Abstract
Recently, many web application frameworks have been developed to simplify the creation of new web sites. However, in the light of well-established web design methodologies such as RMM or OOHDM, these methodologies often fail to close the gap between design and implementation. In this paper, we analyze the elements of several design methodologies in order to classify and group them. We then introduce our Context-Based Navigation web design framework (COBANA) and show how its different parts rely the various concepts of the design methodologies, therefore allowing a 1-to-1 translation from design to implementation. Finally, we compare our framework with related work.

Keywords: J2EE, Web Application Framework, Web Design, Hypertext Navigation, XML, XSLT

1. Web Site Design State of the Art

Design methodologies for hypertext documents and web sites are well-established and numerous (see [1] or [2] for an overview). The Relationship Management Methodology (RMM) ([3], [4]) is a typical example. There are even some object-oriented approaches such as WebComposition [5], the well-known Object-Oriented Hypermedia Design Method (OOHDM) ([6], [7], [8]), and the Scenario-based Object-oriented Hypermedia Design Methodology (SOHDM) [9] which were 'artificial' at the time, since the HTML-based web - as an "information medium rather than an application platform" [2] - scarcely fits the paradigms of object-oriented programming. With the advent of the Java 2 Enterprise Edition (J2EE) specification [10], however, these methodologies are infused with new life, since it provides object-oriented implementation concepts for the development of web-based systems.

While all these approaches have their differences, there are several concepts which appear in most of them. Instead of limiting ourselves to one of these methods, we are going to take a look at their central elements, associate them with either model, view, or controller aspects, and use them as a basis for our web application framework and translation rules later on.

1.1. Data Model

The essence of a web site is the information to be shown. All methodologies considered in this paper offer some way of modeling information. Common approaches to this are either Entity-Relationship or object-oriented diagrams. Data is divided into what is called data objects, whose nature depends on the method used for problem analysis. Alternative names include entities [3] or conceptual objects [6]. The links between the objects or entities already hint at a navigational structure for the site.

To allow fine-grained access to this data, slices [3] are introduced. A slice is a subset of the information found in a single data object. More advanced versions (m-slices in [4] or the components in [5]) may contain other slices. The nodes in [6] are a similar concept. In some ways the relationship between slices and data objects is similar to that between a database and its views [6].

1.2. View

The issue of how data should be presented is part of the view. The slices, as mentioned in 1.1, could be interpreted as being view-level concepts (as done in [3], for example), or the view responsibility can be separated from the aspect of data selection by providing each slice with an associated interface object [6]. The interface object is responsible for the presentation of the data provided by the slice, which is enriched with navigational information relevant for the slice. The presentation encompasses layout and formatting. An interface object can contain lower-level interface objects as components. The Web Site Design Method (WSDM) [11] has a similar concept called "information components".
Some approaches ([5], [12]) use prototype-instance models to allow reuse of created interfaces. This is achieved by variables in the interface objects, as well as introducing new presentation rules. In fact, we have a mixture between instantiation and inheritance. In [5], the prototype-instance model is realized as collection of "HTML templates".

1.3. Controller

This section describes how navigation is handled. The most basic choice for navigation are simple hyperlinks, a well-known concept supported by all design methodologies. While every web application needs links for navigation, the use of 'explicit links' should be minimized. An explicit link is either a static connection between two slices or a reference to an external URL. With an increasing number of links, the design becomes more complex and harder to implement. Although this link type is necessary, especially for links referring to other web sites, it should not be the "main structuring primitive" [6].

Far superior navigation, which better reflects the semantic structure of the site, can be achieved with abstract navigational contexts [6]. Basically, these are sets of data objects that are chosen due to specific properties. A possible example is a context of all people whose name starts with an 'A'. All objects within the context should be reachable via an index or other links. Additionally, it should be possible to associate a slice with more than one context (as described by the 'Nodes in Context' pattern [7]), and the option to allow a different look for each occurrence should be available.

Data objects can either be grouped by properties, by explicit enumeration, or by class type. Contexts may also be used to represent 1-to-n links. A more exotic variant are dynamic contexts based on session information (in effect a history mechanism).

Another name for navigational contexts is views [9]. "Dialogs" as described in [13] can be seen as a subtype of contexts. They are collections of pages that have to be filled out in a certain order, in other words, a complex form.

The contexts are actually more of an organizational construct, albeit with strong orientation towards navigation. The actual movement between the pages is achieved by associated access structures [6]. Examples of these are indices (i.e. collections of links), guided tours (i.e. pages that can be viewed in a certain order) or more specialized forms such as trees. Access structures can be used in conjunction with contexts.

Similar concepts are access structure nodes (ASNs) [9] and "navigational components" [11].

Access structures and abstract navigational contexts allow the introduction of a hierarchical structure into the web site. This is achieved through the nesting of indexed contexts. Important high-level topical sub-divisions (also called landmarks [7]) should be reachable from everywhere on the web site. For example, an internet shop selling books, CDs, and DVDs should allow the user to reach each of the categories' starting page no matter which part of the site he is currently viewing.

It is also important to be able to realize active references [7], i.e. web pages should include links to all other relevant pages, such as indices higher in the hierarchy, siblings and/or child nodes. A non-web example for this navigational structure is the Windows Explorer for the file system. To implement Active References and 'Nodes in Context' is quite challenging as described in 5.3.

To sum it up, web pages are organized through a hierarchy of navigational contexts and can be accessed via the access structure associated with each context. However, not all contexts' access structures are visible to the user at once, a selection is made according to the user's current position within the context hierarchy. This is called active reference.

1.4. Other Elements

Perspectives are an interesting concept advocated by WSDM [11]. Rather than having one object model, there is a separate one for each type of user, e.g. on a corporate site customers and employees will see different (but overlapping) information, presented in a different way, with different navigation. The model is then adapted to sub groups, i.e. new customers will have a slightly different web site than established ones but with enough similarities to derive both cases from a single model. Perspectives encompass all three layers of design, described in sections 1.1 through 1.3, and while their functionality is deemed useful (see [6], [11]), they are difficult to implement efficiently. Internationalization can be seen as a related problem.

When designing a web site, a separation between two different kinds of pages becomes apparent. There are unique pages and groups of similar pages [4], for example descriptions of employees. The latter often involve access to a database and allow the displaying of different contents of the same category in the same layout, the former are often navigational constructs or general information.

Finally, Zhao et al. [14] propose the introduction of elements beyond the normal web paradigm, e.g. dynamic
updates when data changes on the server, i.e. a push concept. In this paper, however, we will stick to the accepted client-side standards with pure HTML.

2. J2EE and the Web

As mentioned before, some of the web design methodologies are based on object-oriented principles. With the advent of component-based systems such as Enterprise JavaBeans (EJB) [15], which are part of Java's Enterprise Edition (J2EE) [10], it is now possible to seamlessly transform object-oriented web designs into web applications. J2EE, which also encompasses servlets, supports complex distributed enterprise applications and is therefore an ideal basis for our intended framework.

3. The COBANA Framework

In this section we explain the basic architecture of the Context-Based Navigation framework (COBANA). The COBANA framework is designed to maximize reuse of standard components, a practice encouraged by some of the design methods (e.g. [7]). This is achieved by using generic classes, which can be used to realize the elements of the various design methods within an J2EE context (see section 5).

3.1. The Core Concept

COBANA is based on J2EE. The framework employs a servlet for communication with the user, the Extensible Markup Language (XML) [16] for the internal representation of data, and Extensible Stylesheet Language Transformation (XSLT) [17] to transform this XML data into the desired output format (e.g. HTML) intended for the end user. The basic idea is to have different views, defined via a configuration file listing them all, whose content is defined in either entity beans or by static XML files, that are called by the central controller according to the user input.

Figure 1 illustrates the simplified concept of the framework. The web server passes user requests to a servlet called Controller, which then looks up the View object required to handle the request and calls the view's getPage() method. In this method the chosen view calls its overwritten method getXML(), transforms the result using XSLT and returns the HTML code. There are some additional complications which will be explained when introducing subviews and context views.

Which views are available to the application is specified in an XML configuration file. This file contains definitions of all views, including information on which class to use to represent a view, view parameters, and navigation information.

3.2. View and its Subclasses

As can be seen in figure 1, View is an abstract class. It provides a method to transform the XML content of a view (wherever it comes from) to HTML applying a view-specific XSLT rule set. It can also contain subviews whose getPage() results are used as HTML fragments for the page to be returned. The getNavInfo() method provides navigational information which can be used by the XSLT rules to create a "navigation bar" for the web site. Figure 2 shows the process of requesting a view that contains a subview.

Three of View's subclasses are shown in the figure 1, although a web site will most likely contain several custom classes as well, for example subclasses of ContextView for database access.

StaticView is a static page (or part of a page) that does not access the database. Instead, its getXML() method reads an XML file from a web server. getPage() remains unchanged, handling the transformation to HTML.

A more complex example is the ContextView which
is a collection of other views, called child views. As opposed to subviews, these are not shown simultaneously. Rather, only one is visible at a time. There are several ways to access the different children. A ContextView has a list mode. In this mode it shows an index of all child views it contains. Choosing one of the children will cause the ContextView to display it. There is also a guided tour mode which automatically links the pages in a round-robin fashion.

Nesting several contexts can lead to a treelike structure for the web site. ContextViews are responsible for the navigation on a site. While the concept seems straightforward, the implementation details for this page are complicated since the requests passed on to the child views have to be altered and navigational info has to be generated with a complex recursive algorithm.

4. Brief Introduction to the Example

We are going to explain the features of COBANA by using a small example from a university background: master theses. Each student has to write a master thesis, and a professor supervises zero or more students during this process. Figure 3 shows a simple class diagram modeling this relationship. We are now going to design a web site that will allow the university administration to keep track of this relationship.

5. Realizing the Design Elements

So far, we have analyzed several design methodologies, extracted and classified their elements, and described the basic concept of COBANA. Using aspects of the example, we are now going to explain how one can implement web site designs in an easy way, using our framework.

5.1. Content Translation Rules

Many approaches start with a data-centric design, which produces either an ER or a class diagram of some sort. Even if the design does not start with the data [11] this diagram can be derived sooner or later.

Data objects for groups of similar pages can be implemented as Entity Beans coupled to a custom view per slice. In our example case, we therefore implement two entity beans - ProfBean and StudentBean. We also have to implement custom view classes (StudentView, ProfView) and extract their XML source data from the entity beans' attribute values, possibly via a session bean that returns entity info as XML strings. Figure 6 shows a possible custom view architecture. Data objects for unique pages are
5.2. View Translation Rules

StaticViews implemented as XML files in conjunction with views with XSLT rule sets.

COBANA. In fact, the XSLT file associated with a view constructed in two ways. Commonly, the XSLT script is entirely responsible for the presentation of its data and decides which parts of the data are to be shown and which not. Custom views for bean access, however, can also be designed for this task. Since views can contain subviews (as can be seen by the self-reference in the class diagram of figure 1), it is now possible to assemble high-level slices from different sub-slices.

The prototype-instance-model can be realized in two ways. On the one hand, XSLT scripts can reuse rules from other scripts by importing them. On the other hand, subviews may be shared by two instances and be interpreted as the prototype for both.

5.3. Controller Translation Rules

Navigation is the central aspect of a good web design, and therefore it holds a central place in COBANA. In fact we try to avoid explicit links as much as possible.

Abstract navigational contexts find their direct equivalent in the ContextView class. Several views can be grouped in a context and be accessed via all possible access structures. These are all integrated into the ContextView. Nesting of several instances allows a developer to create directed-graph hierarchical structures and due to the look up of views via a central table, it is even possible to have a view contained within several separate contexts. The design shown in figure 4 uses two nested ContextViews, START and THESIS, defined in the configuration file (figure 5).

The active reference concept is also supported by the ContextView. Using a recursive algorithm (in the overwritten getNavInfo() method), a subtree is inserted into the XML data used for transformation. This tree contains all nodes of the parent levels of the view currently displayed, i.e. the tree is hidden except for the currently active branch. This information can be used to create a navigation bar with a simple XSLT rule.

Hyperlinks can be generated automatically. Therefore, a view that makes use of explicit links must include a subview storing pure link information. Placeholders within the view reference these links via unique Ids. With the right XSLT rule sets, these placeholders are then transformed to anchor tags. This makes the maintenance of explicit links easier, since one subview contains all the URLs and can be used by multiple views.

The subview concept also supports the implementation of landmarks. A subview with landmark links embedded into the main context page will always be visible.

![Figure 5: The Configuration File](image)

![Figure 6: Example Bean View Architecture](image)
6. Related Work

There is a lot of research done in the field of web application frameworks and many interesting projects are currently in progress.

Struts [18] is probably the most popular framework but it uses JSPs which have their own problems as described in [19]. Also, in our and Kamm's and Klein's opinion [13], it lacks support for a segregated navigation. They also suggest an extension to Struts which is supposed to remedy this fault. Unlike COBANA, however, it does not take into account the various design methodologies for web applications. StrutsCX [20] lacks JSPs but still offers no good navigation. The core Controller design of COBANA was inspired by (but is not based on) these frameworks (i.e. Struts and StrutsCX).

Chu et al. [21] have implemented a system similar to ours - database information being transformed into XML and then into HTML (or the Wireless Application Protocol (WAP) as well in their case) - but concentrate on technology rather than design and do not seem to use XSLT.

Barta and Schranz have developed a content management system called JESSICA [22], which is based on an object-oriented web modeling language used to produce HTML code via a compiler. While this approach is interesting and provides good support for multilingual sites, we believe that there is insufficient separation of content and presentation (as can be seen in their code examples). Also, while database access is possible (apparently through scripts), no special steps are taken to support this through the language itself.

Other related projects are Cocoon [23] which focuses primarily on the XSLT transformation of stored XML documents, however, and many others as listed by [24].

7. Future Work

The COBANA framework is currently in the prototype stage and there are many additional ideas we plan to incorporate. First of all we plan to intensively test the framework to identify weaknesses and optimize the processes involved. One planned improvement is the introduction of standard XSLT rule sets which can be imported to cover certain complex cases, e.g. navigation bars or guided tours.

Additional future work includes:

- Providing support for full web applications - currently user input is possible only via custom views, the classes provided only allow for "kiosk type web site[s]" [3]. Possibly, some kind of generic form view and support for user sessions will be implemented.
- Implementing modules for security concerns and different user roles.
- Support for other technologies such as Web Services, Common Object Request Broker Architecture (CORBA), Juxtapose (JXTA), WAP or Remote Method Evocation (RMI).
- Designing a tool for automatic code generation that creates web applications using this framework.

8. Conclusion and Evaluation

In this paper we have presented a web application framework (COBANA) that has been designed in order to allow easy implementation of web site designs generated with one of the common design methodologies. However, the framework does not depend on any specific methodology.

While we did not describe the technical details due to space limitations, several interesting problems that were solved should at least be mentioned:

- The XML data of a view can contain many fragments from subviews. This hierarchical structure causes multiple XSLT transformations on the subview content, leading to several problems, including prematurely 'unescape' XML entities and the handling of empty HTML tags. As a consequence, strict rules for both the structure of the XML fragments as well as the XSLT transformation parameters had to be devised.
- Passing navigational information to the stylesheet required an adequate XML structure that has to be appended to the getPage() information on the highest level only.

Additionally, many common "web engineering problems" as described in [5] will be avoided when using COBANA:

The navigational structure is easily recognizable in the file that defines the various views. Thus, the organization is evident without having to analyze multiple complex files (such as HTML pages) and changes can be made quickly. Since navigation is an important part of a web application, it deserves special and separate consideration.

Due to the subviews and the XSLT stylesheets, the problem of massive replication of "design artifacts" is eliminated. For example, page layout and certain elements such as headers, footers, and navigation bars have to be defined only once and can be reused.
Both advantages described above vastly simplify maintenance of the web site. Content can be changed via a special client application (in the case of entity beans) or by editing XML files (in the case of StaticViews). Both ways do not require any special computer skills or HTML knowledge. More complex tasks such as altering the site structure become routine, as relevant information exists only once in a single file, avoiding redundancy. Thus, even navigational context information is human-readable. Changing the looks (by changing the XSLT) is a bit more complex but still more efficient than with pure HTML.

9. References


