Client and Legacy Integration in Object-Oriented Web Engineering

Karl M. Goeschka
Vienna University of Technology, Austria

Markus W. Schranz
Pressestext Austria

To be successful, e-commerce and Web information systems depend on systematic analysis and design processes. Even more important, our method is based on standard techniques like UML, XML, and Corba and allows for the integration of different kinds of clients from Java over HTML to WAP into a distributed environment. Through the smooth integration of highly heterogeneous legacy software and databases, our object-oriented Web design provides techniques to achieve this goal.

In a sense, Web engineering is the convergence of software engineering with the World Wide Web, which was initially dominated by the hypertext community. Like software engineering, it requires sound scientific, engineering, and management principles and systematic approaches (see http://fistserv.macarthur.uws.edu.au/san/WebEhome) to successfully design content and integrate middleware and database applications. Handling a Web service becomes more complex as its size increases, so standard Web page editors and site management tools prove inadequate for building complex services because they can't handle all tasks in the Web engineering process.

The engineering framework we present[1,2] manages the entire engineering process by supporting all stages of a service's life cycle: the analysis and architectural design, an object-based model for defining the navigation structure and layout, the implementation, the integration of reusable business objects and legacy database access methods, and the maintenance. We observed the object-oriented paradigm throughout all components of the framework using the object-oriented modeling language Uniform Modeling Language (UML).

We follow a top-down process: In the analysis step, we decompose the problem into subproblems in a so-called problem space. We then transform the result to a solution space, where we design a context-related solution. For the implementation phase, we decompose the solution space into the following areas (see Figure 1):

- the user front end, which deals with layout, navigation details, and usability issues;
- the business objects layer, which concerns standardized and tailored software components for computing server-side and client-side tasks, including legacy software integration; and
- the persistence and database access layer, which lets us automate database management system (DBMS) integration on the server's back end and to integrate legacy database systems.

Object-based front end

The front end consists of three components: a graphical editor, the object-oriented language Jessica,[3] and a Java-based compiler. The graphical editor models the architecture using objects that abstractly describe the entire Web service, written in UML. Jessica defines Web service components of any granularity and their internal and external relations using the Extensible Markup Language (XML). The compiler transforms the abstract service model into a consistent Web site, seamlessly integrating the layout and navigation components with the processing objects and database integration methods.

In engineering a complex Web service, two of the main questions are how to break down the overall content into individual pages and how to relate these pages to each other (that is, define an appropriate site architecture). Since Jessica uses object-oriented concepts such as abstraction, encapsulation, inheritance, and reuse of commonalties, we can adopt an object-oriented analysis and design technique.

We used UML to support the decomposition of complex services into manageable units. It also provides a number of different views for modeling object-oriented software

- UML class diagrams represent class packages, which are sets of logically related classes and the relations between them. Class diagrams in

1070-986X/01/$10.00 © 2001 IEEE

UML enable a high-level architectural view not necessarily related to a site's page structure.

- The UML use case concept defines actors and the course of actions (scenarios) they can trigger. In Jessica, for example, an actor could be a user clicking a specific control element on a page, and the action triggered could be a database query.

- Objects and classes defined in UML correspond directly to objects in Jessica.

Because of UML's widespread popularity, using it for the modeling provides several benefits—standardized graphical notation, easy communication, and well-known design patterns and decomposition methods.

Jessica defines information components, which are identified parts of the complete information modeled in Jessica entities. Often called Jessica objects, these are any tagged constructs that describe information components. The most relevant entities are objects, templates, variables, packages, and references.

UML provides a sufficient concept space to cover all elements required to model Jessica objects. UML includes objects, aggregation, association, and generalization and inheritance for class diagrams. We use these notations to visualize Jessica code.

A *class* is the descriptor for a set of objects with similar structures, behavior, and relationships. UML provides notation for declaring classes and specifying their properties. An *object* represents a particular instance of a class. It has identity and attribute values. Since Jessica doesn't model executable objects, no methods are explicitly presented in the graphical notation. Variables and attributes are properties in the UML sense.

Objects are represented by a rectangle with three compartments separated by horizontal lines (see Figure 2). The top compartment holds the object's name, the middle a list of attributes, and the bottom a list of variables. The code to the right of the graphical notation in Figure 2 illustrates the corresponding Jessica code created in the generation step.

Jessica uses the following UML relations to model component decomposition and service complexity:

- An *association* determines which object is referenced and which object embeds the reference.

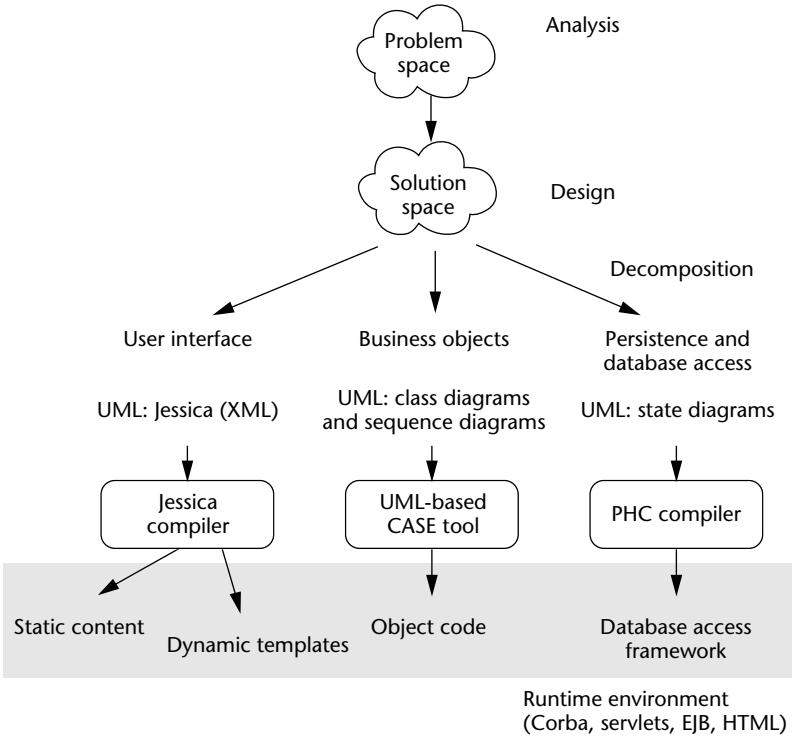
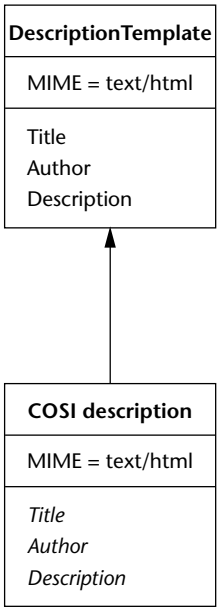


Figure 1. Web engineering framework.

- An *aggregation* in Jessica describes object embedding. Objects composed of other objects are described in this notation to represent the decomposition scheme.

- A generalization is mapped to template inheritance in Jessica. An object can inherit all variables (including the code) from the more

Figure 2. A generalization modeled graphically and in Jessica.




```
<J:OBJECT NAME="DescriptionTemplate"
          MIME="text/html">
<!--object contains variable parts-->
See the performance
<J:VAR NAME="title"/>
by <M:VAR NAME="author">
Get more details from the following description:
<J:VAR NAME="description">
</J:OBJECT>

<J:OBJECT NAME="COSIDescription"
          SRC="package:this.DescriptionTemplate"
<J:PACKAGE>
</J:OBJECT DST="package:this.COSIDescription.title">
Cosi fan tutte</J:OBJECT>
</J:OBJECT DST="package:this.COSIDescrition.author">
Wolfgang Amadeus Mozart</J:OBJECT>
</J:OBJECT DST="package:this.COSIDescription.description">
…</J:OBJECT>
</J:PACKAGE>
</J:OBJECT>
```

Deploying the object-oriented scheme used throughout all components of our Web engineering framework integrates the front-end, middleware, and back-end components seamlessly.

general template. A generalization is shown in Figure 2 as a solid-line path from the more specific element to the more general element, with a hollow triangle attached to the more general element.

- A *package* is a group of model elements. Packages themselves may be nested within other packages. A package may contain both subordinate packages and ordinary model elements. Jessica packages can contain objects, notes, all types of references, and other packages to model encapsulation and handle complexity.

Integrating the front end with the framework

Deploying the object-oriented scheme used throughout all components of our Web engineering framework integrates the front-end, middleware, and back-end components seamlessly.

Front-end objects, as described in Jessica, provide layout components for static information as well as for content dynamically created by server-side scripts, programs, and applications. The Jessica objects define models for Web pages that can be either invoked by static data or used as classes for the business objects, persistence objects, or database access methods to present processed information in a consistent layout.

Client and legacy database integration

We chose the Common Object Request Broker Architecture (Corba) as the middleware standard for distributed applications. On the server side we use Enterprise Java Beans for business logic and Java servlets or database programming for Hypertext Markup Language (HTML) or Wireless Markup Language (WML) generation. On the client side we use thick clients like Java but also thin clients with HTML and WML.

While we can easily integrate Java clients and server-side Enterprise Java Beans, several design breaks exist among a thin client with HTML, an object-oriented middleware, and a relational database. While we can overcome the design discontinuity between object-oriented software and relational databases with several approaches, such as persistence frameworks or encapsulated access, that between the client-side Web interface with HTML and the server-side middleware or database is more troublesome. In many typical Web applications the middleware is thin and often reduced to transforming information between the database and the user interface.

While the Web has become the place for gathering and distributing information,[4,5] integrating databases with it remains quite difficult. However, it turns out that most real-life applications need an underlying database on the server side, and many legacy database must be connected to the Web.[6]

Web developers usually restrict the user interface to pure HTML if they want it to work with any browser and not depend on proprietary or nonstandard extensions like plug-ins, Active-X, VBScript, or JavaScript. However, this means that users can only follow a link, press a button, or click on an image, which always results in a hypertext transfer protocol (HTTP) connection. Hence, any functionality resides on the server side.

Another problem arises from the different natures of databases and the Web. In a standard client-server application, both sides have their current session state, with a connection-oriented protocol connecting them. The Web, on the other hand, wasn't designed for client-server applications but rather for quick and simple delivery of linked hypertext documents. Hence, the HTTP is stateless.[7] The browser does store its user interface state, otherwise the back button wouldn't work, but without JavaScript it's impossible to use the information. Connection orientation, however, is necessary for real-life database transactions.

To overcome this deficiency, we use short uniform resource locator (URL) encoding. A short session identifier, called a handle, is passed back and forth between client and server with every HTTP connection. The server side stores the complete session state information (database transaction

and user interface states), usually in a database. This approach scales well to complex state information, and it can also be used with frames. Alternatively, we could use Netscape cookies (RFC 2109) to implement state management, but they can compromise user privacy. To avoid misuse of the session handle, we use encoded handles, and we use Netscape's Secure Sockets Layer (SSL) when user authentication is necessary for security.

As a connection between Web server and database, an interface compliant with a pure Common Gateway Interface (CGI) suffices for our approach. For a sequence of HTTP connections, however, CGI is rather inefficient due to the overhead of spawning a new process each time, which in turn opens and closes the connection to the database. We chose not to use a proprietary approach that links server programs directly with the Web server, choosing instead a method included in common application server software. Like FastCGI, it preforks multiple processes, which communicate with the Web server and stay connected to the middleware or database.

Architectural model

CERT advisory CA-2000-02 discourages using any client-side scripting language for security reasons. However, an HTML-only user interface is a disadvantage, because no functional dependencies can be implemented between a form's different input fields. Input into one field can't place a restriction on possible inputs for other fields. An HTTP session has to be performed (that is, a submit button pressed) before functional dependencies or restrictions can affect other inputs. Of course, forms prove useful in some cases (for example, multiple selections), but we chose to focus on links.

Our approach differs from most others because we use links to solicit user selections. We thus construct control elements out of sets of links grouped together, called passive HTML controls (PHCs). These links (also called elements) provide user interaction rather than cross references or navigation links. Each element of a PHC has two parts. The anchor tag defines the element's link, and the tag's content defines its output string. This approach is reasonable for a typical Web application, where most user interactions consist of browsing, selecting, and collecting information from a database. We simply predefine possible values for the user input for these cases. We use forms only when exclusive input has to be gathered (such as a name and address).

As long as the affiliated state information isn't changed, a dynamically generated Web page will always look the same. Web applications that ignore this rule can produce unpredictable results and corrupted output.

The general idea of modeling hypertext documents as state machines is well proven.[8] We propose to view a document as an abstract state machine that specifies the process of browsing within it. We can think of the linked structure of a document as the state transition diagram of an finite state machine (FSM). Hence, each document defines a state (that is, the state this document is displayed in, with only one document displayed at a time), and hyperlinks provide the state transitions. Clearly, this describes an FSM with a finite set of pages (states) and a finite set of links per page.

Each PHC has its own state machine. When users click on an element's link, a state transition generates different output. A PHC's complete state consists of all values required for presentation within the HTML page. These states can be grouped together to form a small number of explicitly defined states, a method well known, for example, in protocol machines. Each state now defines a class of dynamically generated pages, all similar in appearance, which are modeled with the front-end components. Hence, a PHC's complete state consists of one state value and a set of parameters that define the slightly different outputs for all the pages from the same class (that is, state). The parameters are mostly filled with values from the static database during state transitions. This separation of the database-dependent appearance of the page from the state value makes the state machine independent of the legacy database's arbitrary content.

This strict separation between state transition and output generation is a key feature to achieve stable and robust Web applications. We must guarantee a deterministic generation of the dynamic HTML pages independent of the last user interaction. Hence, as long as the affiliated state informa-

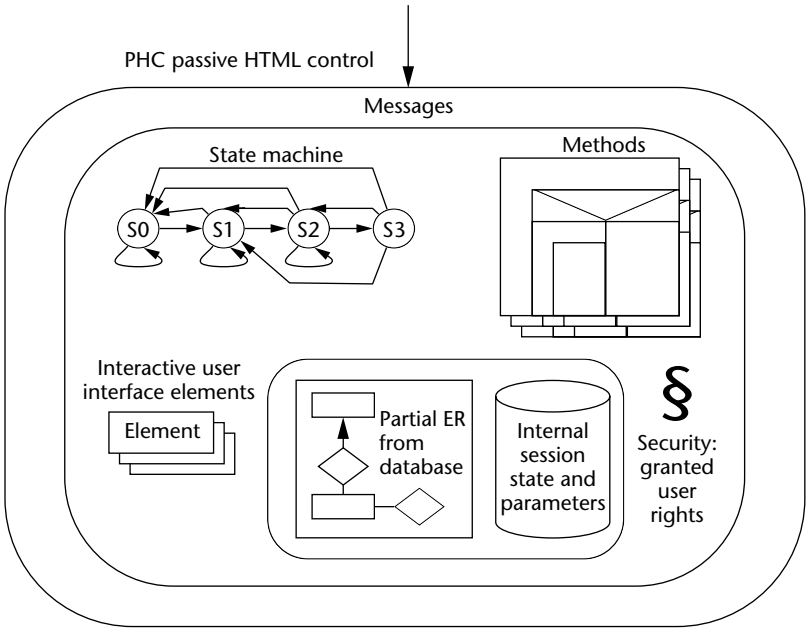


Figure 3. Object-based approach to passive HTML controls.

tion isn't changed, a dynamically generated Web page will always look the same, regardless of user interactions that have taken place in the meantime. Web applications that ignore this rule can produce unpredictable results and corrupted output.

Object-based design and layout

Figure 3 shows an object-based approach to a PHC's complete functionality as an integrative element between a client and a legacy database:

- A PHC presents itself to users as a set of interactive elements (links) grouped together to implement a particular control task. In this model, the only way users can interact with a PHC is by clicking on an element. Each element contains a method for a string and/or an image representation within the user interface and a method indicating what to do if users select this element.
- A PHC's behavior is modeled with a state machine. We used UML notation for state diagrams during design. A PHC has an internal data structure containing the session state information, including the session parameter set.
- A PHC is associated with a particular part of the underlying database. The PHC's methods generally use the session state information and the database contents, thereby integrating the legacy parts of the system.
- A PHC may contain additional methods to be called via messages from other PHCs or from the user interface (by clicking on a link). These methods implement more complex functionality and can also cause state transitions.

A PHC doesn't contain any information about layout or HTML representation. We completely separated this information into layout elements designed with Jessica and UML, as described previously. This makes the PHC functionality independent of possible future changes to HTML, and one PHC can be reused within different pages with different layouts. Moreover, application programmers and database experts can develop the PHC while graphical designers and multimedia experts implement the layout. These two groups, with very different skills, share a common interface instead of having to work together on one code.

Other front ends besides HTML—WML or Java—are possible because of this strict separation. A PHC's abstract description follows a virtual *n*-tier model to integrate different clients with the system's legacy parts, whereas with pure HTML the complete implementation is located on the server side. An application's developer can thus focus on the design, while the otherwise error-prone implementation is generated automatically.

Implementation and tools

In classic database design, development steps following the requirements analysis are strictly separated into information design and functionality design. In the area of relational databases the PHC approach belongs to the functionality design.

Whereas we exploited the advantages of a client-server model for design, in the case of pure HTML a PHC's final implementation occurs on the server side. We don't need functional elements on the client side. All intelligence is thus located on the server, making the control element passive in this sense. Every Web page, including all links, is dynamically generated from the database and the session state of all PHCs within the page, which can be thought of as an automatic subroutine for generating the whole page. The hyperreference URL of every link contains parameters for the global session handle and the state transition and parameter information of the appropriate PHC.

We use a timeout mechanism to overcome the problem of state information. It purges old state information from the database if the session wasn't

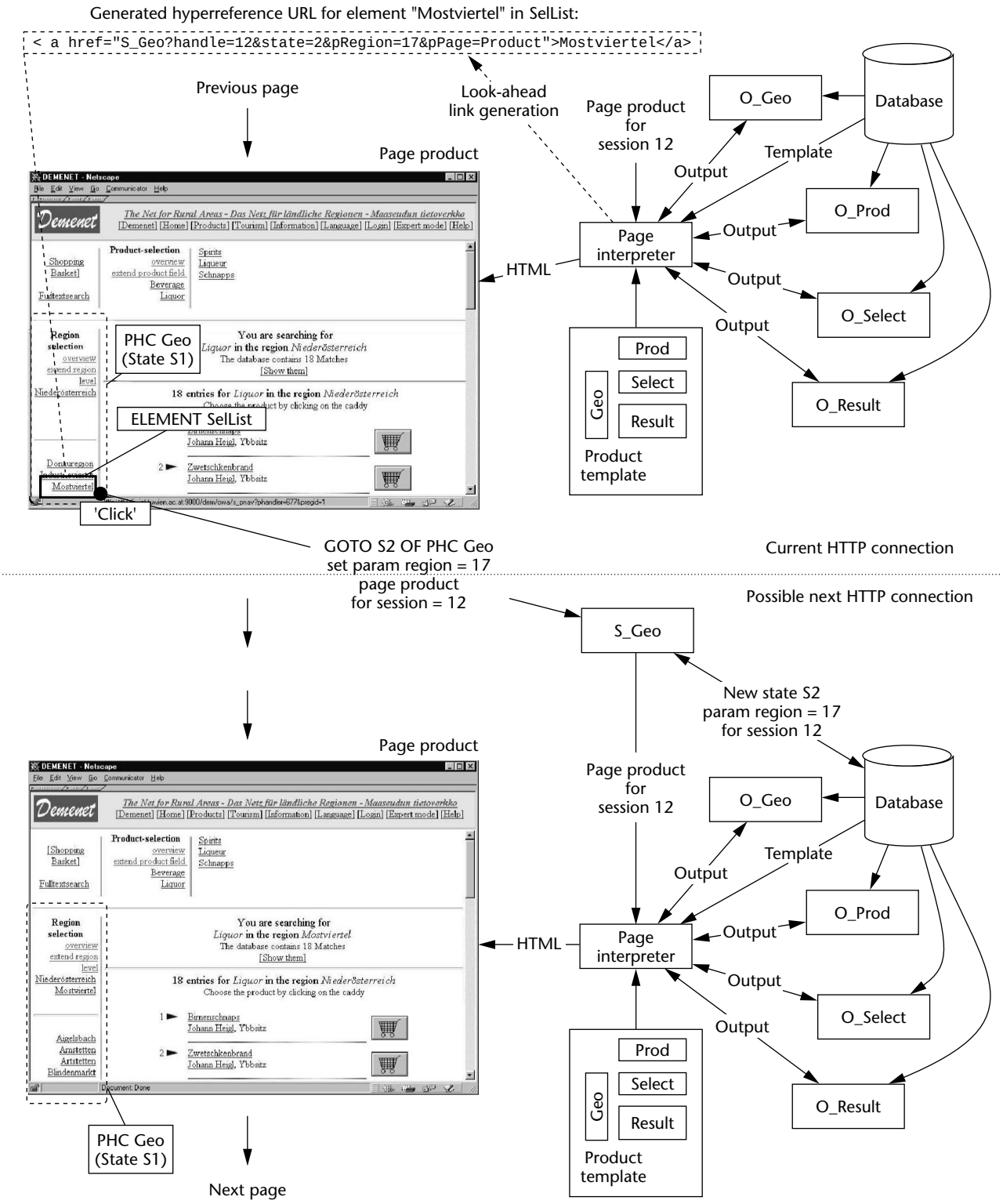


Figure 4. Look-ahead link generation and PHC user interaction.

closed properly, which is typical on the Internet. The problem with the browser cache that every URL-based state maintenance mechanism has to face can be overcome by setting the HTTP response header field "Expires" to a date in the past.

Figure 4 shows how the pieces finally generated—page templates and Java servlets (or PL/SQL procedures)—work together during the applica-

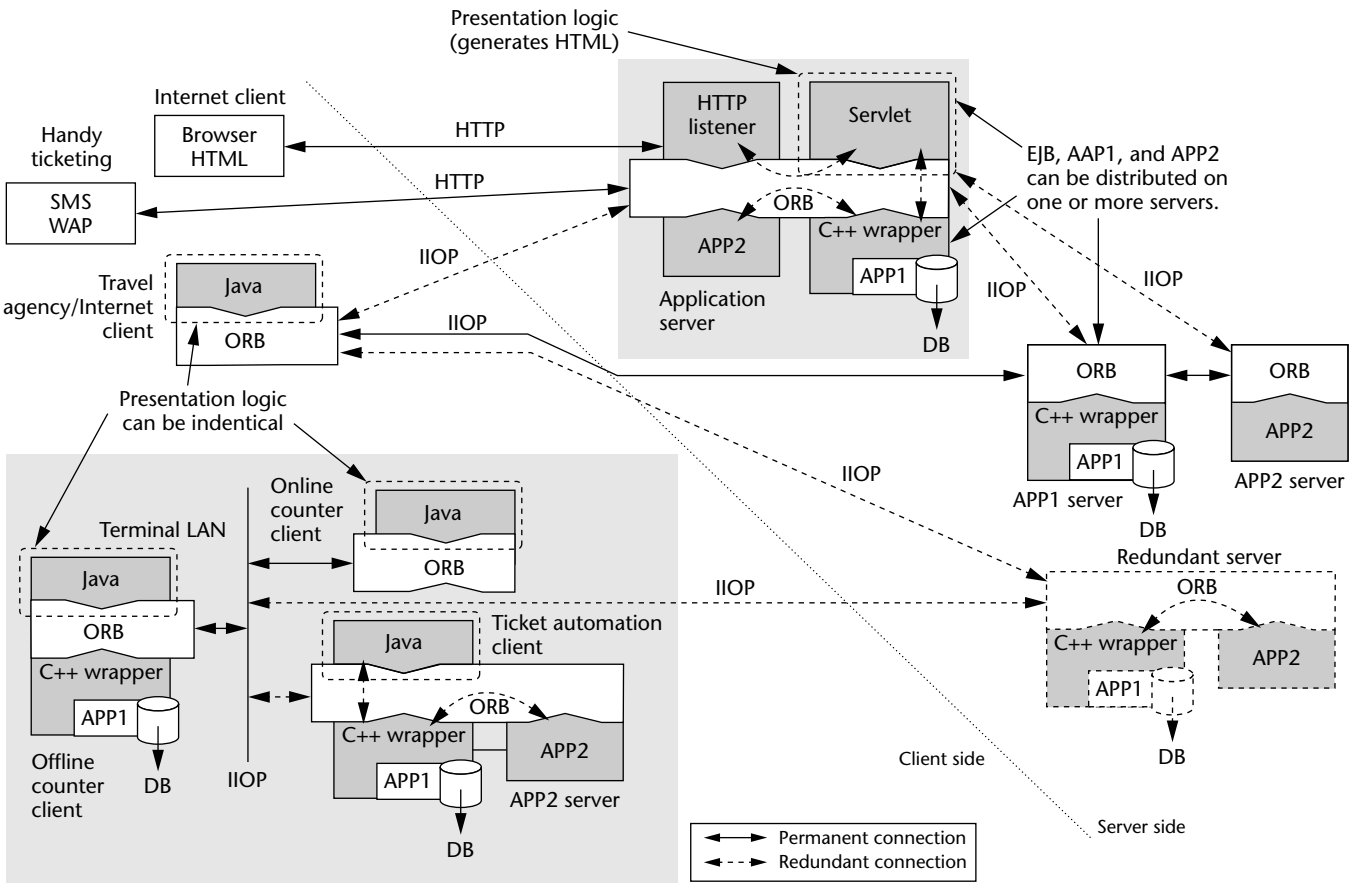


Figure 5. Austrian Federal Railways ticket sales architecture.

tion's execution. The PHC's state-handling method updates the session state information. The page interpreter then interprets the page templates and calls the generated output methods of the PHCs contained to produce the new page. The page interpreter is a database-backed Web application implemented in PL/SQL. Because every component necessary for the working application is stored in the database, we can use all tools for distribution, replication, and backup homogeneously. Moreover, the complete tool set is Web based, thereby also fostering unlimited Web collaboration. The complete design and implementation cycle falls outside the scope of this article but is described elsewhere.[2]

Case studies

We've performed several real-life implementations that show the successful integration of the subprocesses of our Web engineering framework.

Austrian Federal Railways ticket sales

The architecture of the e-commerce solution for the Austrian Federal Railways is currently being implemented by Ascom and Siemens. Figure 5 shows the main architectural concepts. The client-side flexibility is extremely important, since the system must deal with thick Java clients (for instance, travel agencies or the terminal ticket counter) but also with thin Web clients using HTML or ultralight clients using a short message service (SMS) or WAP for mobile phones.

The dashed line in Figure 5 shows the application middleware's dynamic distribution between the client and server sides, which can be transparently assisted by Corba. Moreover, some of the existing applications aren't even object oriented—we had to encase them in wrapper classes to link them with the object request brokers (ORBs).

After the initial wrapping and preparation of the legacy software, we were able to integrate the new Java-based modules smoothly. The system is flexible enough to allow for a consistent evolution of the whole system with future enhancements. Moreover, the Corba architecture improves maintainability, reliability, scalability, and availability.

Vienna Festival Web service

Jessica and UML have been used to create the

Web service for the Vienna Festival (Wiener Festwochen) since 1996. Figure 6 shows the basic architecture. The components are modeled as UML packages. Each package is a collection of Jessica objects and relates to other top-level packages. The topmost package (Service homepage in Figure 6), for instance, is a collection of general objects, such as the service's homepage, descriptions, contact information, and so on.

The screen shot in Figure 6 shows an elegant way of decomposing complex Web services into more comprehensible and maintainable units. The graphical editor for Jessica describes the overall service architecture using the power of the system decomposition features provided by UML to depict abstract relations between packages of objects.

Lower level structures for the Vienna Festival Web service are expressed in Jessica by specifying the contents of packages. The central entity of the abstract service description model is a Jessica object. Objects are abstract repositories for concrete information within a Web service. An object for the Vienna Festival Web service may describe parts of a Web document, specify a single page, abstractly describe models of Web pages, or even define a set or collection of documents.

Figure 7 shows a typical example of what Jessica objects can model. The title object describes a single addressable unit as one part of a Web document. The *Cosí Fan Tutte* opera object defines one specific Web page, in this case consisting of several Jessica objects.

Some objects identify common elements and may be reused to define several documents. A collection of such objects can describe an abstract model of Web pages that defines a common layout—for example, heading banners and trailing navigation bars—which is reused to create multiple concrete documents.

- *Document models.* Jessica templates define abstract models of Web documents. The models contain variables and constant specifications such as page structure and layout definitions. Such document models are defined in Jessica to describe, for example, a Web page's general appearance. An abstract model can define the necessary portions of HTML for the header, an *n*-column table, and a navigation bar at the bottom of the page (see Figure 7). The remaining part is defined by variables. The abstract model may be used to invoke multiple objects, all using the same presentation interface.

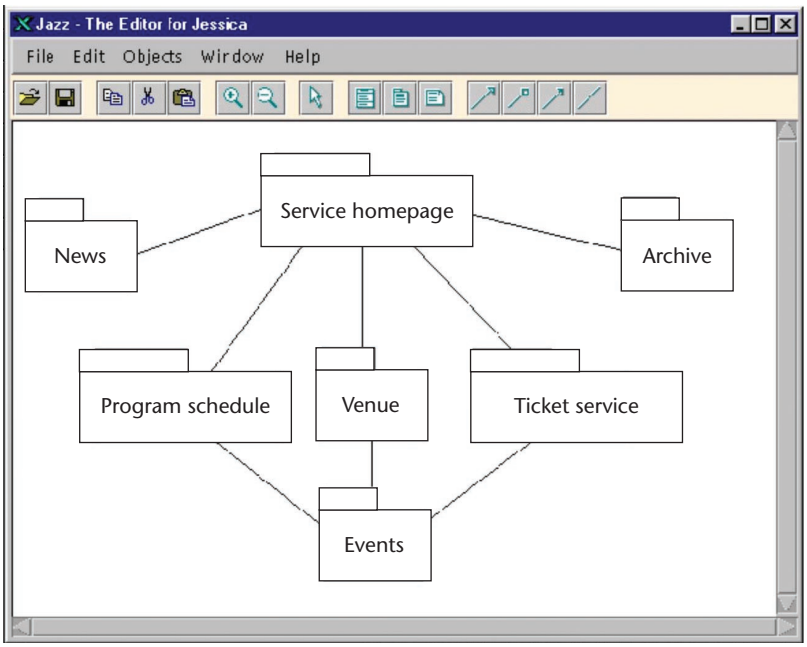


Figure 6. Vienna Festival Web service architecture.

- *Inheritance.* Invoking Jessica templates is one way that variables can be set and common attributes can be reused, but templates can also inherit from other templates. A typical use of inheritance is generic presentation templates for multilingual documents—German and English pages for the Vienna Festival—that inherit from a supertemplate holding more general layout information. Templates can even be invoked by programs (modules) that dynamically generate documents for search engine outputs or for interactive calendars, in the Vienna Festival Web service.

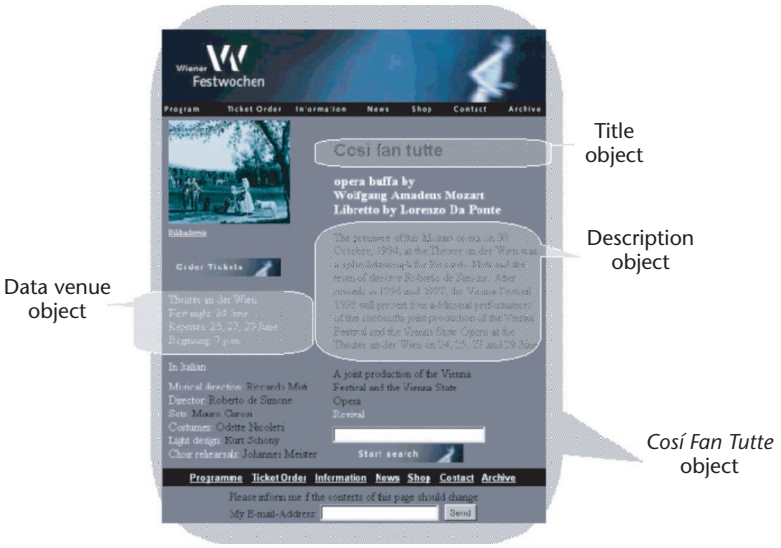


Figure 7. Screen shot of the **Cosí Fan Tutte** *Web page from the 1998 Vienna Festival.*

The inheritance of Jessica templates provides the integration of the decomposed approach of our Web engineering framework. Middleware programs and database engines can reuse the Jessica layout templates to present their results, thus maintaining a consistent layout throughout the entire Web service. The objects that compose the front end illustrated in Figure 7 are integrated from static Jessica objects and from database accesses that retrieve description and venue objects, thus using the middleware and back-end components of our Web engineering framework.

Conclusions

In addition to the case studies presented, we have implemented two further real-life projects—a product marketing and tourism information system for the Demeter project funded by the European Commission and a Web-accessible interactive database training for a graduate course with 90 students per year. Our experiences in these projects prove that our approach scales to larger systems. The main features and advantages of our approach are as follows:

- Building Web services with UML provides a standardized, nonproprietary notation and thus guarantees focused development and stability within complex (architectural) scenarios.
- We achieved strict separation between the layout and the content or application functionality. A widget is neither assigned to a single HTML page nor limited to HTML as the client-side user interface representation. This lets us use our functional elements in more than one page, in different HTML layouts, and even with different client-side implementations (Java,[9] WML, and Virtual Reality Modeling Language).
- With HTML we use links instead of forms to implement user interaction in most cases. Forms are necessary only to solicit user data.
- Web applications designed with the proposed methods are robust and stable despite the Web's stateless nature. We achieved this through the strict separation of state transition and output generation. Our method improves long-term integrity and overall quality.
- The pure HTML approach guarantees compatibility.
- The Corba architecture improves maintainability, reliability, scalability, and availability.
- De facto standards such as XML, UML, Corba, Enterprise Java Bean (EJB), and HTML support long-term evolution. Moreover, it's easier to find human resources with experience in these standards than in some proprietary approaches.

Despite the limits of pure HTML, complex information systems and sophisticated user interfaces can be designed with our Web-based tools and novel design and implementation techniques. We also provide generic interfaces for both intelligent software agents and robots from search engines by introducing a metadata model using richer links. On the back end, we provide interfaces for (legacy) middleware and database access. We can now generate different implementations automatically from the design language. With this development framework, the design and implementation of various database-backed Web applications is easier, faster, and less prone to error. It will also help to implement stable, robust, flexible, and scalable Web applications with complex functionality and to provide new ways to integrate legacy systems. We are currently working on a way to describe an user interface and hence derive client implementations for a particular application server from one single source. **MM**

References

1. M. Schranz et al., "Engineering Complex World Wide Web Services with Jessica and UML," *Proc. Hawaii Int'l Conf. System Sciences* (HICSS-33), IEEE Computer Society Press, Los Alamitos, Calif., 2000, p. 167 (abstract).
2. K. Goeschka and J. Falb, "Using Finite State Machines as Design and Engineering Models for Database-backed Web Applications," *Proc.'Hawaii Int'l Conf. System Sciences* (HICSS-33), IEEE Computer Society Press, Los Alamitos, Calif., 2000, p. 168 (abstract).
3. R.A. Barta and M.W. Schranz, "Jessica: An Object-Oriented Hypermedia Publishing Processor," *Computer Networks and ISDN Systems (*Proc. 7th Int'l World Wide Web Conf.), vol. 30, no. 1-7, Elsevier, New York, 1998, pp. 281-290.
4. G. Arocena and A. Mendelzon, "Viewing Web Information Systems as Database Applications," *Comm. ACM,* vol. 41, no. 7, July 1998, p. 45.
5. T. Isakowitz, M. Bieber, and F. Vitali, guest editors, Special Issue on Web Information Systems, *Comm. ACM,* vol. 41, no. 7, July 1998.

6. B. Adida, "Database-Backed Web Sites," *IEEE Internet Computing*, vol. 1, no. 6, 1997, pp. 78-80.
7. A. Iyengar, "Dynamic Argument Embedding, Preserving State on the World Wide Web," *IEEE Internet Computing*, vol. 1, no. 2, 1997, pp. 50-56.
8. D.P. Stotts, R. Furuta, and C.R. Cabarrus, "Hyperdocuments as Automata, Verification of Traces-Based Browsing Properties by Model Checking," *ACM Trans. Information Systems*, vol. 16, no. 1, 1998, pp. 1-30.
9. K. Goeschka, J. Falb, and W. Radinger, "Database Access with HTML and Java—A Comparison Based on Practical Experiences," *Proc. IEEE 22nd Int'l Computer Software and Applications Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1998, pp. 588-593.

Karl M. Goeschka is Chief Scientist at Frequentis Austria, an international company for fast, safe switching in air traffic control. He is a senior lecturer at the Vienna University of Technology and several colleges. His interests are in Web engineering, distributed systems, and packet-based telecommunication networks. He received an MS in electrical engineering, an MS in computer science, and a PhD in information technology, all with greatest honors.

Markus W. Schranz is currently Chief Technology Officer and board member of pte-online, a European online news agency. He is a senior lecturer at the Vienna University of Technology (VUT) and at an Austrian information management college. His research interests are in content management and syndication systems for the Web. He obtained a MS and a PhD in computer science, both from VUT.

Readers may contact Goeschka at Vienna University of Technology, Institute of Computer Technology, Gusshausstrasse 27-29/384, A1040 Vienna, Austria, email goeschka@ict.tuwien.ac.at.