Applying Patterns for Reengineering to the Web

Author: *Uwe Zdun*

Affiliation:
New Media Lab, Department of Information Systems, Vienna University of Economics, Austria

Address:
Vienna University of Economics
Department of Information Systems
Uwe Zdun
Augasse 2-6
1090 Wien
Austria

Phone: ++43 1 313 36 4796

Fax: ++43 1 313 36 746

Email: zdun@acm.org

Applying Patterns for Reengineering to the Web

ABSTRACT

Today reengineering existing (legacy) systems to the web is a typical software maintenance task. In such projects developers integrate a web representation with the legacy system's API and its responses. Often the same information is provided to other channels than HTTP and in other formats than HTML as well, and the old (legacy) interfaces are still supported. Add-on services such as security or logging are required. Performance and scalability of the web application might be crucial. To resolve these issues, many different concepts and frameworks have to be well understood, especially legacy system wrapping, connection handling, remoting, service abstraction, adaptation techniques, dynamic content generation, and others. In this chapter, we present patterns from different sources that resolve these issues. We integrate them to a pattern language operating in the context of reengineering to the web, and present pattern variants and examples in this context.

Keywords: IS Maintenance, Legacy Migration, Reengineering, Software Patterns, Software Architecture, Object Oriented Design.

INTRODUCTION

Many existing software systems have to be migrated to the web. That means, the legacy system gets an interactive web interface, typically in addition to its existing interfaces. This web interface forwards web requests to the legacy system, decorates the responses of the system with HTML markup, and sends the results to the web client. Inputs are handled via the web browser, say, using HTML links and forms. For small applications building a web interface is quite simple; however, for larger, existing systems there are some typical issues, including:

- high flexibility and adaptability of the web interface,
- reuse of the presentation logic,
- high performance,
- preserving states and sessions,

- user management, and
- serving multiple channels.

That means, implementing a web interface for a larger, existing legacy system is a non-trivial task that is likely to be underestimated. This underestimation, in turn, results in unexpected maintenance costs and development times. However, there are many successful projects that have avoided the common pitfalls.

Software patterns are a way to convey knowledge from successful solutions. To tackle the issues named above, we present patterns from various pattern languages and pattern catalogs that can be applied in the context of reengineering to the web. In particular, we discuss connection and invocation handling, legacy system wrapping and adaptation, dealing with asynchronous invocations, concurrency handling, service and channel abstraction, session management, and dynamic content generation and conversion. The patterns are originally presented in different contexts; in this chapter we present them in variants specific for reengineering to the web. This way we build up a pattern language for reengineering to the web, consisting of patterns already documented elsewhere in related, but yet different, contexts.

The remainder of this chapter is structured as follows. In the next section we give a brief problem outline, and then we briefly explain the concepts 'pattern' and 'pattern language.' Next, we discuss connection and invocation handling in web applications that connect a legacy system to the web. Then we introduce solutions for legacy system wrapping, adaptation, and decoration. In the following section we discuss how to provide the services of a legacy system to more than one channel. Also we illustrate how to provide sessions and state in web applications despite the HTTP protocol being stateless. Next we discuss content generation, representation, and conversion for the web. The following section deals with the integration of typical add-on services, such as security or logging. Finally, we give an overview and discussion of the pattern language that we have explained incrementally in the preceding sections, provide a case study, and conclude.

PROBLEM OUTLINE

In the context of reengineering to the web, a legacy application gets an (additional) interactive web interface that decorates the outputs of the system with HTML markup and translates the (e.g. form-based) inputs via the web browser into the (legacy) system's APIs. At first glance, this

seems to be a conceptually straightforward effort, though it might turn out to be a lot work for larger systems.

In Figure 1 we can see a simplistic three-tier architecture for interactive, web-based applications. A web user agent, such as a browser, communicates with a web server. The web server "understands" that certain requests have to be handled interactively. Thus, it forwards the request and all its information to another module, thread, or process that translates the HTTP request to the legacy system's APIs. An HTML decorator builds the appropriate HTML page out of the system's response and trigger the web server to send the page to the web client.

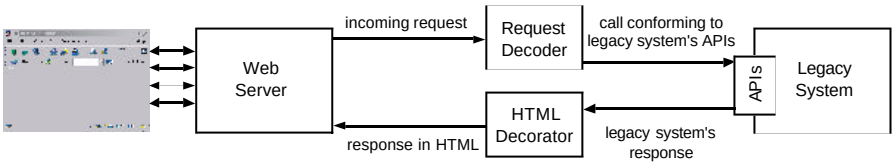

Figure 1: Simplistic Three-Tier Architecture for (Re-)engineering to the Web

Using this simple architecture for a given legacy system, we can use a simple process model for migrating to the web. In particular the following steps have to be performed in general. Of course there are feedback loops during these steps, and they are not in any particular order:

- *Providing an Interface API to the Web:* To invoke a system's functions/methods and decorate them with HTML markup, we first have to identify the relevant components and provide them with distinct interfaces (which may, in the ideal case, exist already). These interface have to satisfy the web engineering project's requirements. To find such interfaces, in general, for each component there are two possibilities:
    - *Wrapper:* The developer can wrap the component with a shallow wrapper object that just translates the URL into a call to the legacy system's API. The wrapper forwards the response to an HTML decorator or returns HTML fragments.
    - *Reengineered or Redeveloped Solution:* Sometimes it is suitable to reengineer or redevelop a given component completely, so that it becomes a purely web-

enabled solution. For instance, this makes sense, when the legacy user interface does not need to be supported anymore.

- *Implementing a Request Decoder:* A component has to map the contents of the URL (and other HTTP request information) to the legacy system's API or the wrapper's interface respectively.
- *Implementing an HTML Decorator Component:* A component has to decorate the legacy system's or wrapper's response with HTML markup.
- *Integrating with a Web Server:* The components have to be integrated with a web server solution. For instance, these components can be CGI-programs, run in different threads or processes, may be part of a custom web server, etc.

In our experience, this architecture is in principal applying for most reengineering to the web efforts and for many newly developed web-enabled systems (that have to serve other channels than HTTP as well). However, this architecture and process model do not depict the major strategic decisions and the design/implementation efforts involved in a large-scale web development project well. There are many issues, critical for success, that are not tackled by this simplistic architecture, including: technology choices, conceptual choices, representation flexibility and reuse, performance, concurrency handling, dealing with asynchronous invocations, preserving states and sessions, user management, and service abstraction.

In this chapter, we will survey and categorize critical technical aspects for reengineering to the web using the patterns of successful solutions. Thus we will step by step enrich the simple architecture presented above to gather a conceptual understanding which elements are required for reengineering a larger system to the web. Moreover, this way we will build up a framework to categorize web development environments and packages. An important goal is to assemble a feature list and terminology for mapping requirements to concrete technological and conceptual choices so that new projects can easier decide upon used frameworks.

PATTERNS AND PATTERN LANGUAGES

In this chapter, we present a pattern language for reengineering to the web. In short, a *pattern* is a proved solution to a problem in a context, resolving a set of forces. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution [Alexander,

1979]. In more detail, the context refers to a recurring set of situations in which the pattern applies. The problem refers to a set of goals and constraints that typically occur in this context, called the forces of the pattern. The forces are a set of factors that influence the particular solution to the pattern's problem. A *pattern language* is a collection of patterns that solve the prevalent problems in a particular domain and context, and, as a language of patterns, it specifically focuses on the pattern relationships in this domain and context. As an element of language, a pattern is an instruction, which can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant [Alexander, 1979].

Our pattern language uses patterns that have been documented in the literature before, either as "isolated" patterns or in other pattern languages. In the pattern language presented, we specifically use these patterns in the context of reengineering to the web. That means, we describe the patterns in this particular context. Note that there is a difference to the general purpose use of these patterns.

We present the patterns within the explanations of the reengineering solutions, where the patterns occur the first time. Each pattern is presented with a problem section, followed by a solution which starts with the word "therefore." We always cite the original pattern descriptions; refer to them for more details on the particular patterns.

CONNECTION AND INVOCATION HANDLING

There are many different kinds of frameworks that can be used to access objects running in a remote server, such as distributed object frameworks (like CORBA, DCOM, RMI), web application frameworks, or object-oriented messaging protocols. In this chapter, we commonly refer to these kinds of frameworks as *remoting* frameworks.

Web application frameworks, such as Java Servlets and JSP, Apache and Apache Modules [Thau, 1996], AOL Server [Davidson, 2000], TclHttpd [Welch, 2000], WebShell [Vckovski, 2001], Zope [Latteier, 1999], or ActiWeb [Neumann and Zdun, 2001], provide a pure server-side remoting framework based on web servers. That means, invocations sent with web requests are interactively handled by code running in the web server. Often for reengineering to the web an existing web application framework can be reused. In some cases it makes sense to implement a custom invocation framework on top of a web server, for instance, if the performance of existing

web application frameworks causes problems or if it is not possible to access or control required parameters in the HTTP header fields.

On client side, in most cases, the web browser is used as the primary client. If other programs are used as clients (or if a web browser should be implemented), an HTTP client suite has to be chosen (or built) as well. A fundamental difference of web application frameworks in comparison to other kinds of remoting frameworks, such as distributed object frameworks, is that the client (e.g. the web browser) is a generic program handling all possible applications. Distributed object frameworks use an application-specific client implementation that is realized with the client components of the distributed object frameworks.

Many web applications wrapping a legacy system have to support other remoting frameworks as well. For instance, distributed object frameworks that are also based on HTTP (such as web services) can be supported, or frameworks that are based on other protocols (such as CORBA). Combining different protocols and remoting frameworks can be handled by a SERVICE ABSTRACTION LAYER [Vogel, 2001].

The named remoting frameworks use a similar basic remoting architecture, but yet there are many differences. The developer, who wants to use one or more of these remoting frameworks, has to understand the basic remoting architecture of the used frameworks well. Of course, in the web reengineering context this especially means to understand the web application framework used. This is important because it may be necessary to tune or configure the remoting frameworks, for instance, to optimize Quality of Service (QoS), tune the performance, use threading, use POOLING [Kircher and Jain, 2002], or extend the invocation process. Also for integrating different frameworks it is important to understand their basic architecture well.

Layers of a Web Application Framework

In this section, we explain the remoting architecture of web application framework using different LAYERS [Buschmann et al., 1996] (see Figure 2):

Pattern 1 – LAYERS:

> Consider a system in which high-level functionalities are depending on low-level functionalities. Such systems often require some horizontal structuring that is orthogonal to

their vertical subdivision. Local changeability, exchangeability, and reuse of system parts should be supported.

Therefore, structure the system into LAYERS and place them on top of each other. Within each LAYER all constituent components work on the same level of abstraction.

In a web application framework, the lowest layer is an adaptation layer to the operating system and network communication APIs. The advantage of using an adaptation layer is that higher layers can abstract from platform details and therefore use an platform-independent interface. The operating system APIs are (often) written in the procedural C language; thus, if the communication framework is written in an object-oriented language, WRAPPER FACADES [Schmidt et al., 2000] are used for encapsulating the operating system's APIs.

Pattern 2 – WRAPPER FACADE:

Consider an object-oriented system that should use non-object-oriented APIs (such as operating system APIs, system libraries, or legacy systems). It should be avoided to program non-object-oriented APIs directly.

Therefore, for each set of related functions and data in a non-object-oriented API, create one or more WRAPPER FACADE classes that encapsulate these functions and data with a more concise, robust, portable, and maintainable interface.

The higher layers only access the operating system's APIs via the WRAPPER FACADES. Each WRAPPER FACADE consists of one or more classes that contain forwarder operations. These forwarder operations encapsulate the C-based functions and data within an object-oriented interface. Typical WRAPPER FACADES within the operating system adaptation layer provide access to threading, socket communication, I/O events, dynamic linking, etc.

On top of the adaptation layer the communication framework is defined. It handles the basics of connection handling and asynchronous invocation. The layer containing the HTTP client (e.g. the web browser) and the HTTP server components, including the web application framework, are defined on top of the communication framework layer.

The server application, finally, contains the wrappers for the legacy system. If the legacy system provides a procedural API, the pattern WRAPPER FACADE can be used to access it. Or the legacy system might run in a different process or different machine (e.g. as a server process or a

batch process). Then some kind of connection is required, such as inter-process communication or another communication framework.

Note that even though only the highest LAYER of the web application framework is concerned with the reengineering to the web task, the wrappers in it may have to access details in lower LAYERS. In general, low-level functionalities are hidden by higher level LAYERS. But sometimes it is necessary to deal with the low-level functionalities directly, say, for configuration, performance tuning, or Quality of Service tuning. Thus the developer of a reengineering to the web solution has to understand the lower LAYERS (APIs), even though it is usually possible to reuse existing lower LAYER implementations.

Web Connection vs. Connection to the Legacy System

As illustrated in Figure 2 there are possibly two connections between different systems. Besides the web connection between web client and web server, there may be a second connection from the web application framework to the legacy system. We might have to cope with characteristics of the web connection when connecting to the legacy system as well. Typical problem fields are asynchronous invocations, concurrency, the statelessness of the HTTP protocol, missing support for sessions, and potential performance overheads. If a particular problem is not already resolved by the web application framework, it has to be resolved within the connection to the legacy system. Out of these reasons, even if an existing web application framework can be reused, a reengineer has to understand the solution to these problems in the web application framework well. The connection to the legacy system has to use the invocation model of the web application framework and/or customize it to its requirements.

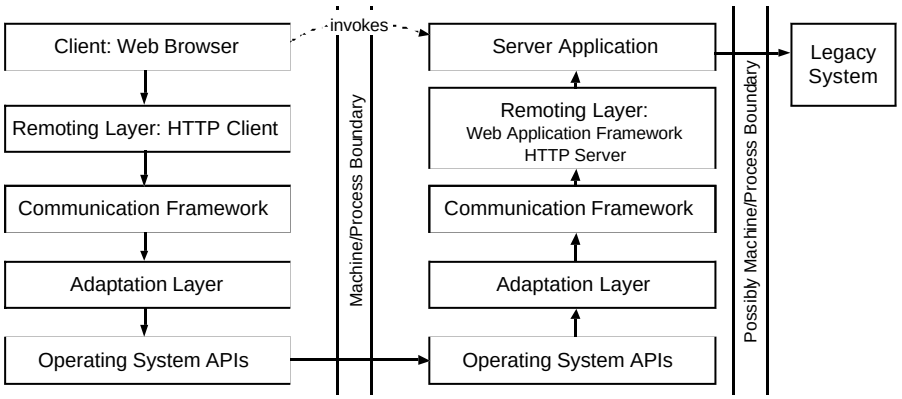

Figure 2: Web Application Layers

As a typical example, consider a legacy system that allows for synchronous processing (e.g. batch processing) only. Web clients send asynchronous and concurrent requests. Thus somewhere these requests have to be demultiplexed and synchronized. Potentially the web application framework can demultiplex and synchronize the requests already. But often for performance reasons such requests are handled by multiple threads in the web server. These threads again can concurrently access the legacy system. Then the connection to the legacy system has to handle demultiplexing and synchronization for the access to the legacy system.

That means, the patterns of connection handling and invocation handling discussed in the next sections either can be implemented within the web application framework or as part of the connection to the legacy system.

Communication Framework: Connection Handling and Concurrency

For remote communication, connection establishment is typically handled by the ACCEPTOR/CONNECTOR pattern [Schmidt et al., 2000]:

Pattern 3 – ACCEPTOR/CONNECTOR:

> Protocols like HTTP are based on connections (e.g. socket connections). Coupling of connection management and configuration code with the processing code should be avoided.
>
> *Therefore,* separate the connection initialization from the use of established connections. A connection is created, when the connector on client side connects to the acceptor on server side. Once a connection is established, further communication is done using a connection handle.

An HTTP server listens to a network port to accept connections sent by the client; in turn, the client (i.e. the web browser) uses a connector to start an HTTP request. A connector and acceptor pair is also used for response handling. The server response connects to the client that awaits the server's responses with a response acceptor.

An HTTP server has to support asynchronous invocations to enable multiple simultaneous requests. Non-trivial clients (like most web browsers) should not block during request and response, and thus, are also asynchronous. Therefore, the programming model of most web applications is event-based both on client side and server side, meaning that an event loop and/or

multi-threading is supported and connection handling is done using callbacks for the network events. A REACTOR [Schmidt et al., 2000] reacts on the network events and efficiently demultiplexes the events. It dispatches all requests to handler objects:

Pattern 4 – REACTOR:

Event-based, distributed systems (may) have to handle multiple requests (and responses) simultaneously.

Therefore, synchronously wait for (connection) events in the REACTOR. The system registers event handlers for certain types of events with the REACTOR. When an event occurs, the REACTOR dispatches the event handler associated with the event.

In web applications, a REACTOR is typically used on client side and server side.

Server side concurrency can simply be handled by the REACTOR and an event loop. That means the REACTOR queues up all network events in the order of demultiplexing. This is a very simple concurrency handling architecture that yields a good performance for relatively small, remote operations. Computation-intensive or blocking operations may cause problems, as queued requests have to wait for these operations.

If computation-intensive or blocking operations can be expected (what may be the case for a legacy system), the web server should proceed with other work while waiting for the legacy system's response. Thus, in such cases, typically a multi-threaded server is used. Then only one thread of control has to wait for a computation-intensive or blocking operation, while other threads can resume with their work.

If we access a legacy system that requires synchronous access from multiple threads, we have to synchronize and schedule the accesses. This can done by using objects in the web application server for scheduling access to the legacy system; there are two alternative patterns for synchronizing and scheduling concurrently invoked remote operations [Schmidt et al., 2000]:

Pattern 5 – ACTIVE OBJECT:

Consider that multiple clients access the same object concurrently. Processing-intensive operations should not block the entire process. Synchronized access to shared concurrent objects should be provided in a straightforward manner.

Therefore, decouple the executing from the invoking thread. A queue is used between those

threads to exchange requests and responses.

Pattern 6 – MONITOR OBJECT:

Consider that multiple client requests access the same thread of control concurrently. An object should be protected from uncontrolled concurrent changes and deadlocks.

Therefore, let a MONITOR OBJECT ensure that only one operation runs within an object at a time by queuing operation executions. It applies one lock per object to synchronize access to all operations.

Often threading is used for connection handling and execution. For each particular connection, a connection handle has to be instantiated. To optimize resource allocation for connections, the connections can be shared in a pool using the pattern POOLING [Kircher and Jain, 2002]. This reduces the overhead of instantiating the connection handle objects.

Pattern 7 – POOLING:

Consider a system that provides access to multiple resources of the same kind, such as network connections, objects, threads, or memory. Fast and predictable access to these resources and scalability are required.

Therefore, manage multiple instances of a resource type in a pool. Clients can acquire the resources from the pool, and release them back into the pool, when they are no longer needed. To increase efficiency, the pool eagerly acquires a static number of resources after creation. If the demand exceeds the available resources in the pool, it lazily acquires more resources.

When the legacy system can be accessed concurrently, we can also pool the instances that connect to the legacy system to avoid the overhead of establishing a connection to the legacy system.

In the context of reengineering to the web, it is especially important to select for a proper concurrency model. Also, connection and/or thread POOLING have to be adjusted to the requirements of the legacy application. Usually, these issues do not have to be implemented from scratch, but only the models of used remoting frameworks have to be well understood and integrated with the model of the legacy application. In contrast, the connection to the legacy system is most often hand-built.

Remoting Layer

On top of the connection handling and concurrency layer (the communication framework) there are a few basic remoting patterns (from [Voelter et al., 2002]) that are used for building the remoting layer.

In a remoting framework clients communicate with objects on a remote server. On the server side the invoked functionality is implemented as a REMOTE OBJECT [Voelter et al., 2002]:

Pattern 8 – REMOTE OBJECT:

> Accessing an object over a network is different from accessing a local object because of machine boundaries, network latency, network unreliability, and other properties of network environments.
>
> *Therefore,* let a distributed object framework transfer local invocations from the client side to a REMOTE OBJECT within the server. Each REMOTE OBJECT provides a well-defined interface to be accessed remotely.

In web application frameworks, the REMOTE OBJECT is responsible for building up the delivered web pages dynamically. In the legacy reengineering context that means it has to invoke the legacy system and decorate it with HTML. That means, if a connection to the legacy system has to be established, the REMOTE OBJECT is responsible for building it up. As explained in the previous section the concurrency patterns and POOLING might also be applied within the REMOTE OBJECT.

The client invokes an operation of a local object and expects it to be executed by the REMOTE OBJECT. To make this happen, the invocation crosses the machine boundary, the correct operation of the correct REMOTE OBJECT is obtained and executed, and the result of this operation invocation is passed back across the network. In a distributed object framework a CLIENT PROXY [Voelter et al., 2002] is used by a client to access the REMOTE OBJECT:

Pattern 9 – CLIENT PROXY:

> The programming model on client side should not expose the networking details to clients.
>
> *Therefore,* provide a CLIENT PROXY as a local object within the client process that offers the REMOTE OBJECT'S interface and hides networking details.

In web applications the CLIENT PROXY usually is a generic invocation handler that sends network requests for web pages using an URL. In the URL or in the HTTP request, the details of the invocation (object ID, operation name, parameters) are encoded. These details are used by the server to find the correct object to be invoked. These tasks are performed by an INVOKER [Voelter et al., 2002] on the server side:

Pattern 10 – INVOKER:

When a CLIENT PROXY sends invocations to the server side, somehow the targeted REMOTE OBJECT has to be reached.

Therefore, provide an INVOKER that is remotely accessed by the CLIENT PROXY with the invocation data. The INVOKER de-marshals the invocation data and uses it to dispatch the correct operation of the target REMOTE OBJECT.

Note that in distributed object frameworks that are primarily designed for program-to-program communication, such as CORBA or web services, often CLIENT PROXIES and INVOKERS are specific for a REMOTE OBJECT type. They can be generated from an INTERFACE DESCRIPTION [Voelter et al., 2002]. In contrast, in a web application the interfaces are solely defined by the server and the correct access interfaces (and parameters) have to be delivered to the client using web pages. For integrating web application with distributed object frameworks it is necessary to integrate these differences of the remoting models. When the legacy system contains full INTERFACE DESCRIPTIONS, such as C header files for instance, these can potentially be used to generate the INVOKERS automatically.

Usually, there is more than one INVOKER in a web application framework; for instance, one for delivering static web pages from the file system and one for handling remote invocations of REMOTE OBJECTS. Thus some instance is required to pick out the responsible INVOKER for an incoming request. On client and server side, a REQUEST HANDLER [Voelter et al., 2002] is used to abstract the communication framework for CLIENT PROXY and INVOKER:

Pattern 11 – REQUEST HANDLER:

On client and server side the details of connection handling have to be handled. In particular, communication setup, resource management, threading, and concurrency have to be handled and configured in a central fashion.

Therefore, provide a SERVER REQUEST HANDLER, responsible for dealing with the network communication on server side, together with a respective CLIENT REQUEST HANDLER, responsible for client side communication. These REQUEST HANDLERS interact with CLIENT PROXIES and INVOKERS respectively.

Usually, the same communication framework implementations can be used on client and server side, but the REQUEST HANDLERS are different: the CLIENT REQUEST HANDLER connects to the server side and uses the REACTOR to wait for the reply, whereas the SERVER REQUEST HANDLER waits for incoming invocations and sends the reply back, after it is processed. CLIENT PROXY, REQUEST HANDLER, and INVOKER build together a BROKER [Buschmann et al., 1996] (see Figure 3).

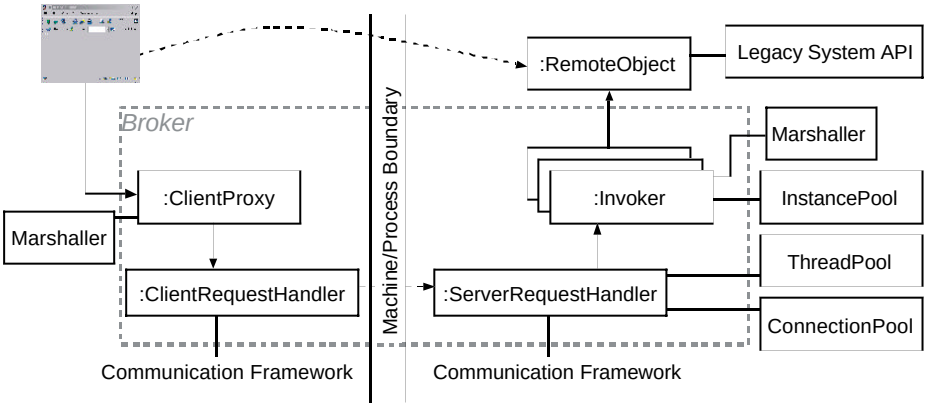

Figure 3: Web Remoting Layer

Marshalling of request and response has to happen on client and server side according to the HTTP protocol. This especially means adding of HTTP headers, encoding URLs and form data, and converting binary data (e.g. using Base64 encoding). This is done by a MARSHALLER [Voelter et al., 2002]:

Pattern 12 – MARSHALLER:

The data to describe operation invocations of REMOTE OBJECTS consist of the target object's object ID, the operation identifier, the arguments, and possibly other context information. All this information has to be transported over the network connection.

Therefore, require each REMOTE OBJECT type to provide a way to serialize itself into a transport format that can be transported over a network as a byte stream. The distributed

object framework provides a MARSHALLER (on client and server side) that uses this mechanism whenever a remote invocation needs to be transported across the network.

A MARSHALLER for web applications also has to convert remote invocations into parts of web pages. Usually, remote invocations are either modeled as HTML links or actions of HTML forms. In contrast to distributed object frameworks, in a web application marshalling of invocations is only performed on server side: the client side invocations are encoded into the delivered web pages. For instance, the following invocation to the REMOTE OBJECT `objectName` and the method `methodName` can be embedded in an HTML document as a link:

```
http://www.myhost.org/objectName+methodName?arg=xyz
```

On server side such invocations have to be de-marshalled, when a request arrives (for instance, the URL string has to be split and/or the HTML form data has to be evaluated). Marshalling can triggered by the INVOKER. If more than one INVOKER is used in the system, the REQUEST HANDLER can de-marshal the invocation partially to find the correct INVOKER to handle the invocation.

Note that the connection to the legacy system might need more context information, such as for instance a session identifier. Such context information also need to be marshalled and embedded in HTML or HTTP (that is either in the URLs of HTML links, in the HTML text with HIDDEN form fields, or as cookies).

Sensitive information, such as user authentications, additionally can be encrypted during marshalling.

LEGACY SYSTEM WRAPPING, ADAPTATION, AND DECORATION

An important aspect of migrating a legacy system to the web is *wrapping*. Wrappers are mechanisms for introducing new behavior to be executed before, after, in, and/or around an existing method or component. Wrapping is especially used as a technique for encapsulating legacy components. Common techniques of reusing legacy components by wrapping are discussed in [Sneed, 2000, Brant et al., 1998].

On server-side of a web application framework, we have to map incoming invocations into the legacy system. The invoker provides the invocation data that it has obtained by de-marshalling the URL and the HTTP request header. The invocation data references a wrapper object. This wrapper is responsible for forwarding the invocation into the legacy system and obtaining the result.

The COMPONENT WRAPPER pattern [Zdun, 2002a] generalizes this form of (legacy) wrapping:

Pattern 13 – COMPONENT WRAPPER:

A component should be used with its external interface only, but still we often require some way to customize and adapt a component to a degree that goes beyond pure parameterization.

Therefore, let the access point to a component be a COMPONENT WRAPPER, a first-class object of the programming language. Within this object the component access can be customized without interfering with the client or the component implementation.

The COMPONENT WRAPPER is a white-box for the component's client and enables us to bring in changes and customizations, e.g. with DECORATORS [Gamma et al., 1994] and ADAPTERS [Gamma et al., 1994]. Note that the legacy system can run in the same process as the COMPONENT WRAPPER, or it can run in a different process. When reengineering to the web, a typical COMPONENT WRAPPER is also a REMOTE OBJECT running in the web application framework. It exports (some of its) operations as URLs to the web.

The COMPONENT WRAPPER methods can be simple forwarder operations. When a procedural API is supported by the legacy system, they are usually implemented as WRAPPER FACADES. Or, alternatively, the COMPONENT WRAPPER might build up a connection to the legacy system either with inter-process communication (e.g. for batch processing applications) or with a communication protocol (e.g. for server applications).

As an indirection mechanism, wrappers can also be used as a place for applying adaptations and decorations [Gamma et al., 1994]:

Pattern 14 – ADAPTER:

A client requires a different interface than provided by a class, yet the class cannot be changed.

Therefore, use an ADAPTER to convert the interface of a class into another interface clients expect. It can be realized as a class adapter using multiple inheritance or as an object adapter which forwards messages to the adapted object.

Pattern 15 – DECORATOR:

An object should expose some additional behavior, yet the class of the object should not be changed.

Therefore, use a DECORATOR to dynamically attach additional responsibility to an object using a DECORATOR object that forwards invocations to the decorated object, after the decoration has taken place.

In the context of reengineering to the web, ADAPTERS are especially used for adaptations of interfaces, if the legacy system does not support a required interface that should be exposed to the web (or other remote clients). There are many tasks for DECORATORS in the context of reengineering to the web, including logging, access control, encryption, etc. Besides DECORATORS of the COMPONENT WRAPPER, DECORATORS can also be applied for the INVOKER, if the whole invocation process should be decorated.

Both tasks, adaptation and decoration, can also be further supported by the pattern MESSAGE INTERCEPTOR [Zdun, 2002a]:

Pattern 16 – MESSAGE INTERCEPTOR:

Consider decorations and adaptations take place for certain objects over and over again. Then it would make sense to support these tasks as a first-class mechanism, instead of programming them from scratch for each use.

Therefore, build a callback mechanism into the INVOKER working for all messages that pass it. These callbacks invoke MESSAGE INTERCEPTORS for standardized events, such as "before" a remote method invocation, "after" a remote method invocation, or "instead-of" invoking a remote method.

A MESSAGE INTERCEPTOR intercepts messages sent between different entities, and can add arbitrary behavior to the invocation and/or the response. Support for MESSAGE INTERCEPTORS can be provided on different levels. A MESSAGE INTERCEPTOR can be supported by the programming language used for building the COMPONENT WRAPPER (or a language extension). Also, interceptors can be provided in distributed systems by a middleware, as implemented in TAO or Orbix (see the pattern INTERCEPTOR [Schmidt et al., 2000] which describes this variant). The distributed variant of MESSAGE INTERCEPTORS can be implemented in a web application framework on top of the INVOKER. This is specifically suitable for tasks that have to be handled per invocation, such as user rights management, marshalling, access control, logging, etc.

In Figure 4 an INVOKER that supports MESSAGE INTERCEPTORS is depicted. In this examples, a request for a COMPONENT WRAPPER arrives. This COMPONENT WRAPPER has two "before" interceptors. These are dispatched before the actual invocation takes place. When these interceptors have returned, the COMPONENT WRAPPER is invoked and it forwards the invocation to the legacy system. The COMPONENT WRAPPER also has one "after" interceptor. It is dispatched by the INVOKER after the COMPONENT WRAPPER returns and before the result is sent back across the network. Using such a chained MESSAGE INTERCEPTOR architecture developers can flexibly register add-on services as MESSAGE INTERCEPTORS at runtime for specific COMPONENT WRAPPERS, as required.

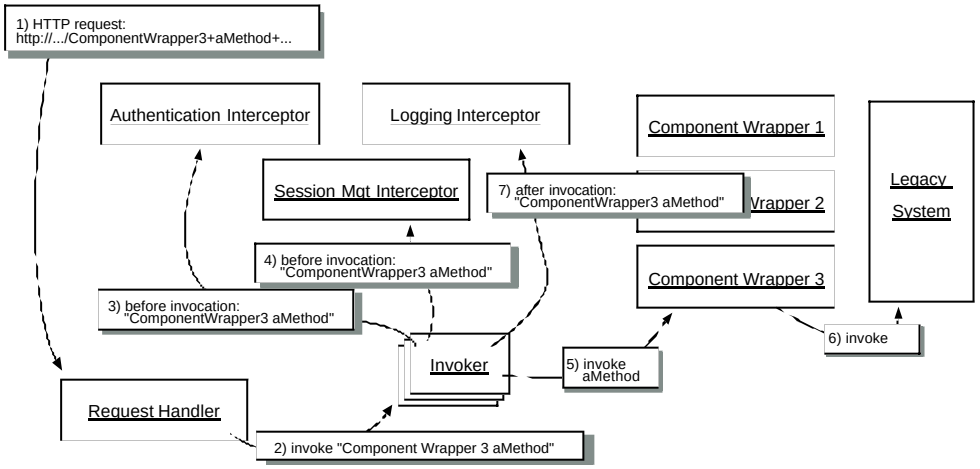

Figure 4: An invoker applying message interceptors for a component wrapper

REMOTE OBJECTS can be pooled using POOLING [Kircher and Jain, 2002]. In case of COMPONENT WRAPPERS for legacy systems that means a COMPONENT WRAPPER pool is used to access the legacy system (one pool for each used COMPONENT WRAPPER type). For each invocation a COMPONENT WRAPPER is taken from the pool, handles the invocation, is cleaned up, and put back into the pool. This way we can avoid the overhead of instantiating the COMPONENT WRAPPER and establishing a connection to the legacy system for each incoming invocation. This, of course, requires the legacy system to support multiple concurrent invocations (or synchronization).

SERVICE AND CHANNEL ABSTRACTION

Often a legacy system provides multiple services. Each COMPONENT WRAPPER type can be seen as one service of the legacy application, and all these services should be offered via the same web application interface. Also, different requests coming from different clients, communicating over different channels, have to be supported by many web application frameworks. It should not be required to change the application logic every time a new channel has to be supported or a new service is added to the application.

When reengineering large-scale systems to the web, it can be expected that the problems of multiple channels and service abstraction are occurring together. These forces are often resolved by a SERVICE ABSTRACTION LAYER architecture [Vogel, 2001]:

Pattern 17 – SERVICE ABSTRACTION LAYER:

Consider a system that serves multiple services to multiple channels. If possible, services should be implemented independently from channel handling, and implementations should be reused.

Therefore, provide a SERVICE ABSTRACTION LAYER as an extra layer to the application logic tier containing the logic to receive and delegate requests. For each supported channel there is a channel adapter. The INVOKER does not directly access the COMPONENT WRAPPER but it sends invocations to the SERVICE ABSTRACTION LAYER first, which then selects the correct COMPONENT WRAPPER.

The SERVICE ABSTRACTION LAYER also converts the invocation data to a format understood by the COMPONENT WRAPPER, and converts the response back to the client format. This is done using CONTENT CONVERTERS [Vogel and Zdun, 2002].

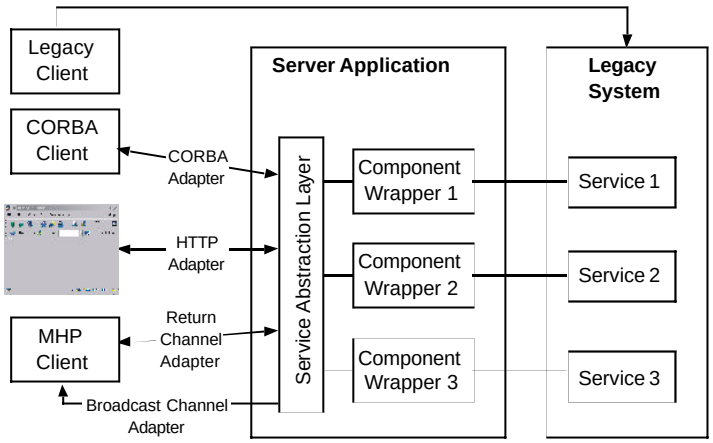

Figure 5: Service Abstraction Layer

SESSION MANAGEMENT AND STATE PRESERVATION

A client can be expected to make several subsequent requests over a time span, and often some state has to be preserved between these requests. For instance, especially in multi-user systems, users might have to log in and out. Most legacy applications are requiring states to be maintained for continued interaction with a client. The HTTP protocol is stateless and cannot be used for maintaining the state of an interaction.

The SESSIONS pattern [Sorensen, 2002] provides a solution to this problem:

Pattern 18 – SESSIONS:

A system requires a state for an interaction with a client, but the communication protocol is stateless.

Therefore, state is tied to a session so that a request can access data accumulated earlier during an interaction. Such data is referred to as session-specific data. There is a session identifier to let REMOTE OBJECTS be able to refer to a session.

In web applications, there are two variants of the SESSIONS pattern: sessions can be kept in the server or in the client. If the session is kept in the server, the session identifier is sent with the each response to the client, and the client refers to it in the next invocation; if it is kept in the client, the client has to send it to the server, which refers to it in its responses.

To map a stateless client request properly to the correct session, we have different options in web applications:

- *URL Encoding:* We can encode information, such as user name and password, in the URL by attaching them as standard URL parameters to the base URL. This works in almost any setting, but we have to be aware that some browsers have limitations regarding the length of the URL. Moreover, the information is readable on the user's screen (if URL contents are not encrypted form). For sensitive applications we can encrypt the contents of the URL.
- *HIDDEN Form Fields:* We can embed information in an HTML form that is hidden from display by using HIDDEN form fields. However, of course, they are readable as plain text in the HTML page's source. Again, we can use encryption for sensitive data.
- *Cookies:* Cookies are a way to store and retrieve information on the client side of a connection by adding a simple, persistent, client-side state settable by the server. However, the client can deactivate cookies in the user agent; thus, cookies do not always work. Cookies can optionally be sent via a secure connection (e.g. SSL).

In any case, when reengineering to the web, it is necessary to integrate these SESSIONS with the session and user model of the legacy application (if there is any). For instance, one solution is that COMPONENT WRAPPERS log the user in and out for each invocation. Or, alternatively, the COMPONENT WRAPPERS can hold the session open until the next invocation. In case of many client sessions, it may be necessary to introduce LEASES [Jain and Kircher, 2002]:

Pattern 19 – LEASES:

A (web) client can abort a stateful interaction without the server realizing it. Then existing SESSIONS are never stopped and opened connections are never closed.

Therefore, provide LEASES for each session (or connection) that expire after a certain amount of time. Let the COMPONENT WRAPPER close a session (or connection) that is held open, when its LEASE expires. When a client sends the next invocation in time, the LEASE is renewed so that the session is held open again for the full lease duration.

DYNAMIC CONTENT GENERATION AND CONVERSION

In [Vogel and Zdun, 2002] we present a pattern language for content generation, representation, and conversion for the web. Here, we use a sub-set of this pattern language in the context of reengineering systems to the web.

Dynamic content generation and conversion are usually central tasks of a project dealing with migration to the web. In particular the web requests received by the INVOKER have to be translated into the API of the legacy system, and the response of the system has to be decorated with HTML markup. What seems to be a relatively simple effort at first glance may lead to severe problems when the resulting system has to be further evolved later on. Often we find systems in which the HTML pages are simply generated by string concatenation, such as the following code excerpt:

```
StringBuffer htmlText = new StringBuffer();
String name = legacyObject.getName();
...
htmlText.append("<BR> <B> Name: </B>");
htmlText.append(name);
```

Hard-coding of HTML markup in the program code may lead to severe problems regarding extensibility and flexibility of content creation. Content, representation style, and application behavior should be changeable ad hoc. In this section, we will discuss two conceptually different approaches for decorating with HTML markup: template-based approaches and constructive approaches.

Template-Based Approaches

A CONTENT FORMAT TEMPLATE [Vogel and Zdun, 2002] provides a typical template-based approach for building up web content:

Pattern 20 – CONTENT FORMAT TEMPLATE:

A system requires content to be built up in a target content format. Content editors should be able to add dynamic content parts and invocations to legacy systems in a simple way. A high performance for building up web pages is required.

Therefore, provide a template language that is embedded into the target content format (e.g. HTML). The templates contain special template language code which is substituted by a template engine before the page is delivered to the client.

Known uses of template-based approaches are PHP [Bakken and Schmid, 2001], ASP, JSP, or ColdFusion. These let developers write HTML text with special markup. The special markup is substituted by the server, and a new page is generated. For instance, in PHP the PHP code is embedded by escaping HTML with a special comment:

```
<body>
  <h1> <?php echo("My PHP Heading\n"); ?> </h1>
</body>
```

In the same way invocations to COMPONENT WRAPPERS can be embedded in template languages to connect to a legacy system.

On the first glance, the approach is simple and even well-suited for end-users (such as content editors), say, by using special editors. The HTML design can be separated from the software development process and can be fully integrated with content management systems. However, some web-based applications require more complex interactions than simply expressible with templates. Sometimes, the same actions of the user should lead to different results in different situations. Most approaches do not offer high-level programmability in the template or conceptual integration across the template fragments. Application parts and design are not clearly separated. Thus template fragments cannot be cleanly reused. Complex templates may quickly become hard to understand and maintain.

Sometimes integration of different scripts can be handled via a shared dataspace, such as a persistent database connection. Sometimes, we require server-side components for integrating the scripts on the server side. In such cases a constructive approach can be chosen as an alternative.

Constructive Approaches

Constructive approaches generate a web page on-the-fly using a distinct API for constructing web pages. They are not necessarily well-suited for end-users as they require knowledge of a full programming language. However, they allow for implementing a more complex web application logic than easily achievable with most template-based approaches.

The most simple constructive approach is the CGI interface [Coar, 1999]. It is a standardized interface that allows web servers to call external applications with a set of parameters. The primary advantages of CGI programming are that it is simple, robust, and portable. However, one process has to be spawned per request, therefore, on some operating systems (but, for instance, not on many modern UNIX variants) it may be significantly slower than using threads. Usually different small programs are combined to one web application. Thus conceptual integrity of the architecture, rapid changeability, and understandability may be reduced significantly compared to more integrated application development approaches. Since every request is handled by a new

process and HTTP is stateless, the application cannot handle session states in the program, but has to use external resources, such as databases or central files/processes.

A variant of CGI is FastCGI [Open Market, Inc., 1996] which allows a single process to handle multiple requests. The targeted advantage is mainly performance. However, the approach is not standardized and implementations may potentially be less robust.

A similar approach integrated with the Java language are servlets. They are basically Java classes running in a Java-based web server's runtime environment. They are a rather low-level approach for constructing web content. In general, HTML content is created by programming the string-based page construction by hand. The approach offers a potentially high performance. As different servlets can run in one server application (servlet container), servlets provide a more integrated architecture than CGI, for instance.

Most web servers offer an extension architecture. Modules are running in the server's runtime environment. Thus a high performance can be reached and the server's feature (e.g. for scalability) can be fully supported. Examples are Apache Modules [Thau, 1996], Netscape NSAPI, and Microsoft ISAPI. However, the approach is usually a low-level approach of coding web pages in C, C++, or Java. Moreover, most APIs are quite complex, and applications tend to be monolithic and hard to understand.

Custom web servers, such as AOL Server [Davidson, 2000], TclHttpd [Welch, 2000], WebShell [Vckovski, 2001], Zope [Latteier, 1999], or ActiWeb [Neumann and Zdun, 2001], provide more high-level environments on top of ordinary web-servers. Often they provide integration with high-level languages, such as scripting languages, for rapid customizability. A set of components is provided which implement the most common tasks in web-application development, such as: HTTP support, session management, content generation, database access/persistence services, legacy integration, security/authentication, debugging, and dynamic component loading. Some approaches offer modules for popular web servers as well, as for instance the Apache Module of WebShell.

In these constructive approaches HTML markup can either be built by hard-coding, perhaps together with FRAGMENTS (see next section), or using a CONTENT CREATOR [Vogel and Zdun, 2002]:

Pattern 21 – CONTENT CREATOR:

Content in different content formats has to be built up dynamically with a constructive approach. But HTML text should not be hard-coded into the program text.

Therefore, provide an abstract CONTENT CREATOR class that contains the common denominator of the used interfaces. For each supported content format there are concrete classes that implement the common denominator interface for the specific content format. The concrete classes might also contain methods for required specialties.

For instance, WebShell [Vckovski, 2001] uses global procedures to define the elements of its CONTENT CREATOR:

```
proc dl {code} {
  web::put "<dl>"
  uplevel $code
  web::put "</dl>"
}
```

To code these procedures we have to interfere with HTML markup. However, we can avoid it later on when we combine such procedures to HTML FRAGMENTS:

```
dl {
  b {My first page}
  em {in Web Shell}
}
```

Here we have created a `<dl>` list entry with a bold and an emphasized text in it. Potentially, the procedure `dl` can be exchanged with another implementation for building a different content format.

In Actiweb [Neumann and Zdun, 2001] CONTENT CREATORS are used to build up pages. Thus we can use the same code to build up pages for different user interface types. A simple example just builds up a web page:

```
HtmlBuilder htmlDoc
htmlDoc startDocument \
  -title "My ActiWeb App" \
  -bgcolor FFFFFF
htmlDoc addString "My ActiWeb App"
htmlDoc endDocument
```

We instantiate an object `htmlDoc`, then start a document, add a string, and end the document. The developer does not see any HTML markup at all. The page is automatically created by the CONTENT CREATOR class.

Many approaches combine the template-based and the constructive approach. However, often the two used models are not well-integrated; that is, the developer has to manually care for the balance between static and dynamic parts.

Caching

When the web client sends a request to the web server, the requested page may be available on the file system of the server or it may be dynamically created. The template-based and constructive approaches introduced so far require dynamic page creation. Compared to static pages, dynamic creation has the problem of memory and performance overheads. A CONTENT CACHE [Vogel and Zdun, 2002] provides a solution:

Pattern 22 – CONTENT CACHE:

The overhead of dynamic page creation causes performance or memory problems.

Therefore, increase the performance of web page delivery by caching already created dynamic content.

In the context of reengineering to the web, caching has to work in concert with the legacy application. Usually, we cannot change the legacy application to invalidate cached elements that are not valid anymore. We can therefore only cache those operations of the legacy system for which the COMPONENT WRAPPER can determine whether the cache is still valid or not.

Caching whole pages is only a good idea for simplistic pages. For larger web applications, the dynamic web pages should be designed in such a way that parts of pages can be cached. The pattern FRAGMENTS [Vogel and Zdun, 2002] solves this problem:

Pattern 23 – FRAGMENTS:

Web pages should be designed to allow the generation of web pages dynamically by assuring the consistency of its content. Moreover, these dynamic web pages should be provided in a highly efficient way.

Therefore, provide an information architecture which represents web pages from smaller building blocks, called FRAGMENTS. Connect these FRAGMENTS so that updates and changes can be propagated along a FRAGMENTS chain.

Thus FRAGMENTS can be cached instead of whole pages. Only parts of web pages that have (or might have) changed since a previous creation of the FRAGMENT are dynamically created. Static FRAGMENTS of web pages and dynamic FRAGMENTS that are still valid are obtained from the CONTENT CACHE.

Web Content Management

The full pattern language in [Vogel and Zdun, 2002] contains some additional patterns not described here (GENERIC CONTENT FORMAT, CONTENT CONVERTER, and PUBLISHER AND GATHERER). These are mainly used for content gathering, storing, and publishing. These content management tasks are usually not required for reengineering to the web, as the content itself is managed by the wrapped legacy applications. Note that, in the content management context, the pattern CONTENT CONVERTER is mostly used to convert to and from a GENERIC CONTENT FORMAT, whereas in the web reengineering context it can be used for converting to/from the content formats of the legacy system.

Note that there are also applications that contain content from multiple sources, and a wrapped legacy application is only one of these sources. For integrating and managing this content, the pattern language from [Vogel and Zdun, 2002] can be used in its original web content management focus.

ADD-ON SERVICES

In a web application that wraps a legacy system many add-on services may be required per invocation. Typical examples are authorization, encryption, and logging. When a request arrives or when the response is sent, the add-on service has to be executed. That means, we can potentially apply those add-on services either within:

- the REQUEST HANDLER for request-specific services,
- the INVOKER for invocation-specific services, or
- the COMPONENT WRAPPER for services that are specific per REMOTE OBJECT or REMOTE OBJECT type.

As discussed before, the patterns DECORATOR and ADAPTER provide a simple, object-oriented solution for providing additional services. MESSAGE INTERCEPTORS can be used for providing a

first-class solution for add-on services. Usually higher-level LAYERS can configure MESSAGE INTERCEPTORS for lower levels.

Security issues are relevant for many web applications that wrap a legacy system. These are often provided as add-on services. For instance, a login with user name and password may be required. Moreover, secure communication or securing transferred data is required. These issues have to be tightly integrated with session management. In general, we require user authentications and encryption as typical means to secure an interactive web application, for instance:

- *HTTP Basic Authentication:* The definition of HTTP/1.1 [Fielding et al., 1999] contains some means for user authentication of web pages, called basic authentication scheme. This simple challenge-response authentication mechanism lets the server challenge a client request and clients can provide authentication information. The basic authentication scheme is not considered to be a secure method of user authentication, since the user name and password are passed over the network in an unencrypted form.
- *HTTP Digest Authentication:* Digest Access Authentication, defined in RFC 2617 [Franks et al., 1999], provides another challenge-response scheme, that does never send the password in unencrypted form.
- *Encrypted Connection (using SSL):* Using a secure network connection, supported by most servers, we can secure the transaction during a session.
- *URL Encryption:* To avoid readability of encoded URLs we can encrypt the attached part of the URLs.

HTTP authentication is usually handled on the REQUEST HANDLER level. The authentication information is obtained from the COMPONENT WRAPPER that connects to the legacy system. URL encryption is usually handled per invocation within the MARSHALLER.

An important aspect of most web applications is the required high availability. Usually a web site should run without any interruptions. This has several implications that have to be considered when choosing frameworks, concepts, implementations, etc. At least the following functionalities are usually provided as add-on services:

- *Permanent and Selective Logging:* All relevant actions have to be logged so that problems can be traced. Some selection criteria should be supported, otherwise it may be hard to find the required information out of the possibly large number of log entries. Sometimes, for legal reasons, even more information has to be logged, such as user transaction traces for e-commerce stores. Thus logging needs to be highly configurable. For instance, WebShell [Vckovski, 2001] supports log filters that distinguish different log levels and redirect log entries to different destinations, like files, stdout, or SMS.
- *Notification of Events:* In cases when certain events happen, such as certain error states, a person or application should be notified. For instance, when a error message is recurring, an email or SMS may be sent to the system's administrator.
- *Testing:* Load generators and extensive regression test suite are required for testing under realistic conditions.
- *Incremental Deployment:* Dynamically loadable components enable incremental deployment so that the application needs not to be stopped to deploy new functionality.

If multiple add-on services are required, as well as dynamic configuration of these, typically a chained MESSAGE INTERCEPTOR architecture is provided. This architecture can also be used for standard services, such as URL decoding, marshalling, etc.

PATTERN LANGUAGE OVERVIEW AND DISCUSSION

The pattern language presented in this chapter mainly can be used to explain different successful technical solutions in the realm of reengineering to the web. There are the following main goals:

- *Design Guideline:* The patterns provide a design guideline for reengineering to the web that can even be used in early project phases, when there is no decision for concrete technologies yet.
- *Project Estimation and Technology/Framework Selection:* The pattern language can help to estimate the effort for a reengineering to the web project in general, and make import project-specific decisions, such as selecting for a web application framework. The project team has to compare the features of particular frameworks with the project requirements. These, in turn, can be obtained by selecting the required pattern variants.

Even though this method cannot ensure to avoid all possible technical problems, it provides a good means to get a concrete estimation or basis for framework selection in early project phases. The aspects discussed in this chapter can also be used as a checklist for a project estimation.

- *Means for Communication:* Patterns provide a means for communication, for instance, between technical and non-technical stakeholders of a system that allows for accurate discussion, but without delving too deep into the technical details (as, for instance, when a discussion is based on the concrete technical design or implementation).

In Figure 6 an integrated overview of the pattern language is presented. The arrows indicate important dependencies of the patterns in the context of reengineering to the web. Only the pattern LAYERS is not depicted here because it structures the other patterns (and thus has relationships to all other patterns).

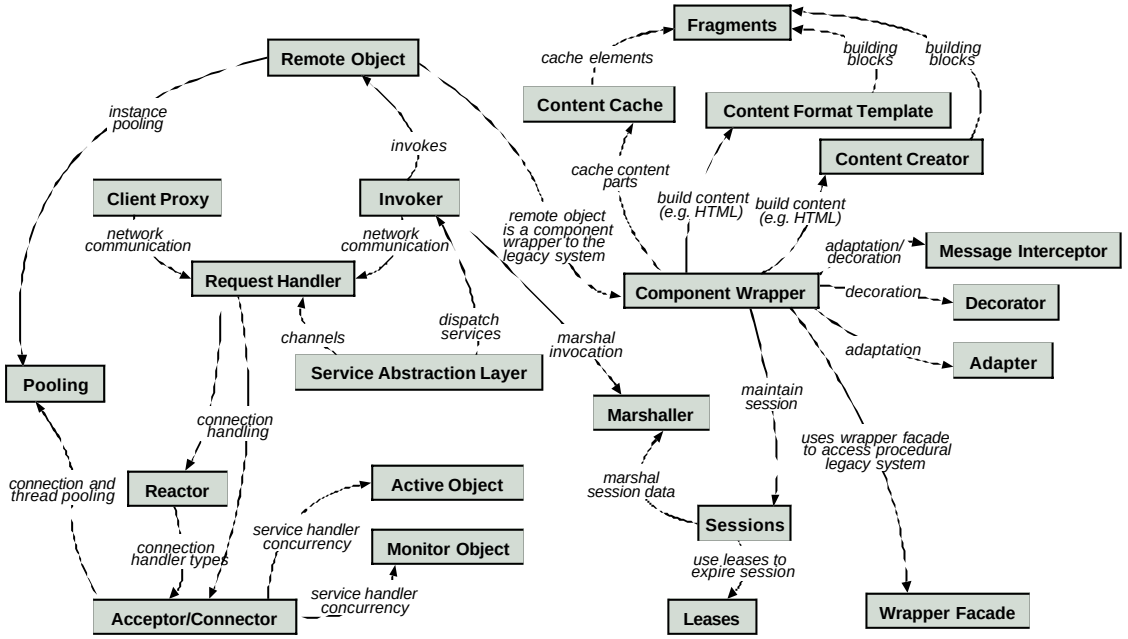

Figure 6: Overview: Pattern Language for Reengineering to the Web

As can be seen, the LAYERS and functionalities, discussed in the previous sections, can cleanly be separated from each other. That means individual parts of a solution can be developed in separate teams, except for common (i.e. reused) code and integration points.

The main integration points are the two connections client/server and server/legacy system. We have to deal with asynchronous invocations, concurrency, state preservation, and performance issues at these integration points. When reengineering to the web, we usually cannot change the legacy system or the CLIENT PROXY (as it is part of the web browser). Thus we have to deal with these issues somewhere in the server's processing chain of REQUEST HANDLER, INVOKER, and COMPONENT WRAPPER. Therefore these three patterns can be seen as the central patterns of the pattern language.

When more than one service is provided to more than one channel, we can use a SERVICE ABSTRACTION LAYER. This situation is quite typical when reengineering to the web, as possibly old interfaces still have to be supported and/or other channels than the web should be supported (in the future). Thus, as far as possible, decoration and adaptation of the server's processing chain should happen at the end of the chain, so that they are reusable for different channels. Only those parts that are specific for the web (i.e. for the HTTP protocol) should be handled at the REQUEST HANDLER. Thus it often useful to implement a generic MESSAGE INTERCEPTOR chain, in which add-on services (like SESSIONS, user control, LEASES, application-level marshalling, security, testing, etc.) can be defined for each COMPONENT WRAPPER object.

From a managerial point of view many important issues are only indirectly addressed by the software design and architecture patterns discussed in this paper. Examples are multi-user concerns, detailed technology selection, security and security audits, or data access related to converting a legacy system. Some of these issues are directly related to the pattern language, such as multi-user concerns which have to be considered when deciding for a SESSIONS model. As discussed above, technology choices only include those technology areas in focus of the patterns, and many important business aspects of a technology (such as costs, maintenance models, or credibility of business partners) are not in focus of the patterns. Other concerns go beyond the scope of this chapter and should be handled using the respective patterns in these areas. A good starting for security pattern is [Schumacher, 2003]. A good source for the broad area for data access to legacy systems and (relational) database management systems is [Keller, 2003].

USING THE PATTERN LANGUAGE IN PRACTICE: A CASE STUDY

As pointed out in the previous section, a main use of the pattern language (from a managerial point of view) is to provide a basis for communication both in early project phases and with non-

technical stakeholders. It can also be used to discuss about technology decisions in case the patterns are already implemented by some technologies/frameworks potentially used within a project.

We have used the patterns as a basis for communication between technical and non-technical stakeholders in various projects (see for instance [Goedicke and Zdun, 2002, Zdun, 2002b] for more details). In this section, we want to consider a few typical design decisions based on the patterns to illustrate these benefits of patterns for the communication compared to technical solutions. For space reasons, we cannot consider a whole project's design decisions here, but will only outline a few central ones. For each design decision we will describe the requirements within the project, explain the alternatives based on the pattern language and considered technologies/frameworks, and finally explain the solution.

Consider a project in which a large legacy system stores, delivers, and manages documents on different kinds of storage devices. The system should get a additional web interface, however, the old interfaces are still required. The code base is written entirely in C and has already distinct APIs for access by custom GUI clients. There are a few crucial requirements for the web interface, including load balancing, performance, and simple adaptation to the customer's IT infrastructure (i.e. simple adaptations of branding and layout of forms). For all other features the web system should be as simple as possible and only add minimal additional costs to the overall system. A management requirement is that a Java-based solution should be used, if possible.

The technology pre-selection for a Java-based solution limits the possible implementation alternatives. We could use a non-Java web server and let a web server module access the Java-based solution (e.g. via JNI). But this solution would require us to implement many parts of the pattern language by hand that are already somehow available in Java. A more Java-like solution is the use of a J2EE application server. Internally we can use different Java frameworks, including servlets, EJBs, JSP, and others.

As the potential customers for the system are quite diverse, the web application system has high demands for load balancing and failover for some customers, for others not. Here a simple solution with the whole web application running in only one application server is not enough for all customer sites, but it may be enough for some customers. That means a solution should be

scalable for customers with high hit rates, but yet impose no extra costs for customers with low hit rates.

These requirements suggest a scalable LAYERS architecture with REACTORS for each LAYER. INVOKERS are used for interconnecting the LAYERS. There is one primary ACCEPTOR/CONNECTOR that accepts web requests running in a frontend web server. This frontend web server also delivers static web pages and static page FRAGMENTS, and it invokes dynamic page creation in a separate application server. The application server contains its own REACTOR waiting for requests for dynamic content. Within the application server the INVOKER is situated. It is actually responsible for invoking the REMOTE OBJECTS that connect to the legacy system. There are also other REMOTE OBJECTS that perform other tasks than legacy integration. Note that there is also a simple variant of the pattern INVOKER in the frontend web server that is responsible for redirecting the invocations.

Each server has its own REACTOR, and INVOKERS are used to access the next LAYER. That means, the system uses an event-based form of communication between these LAYERS. As a positive consequence, there are no cyclic dependencies (back from lower LAYERS into higher LAYERS), which would make it hard to debug and maintain the web application.

Another consequence of this architecture is that the web application is highly down- and up-scalable. The extreme of down-scaling is that the whole web application runs within a single server that serves as frontend web server and application server. The extreme of up-scaling is that the primary ACCEPTOR/CONNECTOR is situated in a simple server that is just there for load balancing, then there is LAYER of multiple, redundant web servers delivering static web pages, then there is another load balancer, and finally there are multiple, redundant application servers. Figure 7 shows this scalable server architecture.

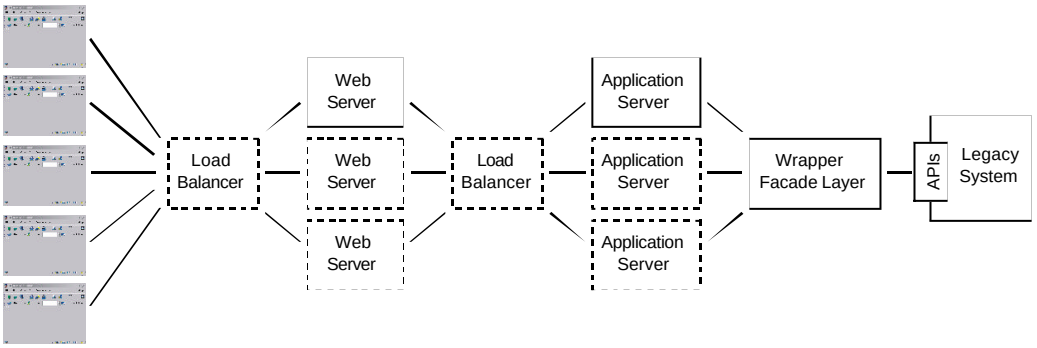

Figure 7: Scalable Architecture for the Web Application. Mandatory elements have plain lines,

optional elements are presented with dotted lines. The optional elements are only present in up-scaled installations.

The REMOTE OBJECTS in the application server provide access to the legacy system. Here, we actually have two choices: In some cases it would be possible to let the web system access the database directly (i.e. when data is only read from the database). In other cases we need WRAPPER FACADES to access the routines of the C-based legacy system to ensure consistency of data, especially when data is changed (i.e. the C-based interface access the internal database exclusively).

Because a distinct C-based interface is already existing for separate clients within the legacy system, writing WRAPPER FACADES is quite straightforward. A WRAPPER FACADE LAYER should only be bypassed, if really necessary; here, the C-based wrappers are quite lightweight and bypassing them for performance reasons is not necessary.

Note that in this architecture the legacy system is the bottleneck that cannot as easily be replicated as web servers and application servers. It is necessary to synchronize access from concurrent application servers to the legacy system. This can be done using MONITOR OBJECTS that queue the access to the WRAPPER FACADES (here MONITOR OBJECT is chosen instead of ACTIVE OBJECT because it is already supported by Java's `synchronized` primitive). If this is too much a performance penalty, direct access to the database and replication of the database might be an option.

There are two kinds of REMOTE OBJECTS: COMPONENT WRAPPERS to the legacy system and helper objects that contain application logic solely used for the web application. The REMOTE OBJECTS are realized as Enterprise Java Beans (EJB). Here, we have two main choices: We can use so-called entity beans that represent a long-living application logic object with container managed persistence. Alternatively, we can use session beans. A session bean represents a single client inside the application server with a SESSIONS abstraction. Session beans can either be stateful or stateless.

The main task of the COMPONENT WRAPPERS in this architecture is to compose invocations to the operations of the WRAPPER FACADES as needed by the web application. A central problem here is that legacy functionalities are required to be called in some order, within the same

transaction, or with other invocation constraints. The same problem also arises for entity beans of the web application (that do not access the legacy system).

As a solution to these problems, all REMOTE OBJECTS should be stateless and just used for composition of invocations to the stateful entities (COMPONENT WRAPPERS and entity beans). From the web, we do not access the stateful entities directly, but use the dedicated REMOTE OBJECTS.

This solution is also chosen because entity beans provide many features (such as synchronization, pooling, and lifecycle management) that impose a performance penalty. If synchronization is already done in the WRAPPER FACADE LAYER for access to a legacy system and/or if only concurrent read access to data is required, these features are not necessarily required, and thus entity beans do not provide an optimal performance.

A more high-level and more complex alternative to the architecture described above is the Java Connector Architecture (JCA). JCA is especially applicable for connecting to heterogeneous information systems such ERP, mainframe transaction processing, database systems, and legacy applications not written in the Java programming language. A COMPONENT WRAPPER (called resource adapter in JCA) defines a standard set of contracts between an application server and the information system: a connection management contract to let an application server pool connections to an underlying legacy system, a transaction management contract between the transaction manager and the legacy system, and a security contract to provide secure access to the legacy system. As most of these features are not required or only in a simple fashion, JCA is not chosen here. This design decision also avoids the performance overheads of connection, transaction, and security management.

Invocations from frontend web server to application server are typically realized by embedding URLs pointing to the pages or FRAGMENTS that are dynamically created in the application server. Somehow the application server has to embed these links as well as session identifiers and other context information in the web pages. This is typically done by multiple levels of MARSHALLERS. First, there is the HTTP MARSHALLER of the web server. It marshalls and de-marshalls all elements in the HTTP header fields. URL marshalling and de-marshalling is used to access the correct REMOTE OBJECT. Other context information can either be transported in the header fields, by means of cookies, or within the URL.

The overall system architecture already implements a simple form of CONTENT CACHE and FRAGMENTS because static FRAGMENTS are "cached" in dedicated servers. If additional performance is required, dynamic parts might be additionally cached by storing the FRAGMENTS in a database. There is also a CONTENT CACHE for the documents retrieved from storage devices within the legacy system. Thus a CONTENT CACHE in addition to these caching measures rather seems to be an overhead.

There are some customization requirements for the system, but the system functionality is similar in most installations of the web application. To ensure rapid deployment at the customer, a programmatic description of the web interface, as in CONTENT CREATOR pattern, is not the best solution. Thus the CONTENT FORMAT TEMPLATE language Java Server Pages (JSP) is used. Java servlets are used where programmatic, dynamic pages are required. JSP provides for simple changeability at the customer, because only the templates need to be adapted to the customer's requirements. The downside is that the rapid changeability is limited to those changes envisioned during design of the CONTENT FORMAT TEMPLATES.

Note that the architecture implements already a SERVICE ABSTRACTION LAYER, even if not used as such. That means, future channels can be integrated easily. For instance, one possible addition would be to offer a web service interface in the future.

Programmatic adaptations of COMPONENT WRAPPERS can be expected to be occurring in rare cases because the legacy system has already a mature interface API that is in long use. Thus changes can be simply introduced with ADAPTERS and DECORATORS. It is sensible to avoid the implementation effort and performance penalty of a custom MESSAGE INTERCEPTOR architecture.

There are already different ways of POOLING supported in this architecture, including connection pooling by the web server or component pooling by the EJB server. Thus it does not seem necessary to add additional pools.

CONCLUSION

In this chapter, we have presented a pattern language for reengineering to the web. It is built from patterns already published in other contexts. The main contribution of the pattern language is that it provides clear alternatives and sequences for applying a project for reengineering to the web. The patterns allow for important technical considerations in a mostly framework-neutral way (like estimations and technology/framework selection), as well as a means for communication.

Also they provide design guidelines for system parts that have to be developed from scratch, such as the wrappers and connections to the legacy system. Patterns, however, do not provide an out-of-the-box solution, but a design effort is required for each project. As different systems in focus of reengineering to the web projects can have quite different characteristics, this variability of the pattern approach is a strength for finding a suitable solution in such a project.

Note that some parts of the pattern language are concerned with forward engineering of web applications. That means they can also be applied without a connected legacy system; yet, in this chapter we have focused on the more complex reengineering to the web case.

REFERENCES

[Alexander, 1979] Alexander, C. (1979). The Timeless Way of Building. Oxford Univ. Press.

[Bakken and Schmid, 2001] Bakken, S. S. and Schmid, E. (1997-2001). PHP manual. http://www.php.net/manual/en/.

[Brant et al., 1998] Brant, J., Johnson, R. E., Roberts, D., and Foote, B. (1998). Evolution, architecture, and metamorphosis. In Proc. of 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). Pattern-orinented Software Architecture - A System of Patterns. J. Wiley and Sons Ltd.

[Coar, 1999] Coar, K. A. L. (1999). The WWW common gateway interface – version 1.1. http://cgi-spec.golux.com/draft-coar-cgi-v11-03-clean.html.

[Davidson, 2000] Davidson, J. (2000). Tcl in AOL digital city the architecture of a multi-threaded high-performance web site. In Keynote at Tcl2k: The 7th USENIX Tcl/Tk Conference, Austin, Texas, USA.

[Fielding et al., 1999] Fielding, R., Gettys, J., Mogul, J., Frysyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol – HTTP/1.1. RFC 2616.

[Franks et al., 1999] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. (1999). Http authentication: Basic and digest access authentication. RFC 2617.

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

[Goedicke and Zdun, 2002] Goedicke, M. and Zdun, U. (2002). Piecemeal legacy migrating with an architectural pattern language: A case study. Journal of Software Maintenance and Evolution: Research and Practice, 14(1):1–30.

[Jain and Kircher, 2002] Jain, P. and Kircher, M. (2002). Leasing pattern. In Proceedings of the Conference on Pattern Languages of Programs (PLoP), IIllinois, USA.

[Keller, 2003] Keller, W. (2003). Patterns for Object/Relational Access Layers. http://www.objectarchitects.de/ObjectArchitects/orpatterns/.

[Kircher and Jain, 2002] Kircher, M. and Jain, P. (2002). In Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany.

[Latteier, 1999] Latteier, A. (1999). The insider's guide to Zope: An open source, object-based web application platform. Web Review, 3(5).

[Neumann and Zdun, 2001] Neumann, G. and Zdun, U. (2001). Distributed web application development with active web objects. In Proceedings of The 2nd International Conference on Internet Computing (IC'2001), Las Vegas, Nevada, USA.

[Open Market, Inc., 1996] Open Market, Inc. (1996). FastCGI: A high-performance web server interface. http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm.

[Schmidt et al., 2000] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd.

[Schumacher, 2003] Schuhmacher, M. (2003). Security Patterns. (2003). http://www.securitypatterns.org.

[Sneed, 2000] Sneed, H. M. (2000). Encapsulation of legacy software: A technique for reusing legacy software components. Annals of Software Engineering, 9.

[Sorensen, 2002] Sorensen, K. E. (2002). Sessions. In Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany.

[Thau, 1996] Thau, R. (1996). Design considerations for the Apache server api. In Proceedings of Fifth International World Wide Web Conference, Paris, France.

[Vckovski, 2001] Vckovski, A. (2001). Tcl Web. In Proceedings of 2nd European Tcl User Meeting, Hamburg, Germany.

[Voelter et al., 2002] Voelter, M., Kircher, M., and Zdun, U. (2002). Object-oriented remoting: A pattern language. In Proceedings of The First Nordic Conference on Pattern Languages of Programs (VikingPLoP), Denmark.

[Vogel, 2001] Vogel, O. (2001). Service abstraction layer. In Proceedings of the 6th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany.

[Vogel and Zdun, 2002] Vogel, O. and Zdun, U. (2002). Content conversion and generation on the web: A pattern language. In Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP), 2002, Irsee, Germany.

[Welch, 2000] Welch, B. (2000). The TclHttpd web server. In Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference, Austin, Texas, USA.

[Zdun, 2002a] Zdun, U. (2002a). Language Support for Dynamic and Evolving Software A chitectures. PhD thesis, University of Essen, Germany.

[Zdun, 2002b] Zdun, U. (2002b). Xml-based dynamic content generation and conversion for the multimedia home platform. In Proceedings of the Sixth International Conference on Integrated Design and Process Technology (IDPT), Pasadena, USA.