECTOR has to be able to offer callback methods, where the BEFORE/AFTER INTERCEPTORS can be registered. The BEFORE/AFTER INTERCEPTORS are called every time a message passes the MESSAGE REDIRECTOR that conforms to a certain condition. For instance, the callback execution can be discriminated on object name, method name, calling object, calling method, etc.

Typical applications of this pattern are, for instance, powerful debugging and interface adaptation mechanisms. Those are often required for stepwise system evolution. MESSAGE REDIRECTOR with BEFORE/AFTER INTERCEPTOR can, for instance, be used to implement Aspects, as in [18]. Aspects are concerns that have to be coordinated across component boundaries.

Consequences

+ Controlling behavior can be transparently imposed over an object.
+ BEFORE/AFTER INTERCEPTOR allows us to implement, support, and language support several patterns relying on message exchanges.
+ We can define orthogonal behavior decomposed from an object, but at runtime object and extension appear for a client as a single entity.
+ Most BEFORE/AFTER INTERCEPTOR approaches enable us to use an extension dynamically. With an OBJECT SYSTEM LAYER we can define a new introspection option to query at runtime which behavior is currently added to an object.
− Indirection through an BEFORE/AFTER INTERCEPTOR may cost performance.
− If no proper language construct for implementing the before/after interception task is available in the programming language, the interception mechanism has to be implemented from scratch. Since we require a MESSAGE REDIRECTOR and possibly an OBJECT SYSTEM LAYER this might be too much an implementation effort for smaller projects.

5. AN ARCHITECTURAL PATTERN LANGUAGE FOR PIECEMEAL LEGACY MIGRATION

In Table 5 we give a short summary for some popular patterns that are used in the compound patterns presented in the previous section. All these patterns are part of the pattern language presented. In Figure 6 an overview of the most important pattern dependencies in the context of building a flexible glueing architecture for component frameworks with the patterns is depicted.

The patterns have rich interactions, and the evolving pattern language has a clear structure. Object System Layer [10] is the pattern of the largest granularity in the pattern language. It introduces a layer providing a highly flexible object system on top of a given language. Object System Layers may provide flexibility for component assembly as offered by dynamic object-oriented scripting language, such as XOTcl [24], OTcl, Python, Ruby, or Perl. But the object system is nevertheless integrated with the base language. In the case study presented, we use Object System Layer as the basics for a piecemeal approach of reengineering because it let us implement the targeted dynamic component concepts, but it does not require us to entirely switch to a different programming language.

But Object System Layer is only useful if such a flexibility in the architecture is required. Otherwise it can cause an unnecessary performance overhead and an additional effort to understand the Object System Layer's language or API for the developers. Moreover, if no existing Object System Layer can be reused, programming and maintaining one can add unnecessary complexity to the software system.

A Message Redirector is a vital part of many Object System Layers. Moreover, it provides a simple and efficient indirection architecture that maps calls to implementations. It often allows one to define callback methods around the calls. These callback methods are the basics for the use of Explicit Export/Import and Before/After Interceptor. But Message Redirector may also enable the "Layer" property of an Object System Layer by being a Facade to the subsystem defined by the Object System Layer.

There are several patterns that we can use to integrate components into an Object System Layer. Wrapper Facade let us integrate functions into an object environment. Proxy can shield an object-oriented component. Component Wrapper describes how to treat such integration patterns and integration of other paradigms, such as a logical interpreter

Prepared using smrauth.cls

Table I. Additional Patterns in the Pattern Language: Pattern Thumbnails

Name	Problem	Solution
WRAPPER FACADE [28]	How can we integrate a procedural component, such as a library consisting of C functions, into an object system?	Provide a (set of) objects that wrap the functions and model them as first-class objects. These object forward to the functions.
PROXY [7]	How can we interact with an object that cannot or should not be accessed directly?	Provide a placeholder for the object with the same abstract interface and forward all calls to the actual implementation object.
DECORATOR [7]	How can we attach additional responsibilities to an object dynamically?	Use a chain of DECORATORS that delegate to the actual object. Let the DECORATORS have the same abstract interface as the object and let clients refer to the first DECORATOR in the DECORATOR chain.
ADAPTER [7]	How can we convert the interface of a given class into another interface that a client expects?	Wrap the object call with an ADAPTER that converts the object call to another interface and forwards to the adaptee.
FACADE [7]	How can we make a complex subsystem easier to use for clients and how can we enhance the interface reliability of the subsystem?	Provide a unified interface to a set of interfaces in a subsystem. FACADE defines a higher-level interface that shields the subsystem from direct access.

or a relational DBMS, in a unique way. Moreover, COMPONENT WRAPPER integrates these foreign paradigms into an OBJECT SYSTEM LAYER, and enables adding decoration/adaptation callbacks uniformly. Finally, it enables the usage of EXPLICIT EXPORT/IMPORT to provide canonical components.

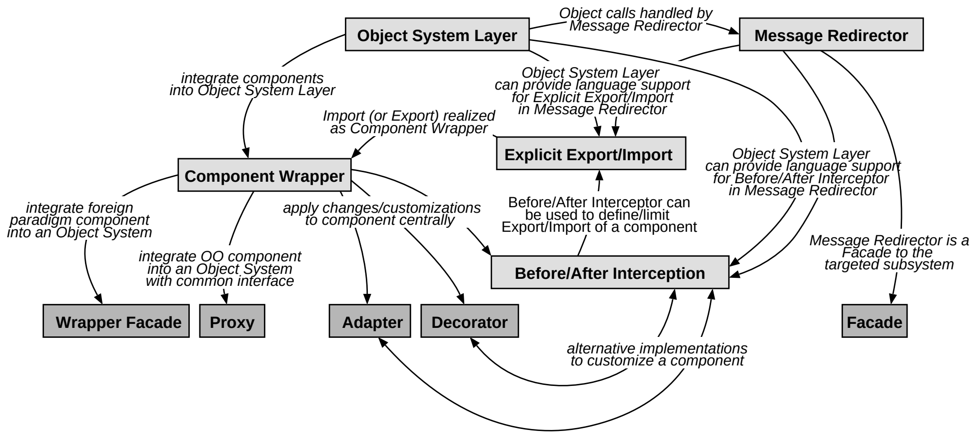

Figure 6. Pattern Language: Pattern Relations and Interdependencies

Component Wrapper handles the flexibility of component wrapping by explicitly defining callback methods for Decorators and Adapters. These can either be programmed by hand with Adapter and Decorator patterns or they can be "language supported" in the Object System Layer by Before/After Interceptor.

6. DOCUMENT ARCHIVE/RETRIEVAL SYSTEM COMPONENT INTEGRATION

For the reengineering project we have to split the existing C implementations into a component architecture. The aim is to enable reengineering of each orthogonal subsystem on its own and in a piecemeal fashion. Moreover, we want to be able to exchange components with other implementations and cope with version changes. A split into separated interfaces is an important part of the task, but it is not sufficient alone for stepwise evolution. Each interface change in one of the component export interfaces (which usually have to be supported in different versions) would cause changes in clients as well. New client requirements would have to be propagated into the components. Runtime changes of interfaces would be almost impossible. There would be no central support for concerns that are cutting across different components, such as many debugging tasks or adding security features. There would be no significant gain in the system's flexibility regarding new versions/components.

An OBJECT SYSTEM LAYER with a MESSAGE REDIRECTOR aims at exactly these problems with high-level object-oriented abstractions. COMPONENT WRAPPER are especially useful for encapsulating legacy components. Thus, the component architecture will be composed from COMPONENT WRAPPERS to the legacy components in an OBJECT SYSTEM LAYER. The OBJECT SYSTEM LAYER handles necessary callbacks and adaptations.

Step by step we will build a compatible interface for each orthogonal system part that is encapsulated in a COMPONENT WRAPPER. With EXPLICIT EXPORT/IMPORT we will define the component interfaces explicitly, so that we can retrieve them at runtime and check consistency. BEFORE/AFTER INTERCEPTORS can be applied for adapting interfaces and for coping with different component versions. Moreover, powerful debugging features can be implemented rapidly.

These elements of our component architecture are conceptually integrated into an OBJECT SYSTEM LAYER with a MESSAGE REDIRECTOR. Thus we define a new little object-oriented language on top of the base language that contains abstractions satisfying domain-specific requirements in the given system.

In a first step, we use the distinct C written subsystems, forming the processes of the legacy system, as components. Then we will refine these components in a piecemeal effort. We have to choose an object system (in order to apply OBJECT SYSTEM LAYER). Then we have to wrap the components with this object system. We have the following choices:

- We can use a mainstream object-oriented language, such as C++ or Java with JNI, and integrate the C components.
- We can build an object system as a library in C, as discussed in [10].
- We can use an existing OBJECT SYSTEM LAYER implementation, as for instance the object-oriented scripting language XOTCL [24].

Here, we propose to use an object-oriented scripting language, such as XOTCL, because this language supports several of the patterns: It is itself an OBJECT SYSTEM LAYER, it implements a MESSAGE REDIRECTOR, and it offers language constructs (filters [22] and mixins

[21]) implementing Before/After Interceptors. Moreover, as we will see, a full-fledged scripting engine is a good choice for many configuration tasks.

Note that a simple implementation of an Object System Layer only takes a few hundred lines of code. A full-fledged scripting engine is far more implementation effort and takes a considerable learning effort for the developers to learn a new language. Those issues have to be carefully considered. Usually, we can only find the proper solution system- and domain-specifically.

In this article we present the reengineering case study with the pattern language independent of the used object system. XOTcl itself is a C library implementing an object system. Therefore, the usefulness of the pattern language is independent of the used object system/programming language. We could use another C library, implementing an Object System Layer, in the same way.

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). Afterwards we connect the export interfaces with the existing implementations, as in the Component Wrapper pattern (the wrappers initially simply forward to the implementations). Now we have a first class representation of all components in the object system. The component representations define their export parts for the Explicit Export/Import pattern. The Component Wrapper can be used to apply changes and adaptations, while the components are used as independent black-boxes.

These components are initially just plugged together again. A minimal implementation in the object system controls the components with the Message Redirector. Now we can extract the hot spots [26] of the application, that is, the parts which are likely to change often, and integrate them stepwise into the implementation in the object system. In the next sections, we present three such hot spots and their migration: the communication subsystem, the database integration, and the configuration and customization framework.

7. PIECEMEAL INTEGRATION OF A CORBA COMMUNICATION SUBSYSTEM

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, 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 a few basic services. Therefore, several services that are known from popular middleware systems, such as 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 with another technology.

The task for reengineering of the document archive and retrieval system's communication was to find an architecture, enabling the use of modern middleware technologies and to benefit from their reusable service implementations. 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.

There were several possible solutions to these problems: The proprietary 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. Note that none of these solutions is absolutely superior and each has certain problems.

For instance, in the concrete project a CORBA middleware solution was imposed by the company. It should serve as a 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, such as messaging service, naming and directory service, transaction service, security service, etc.

However, a monolithic middleware platform is relatively complex and implementations are often rather vendor dependent.

Here, we do not want to further reason why CORBA is used, but explain how the patterns presented can be used to implement a CORBA communication service in a way such that it is easily exchangeable with other communication components. The primary reasons for this design are (a) to be able to cope with future communication requirements and (b) to support different communication channels as well. For instance, during the project it was already foreseeable that a web-based representation will be required soon.

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 across the code. And we can build an equal, generic import interface with EXPLICIT EXPORT/IMPORT. Firstly, the communication classes, such as `AdminComm`, `ArchiveComm`, `RetrievalComm`, and `SQLComm`, are nearly identical to 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 [7] to this class for piecemeal legacy migration.

For each type of communication functionality we simply build 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 still we have to realize the communication component with the middleware of choice (here: CORBA).

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. Besides the IDL, we require import and export interfaces for each component in order to benefit from architectures as induced by the EXPLICIT EXPORT/IMPORT pattern. At this point in our reengineering project import, export, and IDL interface carry the same information. Therefore, we propose to use an automatic generation of export interface and IDL from the import, as in shown Figure 7. Later on, in many case further evolution of the system will lead to differences in export and import interfaces.

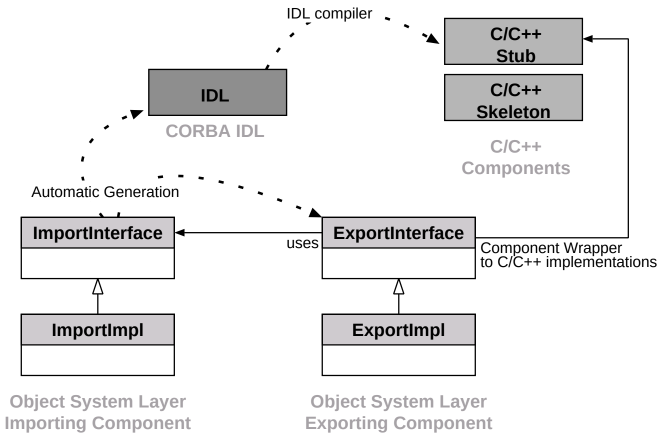

Figure 7. Automatic Generation of the IDL from Interface Class

The import interface is written by hand. A generation tool produces corresponding export and IDL interfaces. With an introspective scripting language, such 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 slightly 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, only forwarding 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.

8. GENERIC DATABASE INTERFACE

The database management system integration faced similar problems as the communication subsystem. The code for DBMS access of the two used DBMSs was scattered across 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 cannot work with the existing version. This 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 existing solution consists mainly of `ifdef` pre-processor directives of the following style that were scattered across 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 8 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. 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

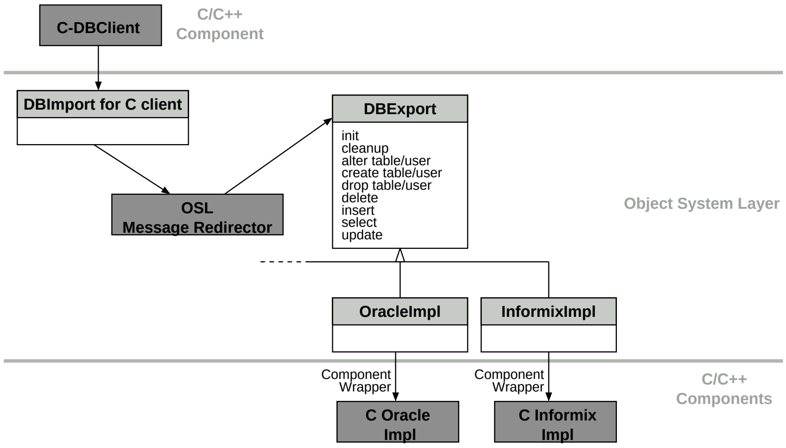

Figure 8. Database Interface With an Object System Layer

the calls, if necessary. Thus we have to pass all calls through a MESSAGE REDIRECTOR that invokes the BEFORE/AFTER INTERCEPTORS.

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 MESSAGE REDIRECTOR 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 MESSAGE REDIRECTOR can adapt the calls to the changes.

Note that the primary achievement of the work, discussed in this section, was factoring out the conditional code and componentizing the database access. Through the layering in the OBJECT SYSTEM LAYER we did achieve a better structuring of the database access layer. Here, we did only discuss a quite simple and straightforward mapping to objects of the OBJECT SYSTEM LAYER. Further evolution steps to a “real” object/relational access layer are still required. The flexibility of the OBJECT SYSTEM LAYER architecture should ease such steps, but it does not make them obsolete.

However, other patterns for object/relational access have to be applied in order to reengineer the simple access form presented to a more sophisticated solution. Our solution is mainly characterized by providing an object- and component-based access form which is relatively compatible to the original system (what eases, for instance, migration of regression tests). Additionally, a new object/relational access layer can be added in further reengineering steps. The interested reader can find a good summary of object/relational access layer patterns in [15]. However, a deeper discussion of such object/relational access layers goes beyond the scope of this case study because (a) it can rather be considered as a future enhancement and (b) it is relatively independent of the pattern language presented. The important contribution of the patterns applied is that they allow for piecemeal integration of more sophisticated solutions on top of the solution presented. This is because the new solution is centralizing, abstracting over, and componentizing the DBMS access.

9. CONFIGURATION AND CUSTOMIZATION FRAMEWORK

An important task for the document archive and retrieval system is the configuration and customization of the system with recurring tasks. E.g., some customers have to archive a certain directory from time to time. Some of these tasks can be launched automatically, e.g., every 24 hours. Moreover, the system has to be adapted to specifics of the customer's IS infrastructure and of the archived documents.

For such tasks a small language extension for the korn shell to embed applications as plug-ins into the system was originally introduced. In a first implementation these scripts have just called a binary that executes in a forked process. But the korn shell was not expressive enough for all requirements of plug-in initialization, execution, and termination. Therefore, a set of new key words was added and a small parser/compiler was implemented to evaluate them. Such a script contained the following sections to define a configuration script:

- *head* – Comments, version labels, actions before plug-in execution.
- *var* – Global Variables.
- *func* – Callable Functions.
- *begin* – Plug-in initialization.
- *loop* – Looping body of the plug-in execution.

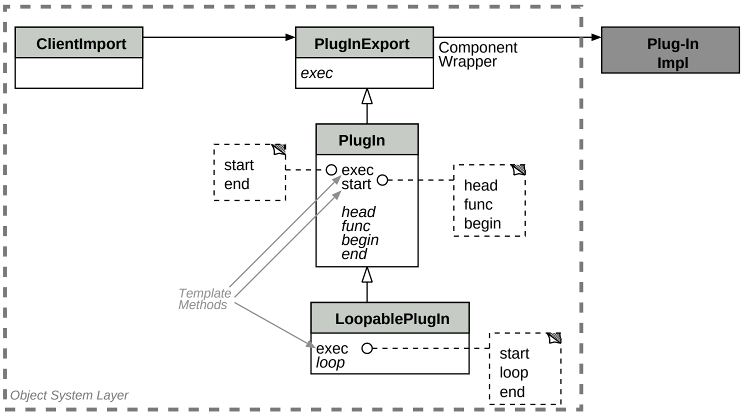

Figure 9. Configuration with TEMPLATE METHOD Framework

- *end* – Termination of the plug-in.

All in all, a small scripting language for plug-in execution was implemented. What seemed in first place as a small effort of implementation and maintenance has evolved over the time into a complex language. Since the underlying korn shell language is not designed for language extension, it was rather difficult to add quite simple additions and it resulted in an obfuscated syntax. Moreover, the solution was neither dynamic nor introspective at runtime. Thus only limited changeability during runtime is offered by this solution. To add an extensible scripting language as an OBJECT SYSTEM LAYER to the plug-in loader, would have meant less efforts and a far better design of the extension.

Again, we wrap the C plug-in component with a COMPONENT WRAPPER. Thus, we can stepwise migrate each plug-in in the existing repository to a solution in the OBJECT SYSTEM LAYER. In the export interface of the plug-in component we implement the little language that is required for configuration of the plug-in with a TEMPLATE METHOD framework. Special plug-ins are derived from the plug-in classes and do only specialize the abstract methods. Clients import the customized plug-ins with an EXPLICIT EXPORT/IMPORT and only use the `exec` method to execute the customized plug-in classes. Customizations can either be introduced in the plug-in classes or with BEFORE/AFTER INTERCEPTORS. We can program generic plug-in types, that are recurring for many customers, in C and reuse these implementations, instead of changing the C code for every installation.

10. OVERVIEW OF THE RESULTS

In the previous sections, we have presented four crucial excerpts of a larger reengineering project: the introduction of component-oriented concepts, the integration of CORBA communication component, the integration of a generic DBMS interface, and a flexible configuration framework. In the case of the document archive system there were several other "hot spot" areas of the system that were tackled by the reengineering project, including cache strategies, distributed archiving, integration of storage devices, automated clients, integrating web services, etc.

In this section, we will firstly discuss the general sequences used to apply the patterns and summarize the results. Secondly, since our work is a case study, we will discuss the generality of the results.

10.1. Reengineering Effort Summary

All sub-systems, in focus of the reengineering project, were treated in a similar manner, following a simple scheme: First, extract the targeted component's implementation into a distinct component. Provide a simple indirection through the OBJECT SYSTEM LAYER using the old interface. Then extract system hot spots, stepwise into the OBJECT SYSTEM LAYER, as architectural problems appear. Evolve the component interfaces incrementally.

In the case study presented, we have used the named excerpts as typical examples for the demonstration of a pattern-based approach of piecemeal architecture evolution. The patterns base on concepts known from object-oriented scripting, component glueing, architecture specification, and so-called meta-programming techniques. In general, the BEFORE/AFTER INTERCEPTOR can be used to implement different meta-programming techniques conveniently, such as aspects [18], roles [19], filters [22], and mixins [21].

We have seen that the pattern-based approach enables us to integrate such approaches with mainstream programming languages and environments, such as C, C++, and Java in the presented case study. Moreover, through abstraction from the concrete realization of the patterns, we can discuss "high-level" issues, such as component integration, component adaptation, or component export/import, even in early project phases in detail.

The discussion presented should mainly demonstrate the ability of the pattern language to build architectural evolution in a piecemeal way. Through component wrapping in the OBJECT SYSTEM LAYER we can implement (and discuss) each new component on its own, without violating an existing design. Thus we can achieve structure preserving transformations of a given software system towards a component-oriented architecture. The resulting architecture enables access to architecture hot spots in the fashion of object-oriented scripting languages. Architectural problems can be treated local to the problem. The change is transparent to clients of the component in which the architectural problem is situated. Interfaces are represented explicitly and can be traced at runtime.

10.2. Discussion of Generality of the Results

This article has given an overview of a pattern-based approach towards improving design and maintainability of large existing software systems. We have briefly sketched the core patterns used. They are discussed in more detail in [10, 11]. In our experience, the pattern language has shown its usability in various projects. The problems identified for the document archive/retrieval system are typical in many legacy systems. Thus, the case study presented and the general outlined approach for stepwise legacy evolution can be seen as applicable to most legacy systems with a similar history. However, for large scale design decisions, as the ones in the project presented, the forces and consequences have to be carefully considered before applying the pattern language.

Of course, there is a set of limitations regarding the general applicability of the pattern language. The pattern language represents a core of patterns for the purpose of gaining architectural flexibility and evolvability. Many more specialized patterns still have to be mined in several sub-domains. For instance, as we have seen in the case study presented, the pattern language works for integrating the core concepts of component glueing into a given C system. However, there are important building blocks missing in the pattern language to go substantially beyond that, say, for the purpose of implementing a complete domain-specific glueing language on top of OBJECT SYSTEM LAYER from scratch.

We have discussed three prototypical example subsystems. These were only a small part of the case study presented. However, the diversity of domains and problems in these three subsystems, on the one hand, and the similarity of the sequences in which we have applied the pattern language, on the other hand, should have shown that the general approach is applicable to many different domains. The pattern-based approach is relatively independent of the used technologies and implementation choices. For instance, if SOAP should be used instead of CORBA, the pattern-based architecture does not have to be changed much. In fact, this is not only a nice side-effect but an intrinsic feature of the language: the patterns aim at component-based encapsulation of subsystems and services which are glued by the Object System Layer and Message Redirector patterns. Thus, components become exchangeable with only little effort.

Other general principles enforced by the pattern language are to apply one change at a time and to perform transformations that preserve the (interface) structure of the given system. As we have seen, this was possible to a high degree for the case study presented here, because the original system had already supported almost orthogonal library interfaces. In many less well engineered systems (e.g. with only one monolithic interface) there may be considerable efforts required for restructuring. Here, the pattern language can help to achieve a stepwise migration.

For any interface migration effort, there is a risk that the resulting system does not work as expected. In our experience, only a rigorous regression test suite that should be in place *before* the reengineering effort starts (and that should be constantly evolved) can help to ensure compliance to the expected behavior.

Even though the patterns abstract from used technologies and implementation details, the general applicability of the pattern language in a project does rely on such "environmental" details. The patterns propose runtime indirection architectures. Thus new runtime structures have to be added and calls have to be dispatched dynamically. Therefore, there might slightly more memory consumption and a weaker performance. However, both properties cannot be observed in all domains, since alternative hand-built solutions to flexibility problems might have an even more severe impact on performance and memory consumption.

Another issue is the resulting complexity of the system and the necessary implementation/design effort. In general, the indirection architectures presented are more complex than simplistic, less flexible hand-built solutions. Thus it might be too much an implementation/design effort for a smaller project to implement a variant of the pattern language from scratch. An option is to use an existing implementation, say, of an object-oriented scripting language. This option imposes the necessity for the developers to learn a new language. Such considerations have to be made per project and per domain and are thus hard to generalize.

A major purpose of the project was knowledge transfer from the domain experts to the authors and vice versa. In our experience, the patterns form a good way to transport complex design decisions in a semi-technical way. However, in general, we believe that such a knowledge transfer requires technical experts in the system's domain and in the component glueing domain.

11. RELATED WORK

In the first part of this section we sketch some approaches that are known uses of the pattern language. They have in common that they only partially implement the pattern language.

Though patterns have become a popular software engineering approach only a few case studies showing the interplay of several patterns exist. These mainly demonstrate (smaller) object-oriented software patterns and are often constructed textbook examples and not real-world (industrial) cases. We will discuss current pattern language case studies in the second part of this section.

11.1. Some Known Uses of the Patterns

In [31] the (mainly black-box) component models of current middleware standards, such as 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 be changed or adapted. Most approaches have enhancements in the direction of the pattern language presented. The IDLs can be seen as a variant of EXPLICIT EXPORT/IMPORT. Stub and skeleton are a special form of COMPONENT WRAPPERS. Several approaches, such as COM interceptors [12] or Orbix Filters [13], 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 are used as first-class objects that represent abstractions over the interaction of objects. They encapsulate and enforce invariant behavior in object communications, can achieve the reduction of complexity of object interactions, and can achieve reusability of object interaction types. Besides being a variant of BEFORE/AFTER INTERCEPTOR, the abstract communicate types are COMPONENT WRAPPERS and implement a form of EXPLICIT EXPORT/IMPORT.

The new CORBA 3.0 standard specifies the CORBA component model that is based in part on the Java EJB component concepts, but goes beyond that by providing the component model to work with the heterogeneous language landscape supported by CORBA. The new CORBA standard includes the CORBA scripting language specification, which is a framework to integrate any scripting language with a CORBA mapping. Thus the specification also combines glueing with component approaches. It can – but does not have to – use an OBJECT SYSTEM LAYER for the glueing task.

Roles, as in [19], meta-object protocols [17], 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 a variant of BEFORE/AFTER INTERCEPTOR: it is transparent, composable and reusable, but it is not introspective, not dynamic, and a pure black-box approach. The approach does not support runtime dynamics.

The architecture description language Π [8] offers support for EXPLICIT EXPORT/IMPORT and component connections. An object can describe the expected environment in any level of detail. In a process of component configuration the imports can be mapped to exports. A system can be assembled in different configurations. The approach can adapt procedure/argument names, but cannot specify behavior for adaptation, such as BEFORE/AFTER INTERCEPTORS. Configurations cannot be changed at runtime.

11.2. Related Work On Pattern Case Studies

In [7, chapter 2] we can find a case study of simple document editor, built with the patterns from the same book. The case demonstrates pretty well the usage of the used patterns. But since the patterns in [7] do not form a pattern language, the architecture generation aspects of pattern languages are only implicitly touched by this early case study. We believe this is due to the absence of patterns of larger granularity that contain/integrate the smaller patterns. In [6] subjects mostly identical are covered and the design of ET++ is discussed.

There is a case study of the JAWS web server in [29, chapter 1], built with the patterns from the same book. These patterns form a pattern language for concurrent and networked objects applications. We can see a similar effect of strong pattern dependencies as in our work: the pattern usages generate open issues, that are resolved by other patterns of the pattern language. E.g., the HALF-SYNC/HALF-ASYNC pattern requires a request queue. But naive request queue implementations will incur race conditions. This problem is resolved by the MONITOR OBJECT pattern. Similar strong dependencies can be observed in our work, as discussed in Section 5.

In [32] we can find several smaller "stories" of pattern usage in various products, including Java AWT, Swing, HotDraw, VisualWorks, etc. Here, mainly patterns from [7] are discussed.

In [16] a case studies of several patterns to get around platform independence problems with Java AWT is presented. These patterns are used in several applications of the company Industrial Logic. The MICROKERNEL pattern [3] is used as a more course-grained solution, that integrates the smaller patterns. That is, it makes applications extensible with COMMANDS that are used on INTERPRETERS and COMPOSITES. Here, we can observe a small pattern language to handle platform independence problems of user interfaces, such as adding new behavior, bandwidth issue, integration, etc.

In [4] a case study of a reengineering project of a parallel program generation environment from the C language to a solution based on several patterns from [7] is presented. However, besides the patterns themselves there is no further architecturally integrating solution, as for instance a pattern language, used. In our point of view this makes it rather hard to understand the technical evolution of the system and thus it hinders communication of the results. Moreover, it seems that a piecemeal approach (in a component-oriented sense) is rather hard to accomplish by replacing small patterns with loosely-coupled pattern-based implementations in another language (here: Java). In our experience its rather unrealistic that a reengineering effort of a large-scale application can be performed in one big step. The code example presented seems to be rather complicated compared to the original solution. In contrast, our work explicitly aims at simplification of code and architecture during reengineering.

12. 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. In a piecemeal process the existing C implementation is split into black-box components. These are customizable through an explicit representation in the object system with a Component Wrapper. Before/After Interceptors let us introduce customizations decomposed and transparent to the component and its clients.

An Object System Layer with an explicit Message Redirector 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.

Often the pattern language generates pattern usages. E.g., using an Object System Layer generates the requirement to integrate foreign language/paradigm components. This task is solved by Component Wrapper. But Component Wrappers have to be customized. Such tasks can be fulfilled by Adapters [7], Decorators [7], or Before/After Interceptors. Before/After Interceptors are especially valuable, if they are automated and transparent. The Object System Layer requires a method dispatch mechanism. Both tasks are fulfilled by a Message Redirector. Explicit Export/Import can be used to wrap a system implemented with an Object System Layer, so that it appears as a base language component. Usually the export is a Facade [7] to the component, while the imports are Component Wrappers. There are several other popular object-oriented software patterns from [7, 3] that are induced by the pattern language (see [10, 11]).

All changes induced by the patterns can be applied in a piecemeal process, and existing implementations can be reused. The patterns are well known computational structures, and there are many successful approaches which are known uses of the patterns (as discussed in the related work section). Instead of directly presenting a detailed technical solution, the presentation with a pattern language should foster trust in the solution presented among the stakeholders, since no “unknown” technologies are imposed during presentation. The technologies and languages actually used can still be changed after initial discussion, without violating the architectural solution.

The patterns are applied in the concrete context of reengineering a large-scale C application in a piecemeal way. The pattern language eases the initial communication of the results to the stakeholders, which were the management and technological staff of the company. In contrast to a more concrete design (which was provided in a combination of C, C++, and XOTcl [24] to the company), the pattern language enables communication across different stakeholders. The patterns are abstract, but not vague in the technical realm. E.g., they leave technological choices open, such as which communication technology is to be used, but do nevertheless enable a suitable technical discussion. However, the concrete, detailed design is more suitable to demonstrate the actual technical realization.

Of course, the architectures generated by the pattern language come not for free. If we do not use an existing Object System Layer implementation, such as XOTcl [24], it has to be programmed by hand. This adds more complexity and higher maintenance efforts to the system. Performance can be decreased through the additional indirections in the patterns and through callback methods, but – as we show, for instance, in [23] – the techniques presented can also be implemented in a very efficient manner. The Object System Layers conventions and interfaces have to learned by the developers. If the Object System Layers is a whole scripting language, a new language has to be learned.

The presented pattern language is language-supported in the scripting language XOTcl. In this article, we have presented a case study for the pattern language in language-neutral way. That is, the presented case study could also be implemented in any other language that integrates well with C. For many industrial case studies language-neutral presentation is an important asset, because the company wants to have the last choice on implementation details in the actual product.

ACKNOWLEDGEMENTS

Part of the work has been supported by the ESAPS/KAFFE (BMBF) projects. Some of the patterns have been workshopped at EuroPLoP 2K and EuroPLoP 2001 conferences: we wish to thank the writers workshop attendees and in particular our shepherds: Ralph Johnson, Gustavo Rossi, and

Andreas Rueping. As a co-author of some of the pattern, we especially wish to acknowledge Gustaf Neumann's contribution.

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. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orinented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.
4. W. Chu, C. Lu, C. Shiu, and X. He. Pattern based software re-engineering: A case study. *Journal of Software Maintenance: Research and Practice*, 12, 2000.
5. J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1998.
6. E. Gamma. *Objektorientierte Software-Entwicklung Am Beispiel Von Et++ : Design-Muster, Klassenbibliothek, Werkzeuge*. Springer, 1992.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. 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.
9. 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.
10. M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.
11. M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In *Proceeding of EuroPlop 2001*, Irsee, Germany, July 2001.
12. G. C. Hunt and M. L. Scott. Intercepting and insturmenting COM applications. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA, May 1999.
13. IONA Technologies Ltd. The orbix architecture, August 1993.
14. 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.
15. W. Keller. Object/relational access layers – a roadmap, missing links and more patterns. In *Proceeding of EuroPlop 1998*, Irsee, Germany, July 1998.
16. J. Kerievsky. Techniques & patterns for writing once and running anywhere. http://industriallogic.com/papers/wora.html, 1998.
17. G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
18. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP97*, Finnland, June 1997. LCNS 1241, Springer-Verlag.

19. B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2:143–160, 1996.
20. R. Martin. Design principles and patterns. http://www.objectmentor.com/publications/Principles%20-and%20Patterns.PDF, 2000.
21. G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, December 1999.
22. G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA, May 1999.
23. G. Neumann and U. Zdun. High-level design and architecture of an http-based infrastructure for web applications. *World Wide Web Journal*, 3(1), 2000.
24. 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.
25. J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31, March 1998.
26. W. Pree. *Design Patterns for Object-Oriented Software Development.* ACM Press Books. Addison-Wesley, 1995.
27. D. Roberts and R. Johnson. Evolving frameworks: A pattern-language for developing object-oriented frameworks. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3.* Addison-Wesley, 1998.
28. D. C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report, SIGS*, 11(2), February 1999.
29. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects.* Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.
30. D. Spinellis. *Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming.* PhD thesis, Imperial College of Science, Technology and Medicine, London, UK, February 1994.
31. C. Szyperski. *Component Software – Beyond Object-Oriented Programming.* ACM Press Books. Addison-Wesley, 1997.
32. VA. Pattern stories. http://wiki.cs.uiuc.edu/PatternStories/PatternStories, 2001.

AUTHORS' BIOGRAPHIES

Michael Goedicke is professor of practical computer science with special emphasis on specification of software systems. He established two research sections within the Institute of Computer Science at the University of Essen. He also served on various program committees in Software Engineering conferences and recently (co-)organized the program of the Automated Software Engineering Conference 2001 in San Diego. He received his Diploma in Computer Science and his PhD from the University of Dortmund in 1980 and 1985 respectively. His current research interests include systematic design of software development methods, and component concepts for constructing software systems especially in the area of distributed and interactive systems.

Uwe Zdun is a research assistant in the Institute of Computer Science at the University of Essen. His research interests include software patterns, scripting, object-orientation, software architecture, and web engineering. He is (co-)author of the object-oriented scripting language Extended Object Tcl (XOTcl) and the active web object & mobile code system ActiWeb.