Experiences in Engineering Flexible Web Services

The life cycle of Web service includes analysis, design, implementation, and maintenance stages. Our experiences in building and maintaining the annual Vienna International Festival’s Web site have led us to create engineering tools that cover all phases.

The World Wide Web is a highly flexible and dynamic medium. Users nowadays expect a Web site to offer interactive and up-to-date content, yet maintaining a useful, interactive site can become a challenge when the number of services and the amount of information exceed a certain limit. The Web engineering discipline deals with all phases in the life cycle of a Web service. The basic requirements for Web-based engineering are

- Scalability to enable future service extensions
- Simple update mechanisms for content managers
- A consistent navigational model for users
- Versioning mechanisms to keep track of changes in content, functionality, and layout
- Multilingual support for users
- The capability to integrate off-the-shelf software components and information in legacy data repositories
- Location independence for migrating the service
- Device independence for supporting the wide range of devices through which users can now access the Web (for example, wireless application protocol technology on cellular phones)

Service maintenance is one of the most important and costly issues in Web engineering. As with software management, the handling of a Web service becomes nontrivial once its size increases.

Our experiences in designing and managing the Vienna International Festival’s Web service since its inception in 1995 led us through three generations of Web engineering tools—HTML++, Jessica, and MyXML. We share eight valuable lessons we learned in building and maintaining such a large, dynamic online service.

In the context of this article, the content of a Web service is the information offered to users (for example, the price for a book), the layout is the way the content is formatted for presentation (for example, fonts, graphics, buttons, tables), and the business logic is the functionality necessary for dynamic interaction with users (for example, purchasing a product).

Web service life cycle

Some authors have likened Web engineering to the software engineering process, but fundamental differences exist. Web engineering includes the additional tasks of data analysis, information structuring, navigation management, and data organization. Using the software engineering process for Web engineering may be both difficult and inadequate.

Every Web service has a life cycle consisting of four major steps that must be supported—requirements analysis, design, implementation, and maintenance (see Figure 1). Most existing Web engineering systems concentrate on the implementation phase, and only a few support the design stage.

From the first, Web tools concentrated on content authoring using Hypertext Markup Language (HTML). Later, more sophisticated tools introduced “what you see is what you get” (WYSIWYG). Navigation, interface design, and site management tools were included in the next generation.

A typical Internet development environment is still quite fragmented, requiring a combination of tools. Several approaches have attempted to support all phases of the Web engineering process.

The vast majority of Web tools available today can create pages and graphical layouts using simple templates but can’t handle major updates.
involving multiple documents, dynamic data, and the integration of external information sources such as databases.

The management of a Web service isn’t finished after the last component or information node has been implemented. In our experience, that’s when the real work starts. Five years on the Web is a long time, and the Vienna International Festival (VIF) Web site has gone through major changes in appearance, technologies deployed, and services offered. The most costly task, in terms of both time and money, has been service maintenance, similar to software engineering, where up to 80 percent of costs are devoted to maintenance. Since changes are always necessary, Web engineering is the process of “designing for change.”

**Web service maintenance**

Most Web sites today change their appearance at least once a year to stay attractive. Minor changes in the look and feel of a site several times a year are common, and major modifications occur frequently. Modifications are motivated by a better understanding of user needs based on their feedback and on new requirements, optimization strategies, and new market directions.

Service maintenance involves information updates and content management, navigation management, version management, and service migration.

Ad hoc navigational links are embedded throughout Web services. Navigation management is necessary to check the validity of the links and resources for consistency and integrity.

Service migration is the movement of all or part of a Web service to another host. This becomes necessary as hardware is updated, performance requirements change, or new versions of software components become available.

Version management is an important issue in service maintenance, since versions increase the service’s manageability and maintainability. Version management must be applied to both the content and the scripts in the Web site. Standard version management systems have shortcomings when applied to the Web’s hypertext structure.

Just as objects have inertia in physics, services have a sort of inertia too—it isn’t always easy to modify an existing service, and the modifications may impact other existing services.

**Static and dynamic services**

We distinguish between static and dynamic services. Static services are rather stable over time and may integrate external data sources from an independent content-management system. Database integration introduces the strengths of database management systems to the Web.

Dynamic services, on the other hand, involve a high level of interaction with the user. A shopping cart e-commerce application is a typical example. The information offered by dynamic services may change frequently, and a business logic is necessary for specifying the interactions.

Dynamic services have a higher rate of change than static services, and modifying them often involves changes in the business logic and thus the source code.

**Web service flexibility**

We define a flexible Web service as one that’s easy to extend and maintain. The ability to modify the service’s graphical layout and appearance is one important flexibility issue for Web services, but the most important aspect is the ability to integrate new functional requirements without having to make major system modifications.

In contrast to printed media, Web users expect information to be up to date and to change frequently, and they expect the layout to reflect the latest graphical design trends. The attention spans of users in hypermedia environments are low, and the typical user is impatient. Changes must be integrated swiftly, without the need for the infamous and highly unpopular “under construction” sign. Providing a consistent service while keeping the layout flexible is the great challenge for Web service engineers.

The first generation of HTML document standards provided little support for layout (user interface) flexibility. Attempts were made to eliminate these shortcomings by extending the HTML standard with Dynamic HTML and Cascading Style...
Sheets. Meanwhile, the World Wide Web Consortium’s promising Extensible Markup Language (XML) standard and its XML Style Sheet Language (XSL) got off to a good start and gained popularity quickly. Recent trends toward mobile computing and an increase in the range of Web-enabled devices have increased the need for device independence. Web engineers today can’t assume that users will be using a popular browser with well-known display characteristics. Services and content must be flexible to support a wide range of display sizes, network bandwidths, and connections.

To achieve complete flexibility, we must separate the layout, content, and business logic of a Web service and cover its entire life cycle. Engineering tools are necessary that can react to data and structural change requirements that come up during maintenance. We developed several tools in our Web engineering research and have deployed them successfully for the VIF Web site.

Case study: The Vienna International Festival Web service

The annual Vienna International Festival (Wiener Festwochen) is a major cultural event, featuring eminent international directors, performers, and ensembles in operas, plays, lectures, concerts, musicals, and exhibitions. Visitors come from around the globe to take part in the festivities, which usually last five weeks and take place in various famous theaters and concert halls.

Our group built and has been managing the festival’s Web presence since 1995. The Web site provides an extensive range of services, such as information on the current program, online ticket sales, maps of major stages and venues, and an archive of performances since 1995 (see Figure 2). The information and interactive services are available in both English and German, with the potential for extending the service to other languages.

The number of services offered varies each year. We analyze user feedback and site statistics annually and tune the services offered and the appearance of the site accordingly. These modifications range from minor changes to significant transformations of the services provided.

Main VIF components

The festival program, ticket reservation system, and archive are the main components of the site. Other services inform users of stage highlights and press coverage and present text translations of some musicals and stage performances. Users can switch between the German and English versions at any time. All of the information is indexed and coupled with a search engine.

All performance and event-related information and the ticket sales data are managed by an external relational database management system (DBMS). Because the program information changes frequently, online Web forms enable content managers to modify the information in the DBMS. We retrieve the information stored there every night and compile static HTML pages from it. Figure 3 depicts the various VIF components.

Ticket reservation service security

The multistage ticket reservation system lets users select tickets for specific events and add them to a virtual shopping basket that can be changed at any time prior to checkout. Because users must supply private information (such as the address and credit card number) to complete an order, security is provided by the common Secure Sockets Layer (SSL) frequently used to encrypt or sign messages and to authenticate users.

Engineering methodology

The festival organization currently supplies us with VIF information in the form of photographs, paper copy, and electronic copy. Because the program changes every year, we’ve found it difficult to define a set of commonalities on which to base our design. We sometimes have had to reengineer the Web site literally from scratch.

Designing and implementing complex, structured information for a Web site requires a methodical approach. The Relationship Manage-
ment Methodology (RMM)\textsuperscript{10} views hypermedia as a vehicle for managing relationships among information objects. RMM focuses on the design, implementation, and construction phases for hypermedia applications. It has seven steps for hypermedia service management:

1. Entity–relationship (ER) design
2. Slice design
3. Navigational design
4. Conversion protocol design
5. User-interface screen design
6. Run-time behavior design
7. Construction

Steps 4 through 7 are beyond the modeling of hypermedia information and must be done either manually or with tools that provide automated support for them.\textsuperscript{11}

We used RMM to structure the VIF information, and we deployed the ER data modeling technique\textsuperscript{12} to model the VIF database.

**VIF Web engineering tools**

In this section, we present the different generations of Web engineering tools we developed for the VIF site. We give a brief overview of HTML++, Jessica, and our latest tool, MyXLM, and discuss how these tools evolved over the years.

**First generation: HTML++**

HTML++ defines an abstract model to engineer complex Web applications. HTML++ objects are abstract descriptions of Web documents. So-called HTML++ classes describe objects that have static and dynamic contents. A compiler maps the abstract model to real-world Web applications by generating the HTML documents described in the model.

Macros have proven to be useful in providing service flexibility. HTML++ uses them extensively for modeling static portions of Web documents, allowing hundreds of documents to be updated instantaneously. For example, we can update a default footer that appears on every page in the Web site simply by editing it and recompiling. In the first year, new sponsors were added to the site regularly, and macros made it easy to accommodate these changes.

HTML++ also has built-in support for Practical Extraction and Report Language (Perl) scripting (see http://www.perl.com). Perl was probably the most popular Web scripting language in the mid-1990s and is still widely used today. From the object information in HTML++ documents, the HTML++ compiler generates Perl templates that can be used in building dynamic Web pages. The Perl scripts encapsulate the layout information, which can be invoked with variable data.

One shortcoming of the HTML++ system was that it didn’t completely separate layout, content, and business logic. The business logic was separated from the layout and content, but they were specified in an HTML-like language and hence were intermixed. Although we achieved a relatively high degree of service flexibility, adapting some services to new requirements wasn’t always easy. Often, the degree of flexibility achieved depended on the developer’s HTML++ design experience.

We used HTML++ extensively for powering the VIF Web site and extended the idea of using an abstract model to describe Web services to build the next solution, the Jessica language.

**Second generation: Jessica**

Jessica\textsuperscript{5} is an object-oriented hypermedia publishing processor that models a hypermedia system based on user-defined objects. As with HTML++, a processor transforms the abstract models into Web documents.

The object-based approach is intended to support usability, manageability, and maintainability of complex Web services. The objects in the Jessica language consist of attributes and content. The attributes may be constant or variable. Objects can...
be embedded in others, inherit information from others, or be grouped into packages. Jessica also supports data organization, making it easy to model entities defined in methodologies such as RMM.

Our aim in designing Jessica was to create a system that supported all stages of a Web service life cycle—the design, implementation, and maintenance. The Jessica language models the data organization and navigation management, and the Jessica processor handles the implementation.

Describing a Web service independently of the target medium requires a suitable document format. Because of the conceptual similarities, the Jessica language was defined using XML. Jessica documents use an XML namespace specially defined for Jessica entities. Detailed information on Jessica can be found elsewhere.

The main difference between HTML++ and Jessica is that Jessica supports XML, which lets us use other XML tools and editors that already exist or are being developed.

Additionally, we wrote a custom-tailored editor for Jessica—Jazz—that uses the Unified Modeling Language (UML) for modeling Web sites. UML can be useful for modeling Web applications, but we found that it isn’t scalable for modeling large, complex Web sites.

Jessica supports content and layout separation by using objects as templates to define a common appearance, but as with HTML++, we discovered that this support was still not enough. Furthermore, Jessica lacks mechanisms for supporting flexible, dynamic pages. Given the importance of personal interaction and e-commerce on Web sites, we felt the need for a more extensive support for dynamic content.

A comparison of Web engineering systems showed that Jessica was useful only for certain service domains. The system is quite complex and is overkill for small Web sites.

Furthermore, Jessica isn’t easy to learn and requires a high degree of expertise. It also has no direct built-in support for relational DBMSs, which we found to be a critical factor in the rapid development of partially or fully database-backed Web services.

Based on these experiences, we designed our latest tool, MyXML.

Third generation: MyXML

Our MyXML template engine uses XML along with XSL to tackle the complete layout and content separation problem. XSL is an XML-based language that lets the developers specify a layout for the information defined in an XML file. Hence, a strict separation exists between the layout and the content. This separation has a special importance for mobile devices (for example, personal digital assistants, mobile phones, palmtops) since the layout can be modeled in real time according to a device’s display characteristics.

MyXML is currently under development, but we’re evaluating a prototype on the VIF Web site. MyXML addresses the complete layout, content, and business logic (LCL) separation problem that was missing in our previous tools.

To remove two of Jessica’s shortcomings, we designed the MyXML language to make it simple and straightforward to learn. The tool has direct built-in support in the language for relational DBMSs, reuse of layout components, and transparent handling of Common Gateway Interface parameters.

MyXML allows developers to generate a layout encapsulating source code for dynamic content. The layout information for an e-commerce ticket service, for example, can be generated automatically from the templates developers have designed.

Additionally, our tools generate XML information from a DBMS. This gives the system considerable layout flexibility. A common layout can easily be applied to the XML data using a simple XSL transformation.

We expect that MyXML will prove useful in reducing development time. We designed the tool to support the different stages in the Web service’s life cycle and to increase reuse. Meanwhile, several other tools (see http://xml.apache.org/cocoon) have been introduced that use similar technologies and approaches to tackle the LCL separation problem. Future Web sites that wish to support a wide range of heterogeneous devices and services will depend on such systems.
Lessons learned

Back in 1995 when we first started building the VIF Web site, the Web was still new to many users. The number of visitors since then has grown at a rate of more than 130 percent every year. This is partially because of the increase in quality and the number of services offered, but the main reason is probably the tremendous increase in the number of Internet users.

At the beginning, the Web site played a trivial role for the VIF. The organizers considered it more a novelty than an important part of their advertising plan. Now, five years later, the site has become indispensable for the VIF. A large number of tickets are sold online through the VIF Web site every year with more than 2000 tickets sold last year.

We can identify eight lessons that we’ve drawn from our experiences in managing and building the VIF Web service every year.

Lesson 1: Invest in the design process

Most commercial Web developers tend to start hacking away at building a Web service without having a good idea about the design and methodology that should be used. Often, the newest and trendiest Web technologies are deployed, and the customer is told that these technologies will “improve” the site. For example, we were once asked to implement a graphical design consisting of nine frames. Because of time constraints, the graphics company hadn’t considered design issues such as download and page rendering times.

A Web engineer should invest time and money, two resources that are always scarce, to go through all stages of the Web service life cycle methodically, using tools that support all these stages. The investment in the design process will pay off once the service has to be adapted and modified.

Lesson 2: Train the content managers

In our experience, managers responsible for the Web site’s content often lack the basic knowledge and understanding necessary for dealing with a dynamic, evolving medium like the Web.

Training the people responsible for the content, informing them about the limitations and possibilities of the Web medium, and teaching them basic concepts will make the Web engineer’s job easier. Content managers will understand the design decisions, and this will increase the acceptance of the system among those who have to use it. Furthermore, users will have realistic expectations about the system.

Lesson 3: Choose flexibility over functionality

Astonishingly, this simple rule is ignored far too often. Flexibility is the most important aspect of a service. Current functionality comes second.

Functionality decisions should be made carefully, since the system’s complexity increases proportionally to its functionality. The general rule of thumb is the more functionality, the less flexible the service. One way to reduce this problem is to implement the functionality using flexible components. For example, we implemented the VIF e-commerce functionality by using separate flexible LCL components, and we tried to avoid any fancy animations that would hurt the system’s flexibility.

Lesson 4: Document the system

Although this lesson sounds trivial, we often failed to document the system we were building. One reason for this was the general misconception that the services we were building were “simple” anyway and were to be used only for “this year.”

A Web service often is put together using a number of different technologies and dependencies. Because of the nature of the Web, the architectures of Web applications are distributed and not always easy to comprehend. Documenting the code will ease reengineering and maintenance. It’s as important in Web engineering as it is in software engineering and often equally ignored.

Lesson 5: Reengineering is harder than engineering

A common belief is that Web reengineering is easier than engineering because functionality already exists, but in our experience, reengineering an existing service is often harder than engineering a new one.

An existing service isn’t always flexible, well documented, and easy to understand. The technologies used might be out of date, and interfaces might not be supported anymore. In our case, the underlying technologies improved over time, but their complexity increased. Hence, a considerable effort was needed to upgrade the functionality from HTML++ to Jessica to MyXML.

Web reengineering hasn’t received much attention by the research community, but its importance will increase as organizations feel the need to adapt their existing services to support emerging technologies such as mobile, wireless computing.
Lesson 6: Be prepared for changing requirements

Just as in software engineering, the aim in Web engineering is to design and build a system that meets the user requirements.

Our experience has shown that new requirements will come up once the implementation phase is over, and the layout will change. Because the Web is a highly interactive and changing medium, a large number of “small” changes are often necessary. For example, new static links may need to be integrated into a dynamic shopping cart application. Once, we had to integrate a new sponsor logo into every single page in the Web site, including dynamic services.

The way to deal with changing requirements is to have an abstract model that can be tuned as needed. This observation is reflected in some of the Web engineering tools we’ve developed. The trick is to keep the design highly flexible and to define clear, open interfaces.

Lesson 7: Choose quality over deadlines

Just as Rosenfeld et al. observed, customers often have unrealistic expectations about deadlines. Inform customers of the possible realistic deadlines that will assure the quality of the design and the service. Obviously, a trade-off is necessary between quality and the deadline to be met. Using the incrementality principle, we build the site in increments. Implementing critical components first and noncritical components after the hard deadline is one way of dealing with the quality and deadline issue. In the VIF Web site, we use an evolutionary model. We first build the core Web services (e-commerce and program information) before the news conference that announces the Web site. Then we incrementally add noncritical functionality such as Web-based content update interfaces.

Hacking a complete service to meet a deadline will have serious consequences in the maintenance phase, possibly raising costs and adaptation time.

Lesson 8: Use version management

Software engineers use version management to keep track of source code changes and releases. Web engineers can’t create a release and “freeze” the state of the Web service, because the state constantly changes. Web services consist of collections of heterogeneous applications and files, and modifications to different parts of the service are often necessary every day.

Most sophisticated version management systems have concurrency support. A Web service usually has more than one content manager and system developer. Mechanisms are required to ensure that changes aren’t lost when people work concurrently on the same set of files. A version management system takes care of this by providing concurrency control mechanisms—files and objects must be checked out and back in by the person working on them.

We’ve been using the CVS versioning system for the VIF site during the last couple of years. Version management has proven to be highly useful, especially during the implementation phase. If modifications made to the system aren’t stable and problems occur, version management lets us revert to previous versions. Furthermore, concurrency control mechanisms let us work safely on a large number of files without worrying about the possibility of changes being lost. Even if only two people were working on the system, we found that concurrency conflicts occurred frequently.

Conclusion

The Web engineering discipline has received far too little attention from the research community. The general trend has been to develop tools to solve small problems, whereas the root of the problem—the lack of design patterns and Web requirements analysis techniques and methodologies—has been ignored. Not enough attention has been paid to experiences gained from previous hypertext research.

Nielsen predicted that the Web eventually will suffer a usability meltdown unless the vast majority of Web sites are improved considerably. He stated that the emphasis must be placed on quality content and software and not on “dazzle and coolness.” The development and use of sound, proven Web engineering techniques will determine the Web’s future.

Acknowledgments

We’d like to thank the Vienna International Festival for its financial support and cooperation and Robert Barta for his pioneering work in developing tool support for the VIF Web site.

References


Engin Kirda is a research assistant and PhD student in the Distributed Systems Group at the Technical University of Vienna (TUV). His main research interests are Web engineering device-independent flexible services, software engineering for distributed systems, and mobile computing. He holds a MSc in computer science from TUV. He is a member of IEEE and ACM.

Mehdi Jazayeri is professor of computer science and head of the Distributed Systems Group at the Technical University of Vienna. His research interests are software engineering, programming languages, and distributed systems. He was program chair of ESEC/FSE 97 and program co-chair of the International Software Engineering Conference (ICSE 2000).

Clemens Kerer is a research assistant and PhD student in the Distributed Systems Group at the Technical University of Vienna. His interests include distributed systems, Web engineering, Web and Java security issues, and software components. He holds a MSc in computer science from TUV. He is a member of the IEEE and ACM.

Markus Schranz is currently Chief Technology Officer and board member of pte-online, a leading European online news agency. He is a lecturer at the Technical University of Vienna and at an Austrian information management college. He obtained both his MS and PhD degrees in computer science from TUV. His research interest is in content management and content syndication systems for the Web.

Readers may contact the authors at DSG, TU Wien, Argentinierstrasse 8/ 184-1, 1040 Vienna, Austria, email {E.Kirda, C.Kerer}@infosys.tuwien.ac.at.