Dynamically Generating Web Application Fragments from Page Templates

Uwe Zdun
Specification of Software Systems
Institute for Computer Science
University of Essen, Germany
zdun@acm.org

ABSTRACT

Web-based applications are typically required to be highly customizable and configurable. New application requirements have to be introduced rapidly, often without stopping the running application process. Moreover, in many cases the same business logic has to be presented to different channels and/or user interfaces. In this paper we present a dynamic page template architecture for decomposing configurable and representational fragments of the application from the business logic. Page templates consist of static XML files and of dynamic class definitions. The XML-based page templates can be used for declarative definitions of configurable fragments, say, by the end-user with a graphical tool. The page template classes can be used for behavior specification, say, for defining common styles of decoration of the presented pages. Both parts are dynamically loaded into the web application environment and composed with the web objects. Thus, the configurable and representational fragments of web objects are dynamically generated from the changeable and extensible descriptions in the page templates. The concepts presented are applied in various larger software systems. We will present a highly customizable conference management system as an example case.

Keywords

Web Engineering, Dynamic Software Architecture, Object-Oriented Scripting

1. INTRODUCTION

Web-based applications usually play an important, business critical role in a company's information systems. For instance, web-based applications may be the portal for a company's customers to reach the company's services. Or the web browser may serve as a simple way to let employee's access the company's information systems from any remote location. Despite their importance, many interactive web applications are not built with a conceptually clean, integrating model. There are several typical, recurring problems in the realm of developing and maintaining interactive, web-based applications. This work discusses solutions to some typical problems in the context of configuring and representing applications on the web.

In general, the application class of interactive, web-based applications can be characterized by generating of formatted content on request. That means, the content is usually not available in pre-built files. The generated content often has to be formatted in different markup languages, such as HTML, WML, and XML. In many cases, as for legacy integration, other formats, such as graphical user interfaces or textual representations, have to be supported as well. Moreover, the content usually has to be provided to different channels with different protocols, such as HTTP, COM, CORBA, UMTS, etc.

Interactive, web-based applications have a set of more or less unique requirements. Each web-based application has to be capable to represent its services using HTML pages. Those have to be transfered via the HTTP protocol to a remote client. Thus, a web-based application has to integrate an HTTP server or use it for transporting the created HTML pages to the client.

Some web-based applications only serve web-based clients, but often other information systems of the company have to be integrated as well. Therefore, in many applications, especially when legacy applications are extended with a web-based interface, other forms of representation than HTML pages have to be supported as well. That means, a "conceptually clean" software architecture of a web-based application separates business logic from web-based representation and from the HTTP transport services of the web server.

Persistence services are another typical requirement. Since HTTP is a stateless protocol we typically have to preserve states, say, of a session, a business transaction, or a web object life-cycle. Typically, when HTML content is sent over HTTP, the state may be preserved for a session by encoding a state identifier in the URL, in cookies, or in HIDDEN form elements. For some applications persistent objects have to be supported. Therefore, a persistence store or a database have to be integrated into the web application environment. Moreover, a mapping from state identifier to persistent object has to be performed (automatically) by the web application environment.

There are a set of other typical components, such as database access, certain security services, registry services, directory services, etc. that have to integrated in many web-based applications. Therefore, an important aspect of a platform for web-based application development is a component model for simple extension with new black-box components.

Another important aspect of typical web-based applications is the coherent representation of the business logic on the web (and on other presentation destinations). Regarding web-based representation we can identify a set of recurring problems:

- *Extensibility:* The representation has to be easily extensible with new content. The representation style and the behavior of the application have to be extensible as well.

- *Changeability:* Content, representation style, and application behavior should be changeable ad hoc. In principal, it

SAC 2002 Madrid, Spain

should be possible to change the web-based application without stopping the web services.

- *Rapid Prototyping:* Often web-based applications are rapidly developed in an interactive process with the users. Thus, the programmers and editors should be able to rapidly implement new functionality as a prototype that can be evolved into an operating web service in a stepwise manner.
- *Short Deployment/Maintenance Cycles:* Usually the deployment/maintenance cycles for web-based applications are only a few hours or days. Therefore, incremental evolvability is a central requirement.
- *Integration with Other Representation Forms:* Often the business logic of web-based applications has to be represented to different channels, such as widget sets, distributed object systems, WAP, UMTS, etc., as well. It is undesirable to create application and content for every channel/user interface anew by hand. In an integrated concept we have to deal with limitations or special capabilities of the various formats.
- *Non-Programmers as Editors:* Usually the editors of web-based content are non-programmers. Thus, it is often undesirable that program code has to be modified in order to modify the web-based representation. But often flows of web applications tend to be quite complex, so that some modifications require programming. Therefore, the web application environment should support proper ways of modification for all stakeholders in the development and maintenance processes, including high-level programming and content editing. If different views for different stakeholders are provided, the web application environment has to ensure consistency.

Some aspects of those and other problems regarding the dynamic and coherent creation and modification of web-based applications are tackled by various approaches. In the next section we will explain and classify such approaches and motivate problems that are not solved well by the named approaches yet. Afterwards we will present a generic architecture as a solution to these problems. The architecture consists of two main constituents: XML-based page templates and class-based page templates. Those are composed with the web objects. Thus the business logic is cleanly encapsulated, but nevertheless a generic generation of a web-based representation from these page templates is achieved. Finally, we will present a highly configurable and extensible conference management system as an example case.

The generic architecture is implemented in the object-oriented TCL extension XOTCL [11] with the ACTIWEB components [12]. ACTIWEB provides us with an environment that supports the development of web-based applications by offering an integration with an HTTP server [10], a web object model [12], and services such as persistence, registry, metadata, etc. The given code examples are implemented in XOTCL. Nevertheless, the presented concepts can be integrated in any (extensible) web-server or application server environment (see the next section for a short survey of such environments).

2. APPROACHES TO FLEXIBILITY IN WEB-BASED REPRESENTATION

The basis of any interactive, web-based application is the HTTP protocol [4]. When the user agent (e.g. a web browser) sends a request, the requested page may be available on the file system of the server or it may be dynamically created. There are various approaches to create and decorate content dynamically. Here, we want to introduce a major distinction into template-based approaches that generate HTML pages by substituting certain elements in template files and constructive approaches that construct a web page on the fly. Of course, there are several systems supporting both approaches.

Template-based approaches, such as PHP [1], ASP, JSP, or ColdFusion, let developers write HTML text with special markup. The special markup is substituted by the server, thus, a new page is generated, which composes content dynamically into the template. On the first glance, the approach is simple and well-suited for end-users, say, by using special editors. Thus the HTML design can be separated from the software development process and can be fully integrated with content management systems.

However, real web-based applications usually require more complex interactions than simply expressible with templates. Sometimes, the same actions of the user should lead to different results in different situations. In some applications the observable states of the HTTP protocol (that is, the events generated by the web server) are not sufficient to model all business transactions. Thus additional programs for management of such transactions have to be introduced. Those are separated from the template files and often not conceptually integrated.

Most template approaches do not offer high-level programmability in the template or conceptual integration across the template fragments. Thus, the same fragments of a template often have be implemented redundantly. Application parts and design are not clearly separated. Thus template fragments cannot be cleanly reused, but only by cut-and-paste. Complex templates may quickly become hard to understand and maintain.

Constructive approaches generate a web page on the fly in a program. Often they offer a distinct API for constructing web pages. Usually they are not well-suited for end-users since they require knowledge of a programming language. However, they allow for implementing a more complex web application logic.

The most simple constructive approach is the CGI interface [2]. 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 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 [15] 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.

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 [16], 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 API's are quite complex, and applications tend to be monolithic and hard to understand.

Custom web servers, such as AOL Server [3], TclHttpd [19], WebShell [17], Zope [9], the Ars Digita community system [7], or ActiWeb [12] 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. Most often 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, such as WebShell, offer modules for web servers, as in this case Apache, as well.

Many approaches combine the template-based and the constructive approach. However, often the two used models are not well-integrated.

Often different requests coming from different clients communicating over different channels have to be supported. It should not be required to change the business logic every time a new channel has to be supported or a new service is added to the application. In many business systems, this problem is solved by a SERVICE ABSTRACTION LAYER architecture [18]. A service abstraction layer is an extra layer to the business tier containing the logic to receive and delegate requests. It implements different channel adapters to support calls via different protocols. When pages have to be generated dynamically, as in interactive web applications, the SERVICE ABSTRACTION LAYER may serve for more purposes: it may be a MESSAGE REDIRECTOR [6] for indirecting symbolic calls (e.g. encoded in an URL string) to actual implementations. This indirection can be implemented in a configurable way. E.g. in our architecture we also use it for implementing user authentication and other security services centrally.

3. A GENERIC ARCHITECTURE FOR GENERATING APPLICATIONS FROM DYNAMIC PAGE TEMPLATE

In Figure 1 we can see an overview of the base classes involved in a generic architecture for building applications with dynamic page templates. Essentially, the presented dynamic page template architecture consists of the following parts: *XML page template files* contain the configurable description of the web pages and their data. *Pool pages* are web objects which contain the business logic described in one or more data objects. Thus the pool pages are used to display a set of similar data objects. An XML configurator component is used to interpret the data in the XML page template files. The web server component is used to present the information in the pool pages to the user.

The representation of the pool pages is decomposed from them. First, each pool page aggregates one or more *user interface builder objects*. For instance, for a purely web-based application one `HTMLBuilder` is aggregated. It is used to construct HTML pages using a generic interface. However, with the same interface we could also arrange widgets or build textual representations. Finally, the concrete representation style, that is, how pages are decorated, is defined in a *page template class*.

Consider the simple example of an address book: The internal logic of the entries of the address book can be stored in one data object per address book entry. A pool page aggregates them and defines the logic of the collection. Moreover, a page template class defines the style of representation. The concrete building of representations in the targeted HTML format is delegated to an HTML builder object. If a request is sent via the web browser, then the pool page dynamically builds the page in its responsible user interface builder and the resulting HTML page is given to the HTTP server.

Note that the described architecture is a combination of a template-based and a constructive approach for building web applications. Parts of the page definition that require constructional definitions are defined in the pool page and its page template class mixin. In the associated XML files we can define templates of configurable and representational fragments. Moreover, the approach decomposes the representation style, which applies for all pages of one or more pool pages, from the construction of the information contained in a page.

We do not have to access the concrete user interfaces internals, but only use the generic methods of the user interface builders. This way, pages for different user interfaces can be built with almost the same code.

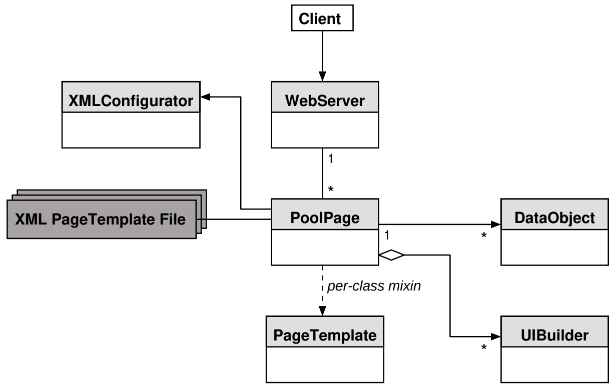

Figure 1: Architecture Overview: Page Templates are Defined as Mixin Classes and in XML Files

4. XML-BASED PAGE TEMPLATE FILES

XML-based page templates are used to encode the configurable data, page content, and page style information of a web-based application in a generic way. The information contained in the XML page templates are typically required to be changeable and extensible. However, the information should not be completely independent from the representation, but should contain viable representational aspects, such as color, position, etc. as well. Those abstract representational aspects can be used by the responsible user interface builder component to achieve a consistent layout even in different representation styles and user interfaces.

XML is used as a data format for page template files because it is a simple, text-based standard which is readable for human beings. Efficient parsers for XML exist and can be reused. XML can be used independently from concrete representation formats and styles, that means it is independent from the concrete HTML-based representation on the web. Moreover, it is language and platform neutral.

In a web application one or more XML files describe the configurable elements of a web page. From these XML files an abstract, user interface independent view of the representational information *and* the information contained in the pages can be deduced. Moreover, since the XML file is a textual representation of the information, presented labels can be exchanged. This way, we can for instance deal with applications that should support labels in multiple languages or which should be changeable ad hoc for other reasons. Since it is quite simple to write customized parsers for special XML representations (see [13] for a discussion), it is not intended that human editors – who are usually non-programmers – necessarily do have to change the XML text itself. Instead simple GUI- or web-based editors can be written, that allow for application-specific editing of the information contained in the XML text.

To implement all page forms, required by a typical web application, several different page template types are necessary. Here we will only discuss two typical templates: a template that represents a bundled key/value list and a template for forms. Other useful templates are for example: frames, menus, option list, navigation routes, etc.

4.1 XML Page Templates for Bundled Key/Value Lists

Page templates for bundled key/value list are a simple form for introducing configuration options into a web object. They represent a list of options with values that can be bundled together, say, because they share a common purpose or they belong to the same web object. Typical option lists are used to specify configuration options or to specify strings, so that labels can be changed on the fly.

Each group has a group name, which is defined as a top-level XML node. Inside of each group the values are simply given as XML elements, like:

```
<rating>
  <0> Awful </0>
  <1> OK </1>
  <2> Great </2>
</rating>
```

Here, a list named `rating` is defined with just three elements. Each element contains the associated label string as value. Thus, for instance, the value `0` is configured with the label `Awful`. If users want to exchange labels or ratings, they just have to change the XML file.

At runtime (for instance, when the associated pool page object is initialized) the information in the XML file is parsed and mapped onto dynamic variable slots. In XOTCL variable slots are language-supported, in other languages such as C++ or Java, we can simulate them with simple techniques, as for instance discussed in the PROTOTYPE-BASED OBJECT SYSTEM pattern [14].

In the concrete XOTCL solution, we dynamically create a new associated TCL array on the pool page object. In the client we only use the method `getArrayIntoSelf` from the `XMLConfigurator` component:

```
Class Result -superclass PoolPage
Result instproc init args {
  next
  [[self] set configurator] getArrayIntoSelf \
    [self] rating {0}
}
```

In this simple example a result class is created. In its constructor `init` an associative array with the name rating is created on the `self` object, which is the newly created result object. The array is automatically filled by the configurator with array names `0`, `1`, and `2`, and with the corresponding values from the `rating` XML element. As last argument a list of required options is given. Here, the application only requires the value `0` because it is used as a default value. All other values are application independent and can be defined entirely in the XML file. For instance, we may add a value `3` to the XML file. It then would be automatically used in the web application.

4.2 Page Templates for Forms

As a second simple page template example, we discuss form templates which resemble HTML forms. The data defined in the XML forms should be automatically defined on each data object of a pool page and it should be possible to make the new slots automatically persistent. The form defines an HTML-like layout, but it is abstract enough to be displayed in other widget sets/user interfaces as well.

A simple example is an address form:

```
<address method="get">
  <name type="text" size="30"> Name </name>
  <email type="text" size="30"> Email </email>
  <address type="textarea" cols="50" rows="8">
    Address
  </address>
</address>
```

In the example, we firstly define the name of the form as top-level XML element. As an argument the method for data submission of the form is defined. Here, the default `GET` is used for submitting data, but some forms also require uploading of data. This is handled with the method `POST`. Afterwards, the form elements are defined with optional arguments defining the type and layout of the used form elements. Again, the labels visible in the form are defined as XML elements.

In an address list class, which has addresses as data objects, at first, the address input form is extracted. Thus a new object is created dynamically which defines the form. In a second step, the data from the form is extracted:

```
Class AddressList -superclass PoolPage
AddressList instproc init args {
  ...
  [self] set addressInputForm \
    [[self] set configurator] extractForm address]
  set data [ConfigForm getDataFromForms \
    [[self] set addressInputForm] {name email}]
  ...
}
```

With `getDataFromForms` a list of all data elements defined in the XML-based template is returned. Again, we are able to define required options on the form data, such as `name` and `email`. All other data elements are defined on the data objects as they are found in the XML file. The corresponding HTML forms are automatically created accordingly. For instance, to extend the address list with `phone` data, the user simply has to add this information to the definition in the XML file. To remove a data element, such as `address` or `phone`, it just has to be deleted from the XML definition. Afterwards, it is automatically not displayed to the user interface anymore.

4.3 Extending the Page Template Mechanism

In general, it only requires a few lines of code to add new types of XML page templates to the XML Configurator or a subclass of it. For instance, let us consider the implementation of the mapping of bundled key/value lists (as we have discussed them above) to associative arrays:

```
XMLConfigurator instproc getArrayIntoSelf \
  {co arrayname {requiredOpts ""}} {
  set p [self]::xmlParser
  foreach topNode [$p info children] {
    if {[$topNode content] == $arrayname} {
      foreach child [$topNode info children] {
        $co set ${arrayname}([$child content]) \
          [$child getFirstPCData]
      }
    }
  }
  foreach var $requiredOpts {
    if {![${co} exists ${arrayname}($var)]} {
      error "Required option <$var> is missing."
    }
  }
}
```

We can see that the interpretation of bundled key/value lists by the `XMLConfiugrator` is handled in a single method. During initialization, the `XMLConfigurator`'s XML parser has read all XML template files, so that we use these information later on. In an xoXML parser [13], all XML nodes are children of the `xmlParser` object. Thus, we can iterate with a `foreach` loop over the top nodes and check whether it is the searched array name. If it is found we simply create an array slot dynamically on the calling object (`co`) with the array name of the XML element. The PCDATA of the child is used as array value. Finally, in a second `foreach` loop, we check whether all required data is there; otherwise we raise a runtime error.

In a similar fashion we have also implemented forms. Forms build a form object dynamically, instead of setting a dynamic associative array on the calling object. Some extensions require more than one XML element to be parsed at once. For instance, forms with options lists are implemented with one form template plus one array for each option, defining the option's values/labels list.

5. BUSINESS LOGIC IN POOL PAGES

The business logic is encapsulated in pool pages. Each pool page is a web object, that is, it inherits from the `WebObject` class of the ActiWeb framework. With the inherited methods from `WebObject` it can represent itself to the web and can be accessed with an URL. In other words, the web server displays a set of coupled web pages per pool object. Usually, each exported method of a web object is responsible for displaying one page. In ActiWeb we use the standard form of URL encoding for distributed calls to web objects:

```
http://host:port/objectName+methodName+arg1+...+argN
```

For instance, if we want to access the method `addName` on the object `names.html` with the argument `Annie` on a host `www.names.org`, we would use the following URL e.g. in the web browser:

```
http://www.names.org/names.html+addName+Annie
```

Each object may have a method `default` which is automatically called, if no method name is provided.

In ActiWeb dispatching of remote calls is handled by a place which integrates the web-server. We have to define in the place which web objects are exported. For each web object we can additionally define which methods are exported. Only those methods of the exported objects can be accessed remotely. In Figure 2 we can see that in general all pages are accessed via the place's Invoker component, which acts as a Service Abstraction Layer [18]. It indirects the calls to the responsible pool pages with a Message Redirector [6] architecture. The pool pages operate on their data object and return the result to the Invoker. The Invoker handles all central, call-related concerns, as for instance security or authentication of web objects.

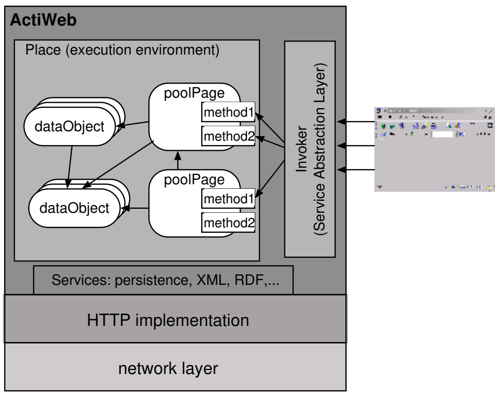

Figure 2: Pool Pages are Accessed via the Place's Invoker

5.1 Data Objects

Each pool page may contain one or more data objects. If only one object has to be represented to the web, the pool page object may also be a data object (e.g. with mulitple inheritance from the data object and pool page classes). But most often more than one object with the same or a similar web representation have to be displayed. Then we can decompose into pool page and associated data objects.

Since many data objects require persistence of the stored data, the most prominent task of the data object class is to handle the connection to the persistence store. Persistence in ActiWeb is handled transparently for the persistent web object and its clients. This way, the data which is imported from the XML-based template files is dynamically created on the data objects and defined as persistent data. The data object class contains methods for initializing, setting, and getting the list of persistent variables on its instances. The class maintains a list of currently persistent variables on the instances, which may be changed dynamically as well. The list can be accessed from each instance. Therefore, persistent data of a web object may be arbitrarily changed at runtime, e.g. by template reloading (when changes to the template occur).

As we have seen, the method `getDataFromForms` of the `XMLConfigurator` component returns the list of data in an XML form page template. This way, we can automatically find out, which data elements are defined in the XML-based template definition. In the default case, a pool page may initialize its data object class with the persistent data from all its forms:

```
set pdata [ConfigForm getDataFromForms \
  [[self] set addressForm] {name email}]
DataObject setPersistentData Address $pdata
```

Here, the data from the `addressForm` is retrieved. Afterwards, we define the persistent data for the `Address` class as the result (`pdata`) of the `getDataFromForms` operation.

5.2 User Interface Builder

An important part of building a web representation are user interface builders. They are used to translate the user interface independent representation which can be deduced from the page template files/classes into a concrete user interface representation. In the presented framework, the user interface builder are relying on an abstract subset of HTML because HTML is the primary user interface used in the domain of web applications. It is relatively straightforward to map the calls to the `HTMLBuilder` to other user interfaces, such as widget sets. Therefore, clients use an abstract interface of the class `UIBuilder` from which special interfaces are derived, like an HTML and an TK builder:

```
Class UIBuilder
Class HTMLBuilder -superclass UIBuilder
Class TKBuilder -superclass UIBuilder
```

The special user interface builders have to implement the used subset in the targeted user interface. Moreover, they should display the pool pages according to the layout information contained in the page templates. For all other layout tasks the builders are free to manage the layout according to the conventions of the user interface. For some user interface types, such as WAP or textual representations, a simpler, reduced interface can be provided, since not all capabilities of the HTML subset are fully supported.

In the builder classes, a method `clear` cleans up the builder object. In the HTML builder a method `toString` returns a string-based representation of HTML text. Similarly, other user interface types may contain other finalization methods to retrieve the result.

For each element `X` of the user interface we usually have a `startX` and `endX` method in the builder classes. For instance there are `startDocument` and `endDocument` methods to start/end a new document. The optional configuration of such elements is given by "-" arguments, as for instance `title` for the whole document. For simple elements, such as text or table cells, there is also a method `addX`, which allows for combining start, content, and end of an user interface element with one method call.

As a simple example we build up an HTML page in the object `ui` for displaying a name. After building the HTML page in the memory it is simply written to the standard output with `puts`:

```
set name "Annie"

HtmlBuilder ui
ui startDocument -title "Annie's Name"
ui startParagraph
ui addString "Name: $name"
ui endParagraph
ui endDocument

puts [ui toString]
```

Here, we see that it is quite easy to exchange the user interface to another type of interface by just changing the class of the `ui` object.

We would only have to change the finalization method `toString` to the finalization method of the other user interface (like starting an event loop and/or refresh for a widget set).

Sometimes applications require more than one user interface at once. For instance, in the process of reengineering legacy application to the web, the old interface should often be still supported as well. In such a case we can write a user interface builder as a wrapper for the old interface and stepwise move the old interface to that builder. In a second step we can support the second web-based interface as well. Thus we have to support building of multiple user interfaces for a single pool page.

Usually a pool page aggregates its user interface builders. XOTcl provides an aggregation facility with introspection of children. Moreover, we can dynamically introspect the type of an object. Thus, we can iterate over all children and build the user interface on all children with the type `UIBuilder`:

```
foreach uiObj [[self] info children] {
  if {[$uiObj isType UIBuilder]} {
    $uiObj startDocument -title $title
    # build up document
    ...
    $uiObj endDocument
  }
}
# call finalization routines of each UI
...
```

6. PAGE TEMPLATE CLASSES

Some aspects of a page template cannot be sufficiently described by the purely descriptive elements of an XML page template. Behavioral elements, which are independent of the business logic, have to be imposed on the pool pages. A simple example is a presentation style which has to be build with the methods of the user interface builder classes. Those behavioral page templates are defined by page template classes. Page template classes are usually mixed-in as a per-class mixin to the pool page.

Per-class mixins are classes which are mixed into the precedence order of all instances of a class and its sub-classes. They are a special language construct of XOTcl, which supplement multiple inheritance by introducing extrinsic concerns co-existing with the instances' heritage order. Since mixins can be used as dynamical roles of a class, we can use them to describe the representation style of a class of web objects in a decomposed way. Per-class mixins are dynamically changeable at runtime.

It is not intended that the end-user has to write the page template classes, but they can be written by programmers once and then be used by a user tool also dealing with the XML-based page templates. Then class-based page templates are rather used as styles which dynamically decorate the pool page classes. Note that XOTcl also supports per-object mixins, which extend a single object with an extrinsic class, in the same way as per-class mixins extend all instances of a class and its sub-classes. Therefore, a behavioral page template can also be imposed per object, if necessary.

We use page template classes in most cases to define the general layout of a set of pool pages. For instance, we may collectively define for a set of objects how page construction is started and ended:

```
MyPageTemplate instproc startPage {title builder} {
  [[self] place] instvar {opts(logo) logo}
  set uiObj [$builder newChild]
  $uiObj startDocument -title "MyPages: $title"
  if {[info exists logo]} {
    [self]::html addImage -src "$logo" -align middle
  }
  ...
}
MyPageTemplate instproc endPage {} {
  foreach uiObj [[self] info children] {
    if {[$uiObj isType UIBuilder]} {
      [self] addFooter $uiObj
      $uiObj endDocument
    }
  }
}
```

Here, two methods `startPage` and `endPage` are defined. `startPage` firstly builds a new object of type `uibuilder` as a child with an automatically created name. Then the document is started with a customized title. A logo is added, if it was defined on the array `opts` from the corresponding XML-based page template file. In `endPage` all user interface objects get a footer and then the document end is added.

Note that directly programming page template classes is a convenient way for a programmer to change representation styles. However, it is rather cumbersome for the end-user. For systems in which the end-user requires control over the representation style, we propose to connect an XSLT engine to the page template class so that the user can specify representation styles in standard XSLT files (and has not to learn the programming language).

7. AN APPLICATION CASE: A CONFERENCE MANAGEMENT SYSTEM

In the previous sections we have seen the (internal) design and extension mechanisms for a page template framework. In this section, we will demonstrate their usage with a conference management system as a simple, but non-trivial, example of an interactive, web-based application. The system presented was used successfully for the whole conference management life-cycle of the OOSS'00 and ASE'01 conferences (164 submissions, 50 referees). The system provides pages for abstract and paper submission for the authors. The conference management pages let the conference chair create referees, notify authors and referees per email, view various overviews, and enter the program committees decisions. Referees can then vote for the papers which they want to referee. After finishing the voting period, the conference chair is able to assign referees to the paper. Now the referees must be provided with a page where they can see the data of their papers and download them. After refereeing the referees must be able to enter the data via a web form. The referees can access other referee results for consensus review.

Obviously a conference management system must be highly configurable and extensible: submission processes of different conferences are usually not exactly the same, different conferences require different presentation forms of different data elements, and the pages should be presented with the conference logo, style, and other visual properties. Some of the pages have to be access protected: the referee pages should only be accessed by the assigned referee, the conference management page only by the conference chair. It cannot be expected that each conference chair is an expert in programming web applications. Thus end-users should be able to make simple changes, such as adding properties to forms or changing the logo, by themselves.

In Figure 3 we can see how the architecture of pool pages and data objects in the conference management system is constructed from the architecture presented. First a special page template class for the conference is derived. It retrieves information, such as logo, background color, conference name, etc. from the XML page template, and builds up a HTML page style for all conference pages. It uses the aggregated HTML user interface builder object. Thus it can also be used with other representations in the future. The `ConfPageTemplate` is mixed onto all pool pages.

For users the conference management system should be easy to reuse and customize, thus, we have implemented a generic page template class for different conference management systems. The user only has to edit information in the corresponding XML file to change layout, background color, and logos. Such style changes are sufficient for presenting many conference. For more sophisticated behavioral changes, a new style class has to be implemented.

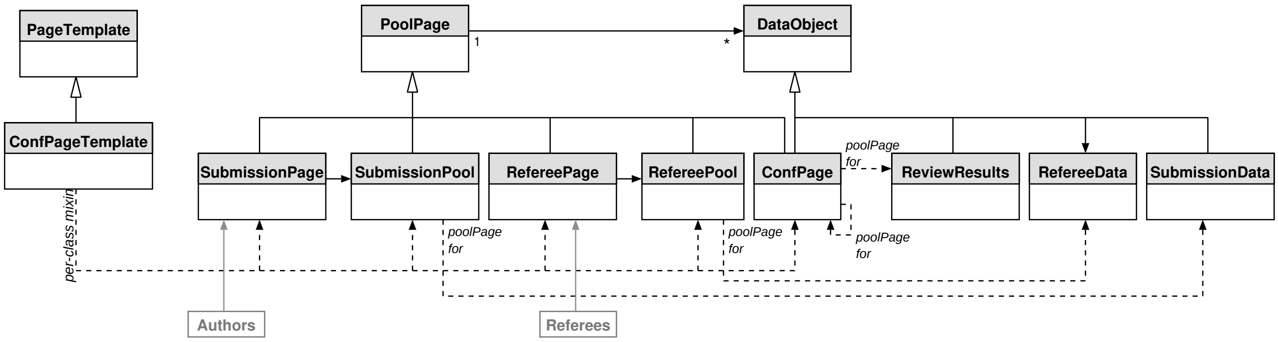

Figure 3: Conference Management System: Pool Pages and Data Objects

Each pool page has a set of different views onto its data objects. In general, each HTML view is encapsulated in a method of the pool page. E.g. the `SubmissionPool` has method for listing the papers, editing the papers, deleting the papers, acknowledgement of receipt via email, and for sending the results of the review process. If necessary, the name of the targeted data object is given as an argument to these method, so that the pool page can retrieve the data from it or manipulate it accordingly.

All pool pages are implemented in a simple similar fashion. Each pool page inherits from the general `WebAgent` class of the framework and have the generic conference page template as a class mixin. Therefore, all methods defined on these two classes are accessible for instances of the pool page:

```
Class SubmissionPool -superclass WebAgent \
  -instmixin ConfPageTemplate
```

In its constructor the pool first retrieves the information in XML file, as customized by the user for the concrete conference. Then the persistent data is automatically extracted and the data objects, defined so-far persistently, are re-instantiated. For each possible action on the data objects, we export one method. Let us consider the method for editing papers as an example:

```
SubmissionPool instproc editPaper {paper} {
  set formData [[self] getFormData]
  if {[llength $formData] == 0} {
    return [[self] formPage abstractForm \
      "Edit Paper" [$paper getPersistentData] \
      [[self] selfAction "[self proc] $paper"] \
      [list "Back to submission page" \
                   "[[self] selfAction listPaper]"]]
  } else {
    [self] instvar submissions place
    foreach a [[self] getFormData] {
       ### store form data and save uploaded file
       ...
    }
    return [[self] listPaper]
  }
}
```

The method `editPaper` simply consists of two parts. If no form data is given, we have to return the form for editing papers as an HTML page. It is simply constructed by the predefined `formPage` method. If form data was submitted, the user has filled in the form and we have to store the results persistently (and store the uploaded file). Afterwards we return the referring page (which is `listPaper` as a result).

The submission pool page is access protected via HTTP basic authentication (in ACTIWEB digest authentication can be used as well, if necessary). Thus it can only be access by the conference chair. The `SubmissionPage` is used as a not access protected view on the `SubmissionPool`. It offers only the abstract and paper submission pages visible to the authors.

Similarly the referee pool associates all referee data objects. It can only be accessed by the conference chair. The referee pages are access controlled views, which allow the referees to download the papers and submit their reviews. Each referee is automatically informed with an email about her/his page and ID/password. The conference page class is a pool for review results, but it is also a data object. Since there is only one conference page with unique data in the application, it makes no sense to divide the conference page into pool and data object. Therefore, the conference page is a data object as well.

All forms and data objects are given in XML-based page templates which can easily be customized for different conferences. Only a few values are required definitely by the application at runtime and the application raises an error at the startup time, if they are missing. For instance, each review result must have at least a default value and one mark for the default value. How many other marks are there and how they are labeled can be changed entirely in the XML file. Another part of the review form are comments of the reviewers: some conferences require merely comments to the author, some require a comment for each review element. Again, the necessary text areas can simply be composed by the conference chair in the XML-based template.

8. CONCLUSION

In this paper we have presented a concise and simple architecture that deals with many problems which we can identify in today's interactive web applications. The approach bases on dynamic generation of content from page templates, which conceptually integrate template-based and constructive ideas of web application development. Some prominent benefits are:

- *Multiple Channels*: The architecture contains a SERVICE ABSTRACTION LAYER [18] architecture. In the web server new channel adapters may be integrated to access the services in the pool pages. The services are accessed by the central MESSAGE REDIRECTOR [6] of the ACTIWEB framework. This indirection, for instance, allows us to export only certain methods of certain objects. And it allows for conveniently implementing authentication of certain web objects.

- *Multiple Representations*: The abstraction into user interface builders gives us the opportunity to rapidly support multiple different user interfaces. Only the specifics of each user interface have to be programmed by hand.

- *Separation of Concerns*: Abstractions for specifics of content, styles, and channels are provided. They can be given in declarative and behavioral fashion. There is an architectural model for integrating both fashions. However, the concerns can be cleanly separated through high-level object-oriented abstractions in the class-based page template fragments and through different XML elements in the XML-based page template fragments.

- *Runtime Generation*: All generational aspects are handled at runtime. Therefore, the important issues of introducing changes into a running program is supported by the architecture almost without additional programming effort.

- *Legacy Integration & Wrapping*: The concept presented is especially well-suited for integration and wrapping of legacy applications into the web. Legacy services can be integrated as pool pages. Old access forms/channels can still be supported. Or they can even be migrated to the architecture presented by supporting a user interface builder and a channel adapter for the old channel and representation.

- *End-User Changes*: End-users are able, possibly with tool support, to configure and extend the application directly. Such tools can introspect the parsed XML files or the current page template mixins. In XOTCL such introspection options are provided by the language. Thus, end-user tools can quickly be written (with very similar code as in the pool pages/XML configurator). However, the approach is still comfortably programmable by expert programmers as well.

- *Language Support for High-Level Programming*: Many tasks fulfilled by our architecture are directly language supported in XOTCL. For instance, language dynamics and introspection allow for rapid changeability, new components can be dynamically loaded into the system, dynamic slots for refining data objects on the fly are supported, mixins allow for combining of page templates with given pool page classes, the integrate XML parser component lets us treat XML text as a dynamic object aggregation tree, and persistence services let us make data objects transparently persistent. It requires a certain programming effort to support those language elements in many other languages. However, with patterns, such as PROTOTYPE-BASED OBJECT SYSTEM [14], MESSAGE REDIRECTOR [6], OBJECT SYSTEM LAYER [5], or Type Object [8], such tasks can usually be programmed in a few hundred lines of code in almost any procedural or object-oriented language. Thus, our architecture benefits from the used language constructs, but it does not rely on their presence.

Of course, we can also identify a few liabilities for the given architecture:

- *Performance*: Some more low-level approaches may offer a better performance, since they avoid the indirections involved in the architecture.

- *Complexity*: The complexity of the approach is higher then simple architectures, such as template-based approaches or CGI scripts. Thus, resulting applications may be more complex for very small tasks. For larger tasks, the complexity of the simpler models usually grows exponentially higher, e.g. because of cut-and-paste code and missing integration models. Most parts of the archtiecture presented have to be build by hand in such approaches.

- *Robustness*: The approach proposes a single or a few running processes. Measures for reaching higher robustness and failure safety currently have to implemented by hand.

- *Common User Interface Denominator*: Our approach allows for using different user interface/channel access models. This makes especially sense for bringing a legacy application to the web without changing its old interface. But, if the user interface builders are used exclusively, the user interfaces are reduced to the common denominator defined in the abstract user interface builder. Of course, certain user interface builders may also ignore certain formatting instructions, say, like a WML user interface builder that cannot fully support the HTML subset.

The architecture is implemented in XOTCL with ACTIWEB. Both are available from `http://www.xotcl.org`.

9. REFERENCES

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

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

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

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

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

[6] M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In *Proceeding of EuroPlop 2001*, Irsee, Germany, July 2001.

[7] P. Greenspun and E. Andersson. Using the ArsDigita community system. *ArsDigita Systems Journal*, Feb 1999.

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

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

[10] G. Neumann and U. Zdun. High-level design and architecture of an http-based infrastructure for web applications. *World Wide Web Journal*, 3(1), 2000.

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

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

[13] G. Neumann and U. Zdun. Pattern-based design and implementation of a XML and RDF parser/interpreter. Submitted for publication, 2001.

[14] J. Noble. Prototype-based object system. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 2000.

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

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

[17] A. Vckovski. Tcl Web. In *Proceedings of 2nd European Tcl User Meeting*, Hamburg, Germany, June 2001.

[18] O. Vogel. Service abstraction layer. In *Proceeding of EuroPlop 2001*, Irsee, Germany, July 2001.

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