Site-Wide Templates for Internet Sites

by

Michael Bryzek

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering
In Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2000
[June 2000]

Author [illegible]
Department of
Electrical Engineering and Computer Science
May 2000

Certified by [illegible]
Harold Abelson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Site-Wide Templates for Internet Sites

by
Michael Bryzek
Submitted to the Department of
Electrical Engineering and Computer Science
on May 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering
In Electrical Engineering and Computer Science

Abstract

This paper presents the design and implementation of a system to maintain multiple site-wide templates for one website. The idea is to separate the core contents of a web page from the graphical elements making up the page header and footer. Our goal is to non-intrusively allow both programmers and graphics designers to work mostly independently, decreasing development time while increasing the reliability and easing maintenance of the templates.

SWTM accomplishes most of these goals through a simple programmer's API, an object-based approach to building templates, and a stored repository of all objects previously created. Thus, even when the separation of tasks between programmer and graphics designer is not possible, the programmer need only solve each problem once, allowing the graphics designer to reuse initial work in future templates.

Thesis Supervisor. Harold Abelson, Professor of Computer Science and Engineering

Acknowledgements

I would like to first thank those people responsible for making this project possible, including Professor Harold Abelson for supervising the work, Atisaya Vimuktanon, Rip Taggart, and Wendy Simmons for creating the initial need for such a system at GuideStar.org, and ArsDigita Corporation for providing both hardware and software resources to be able to implement the results. I would also like to thank Randy Graebner and Aileen Tang who were concurrently working on an online education network; they offered their system as a test bed for this project. I also thank Tracy Adams who helped with the design of a prototype system in 1999.

I also would like to thank Rhonda Johnson and Leo Leung at the United Way of Massachusetts Bay who provided many suggestions from the point of view of non-programmers. I also owe many thanks to Brian Reid who helped me locate notes from his PhD thesis on Scribe, written before the Internet made papers so easily accessible, and Jacek Siembab, who helped edit the final draft of this paper.

And of course, I'd like to thank my parents, brother, and sister for supporting and encouraging me to pursue an advanced degree

Finally, this thesis is dedicated to my fiancée, Lisa Marchand who managed to plan most of our upcoming wedding to allow me to spend more time on this project. She also provided constant support, encouragement, and suggestions to help this project become successful.

Table of Contents

1 Introduction

One of the ongoing frustrations in website design today, circa February 2000, is the lack of separation between programmers and graphics designers. Many individual pages on the web include a disproportionate amount of graphical context that is difficult to maintain or to change and usually requires the programmer's assistance. A more ideal development environment would allow the programmers and graphics designers to work independently. The separation of tasks between programming and layout would lead to a more maintainable code base and a shorter development time.

Many people have attempted to solve the dependency between programmers and graphics designers. However, today's solutions often go too far in solving the problem for either the programmer or the graphics designer. This paper presents a distinct type of templating system that is a compromise in convenience between the programmer and designer, allowing both to work productively with minimal changes to either's usual working environment.

The big picture is to introduce a simple layer of software between the programmer and graphics designers. We call this layer the site-wide template manager (SWTM). The programmer's API will consist of a very small set of instructions that replaces the programmers original API to setup each individual web page. Graphics designers will receive a new process by which they incorporate their graphical changes to the site. SWTM then can be modified, as needed, mostly independent of the work of the programmer and graphics designer.

In real life, this system has many applications – any database-backed website that needs, or may need to, change the templates used on the site across a large number of pages will benefit from SWTM. In particular, we will focus on the following implementations to evaluate the success of SWTM:

- A university online education system: The MIT Sloan school is planning to adopt an online course system to manage its courses.[1] Each course will need its own look-and-feel.

- VolunteerSolutions.org: This online service matches volunteers to opportunities with nonprofit agencies. The service is available in many metropolitan regions and on independent university campuses, each with its own templating needs.

All of these examples will be implemented using the ArsDigita Community System version 3.2 (ACS) and the ArsDigita Architecture: Oracle RDBMS → AolServer → TCL scripting language, but the concepts presented are applicable to any development environment.

1.1 On the personal side...

The author's primary motivation in developing the site-wide template system comes from preliminary work in building a rudimentary template system for GuideStar.org. This online service needed to template its search engine and corresponding drill-down pages for various partners[2]. As GuideStar.org continued to add partners with various templating needs, a system to manage all of the templates became necessary. However, at the time (circa October 1999), no system existed that could be easily incorporated into the site.

[1] The online education system is being developed concurrently by Randy Graebner and Aileen Tang at MIT.
[2] See, for example, http://www.guidestar.org/search and http://www.guidestar.org/aol/search

1.2 Motivation

After creating the initial version of EMACS, Richard Stallman wrote:

> Extensibility means that the user can add new editing commands or change old ones to fit his editing needs, while he is editing. EMACS is written in a modular fashion, composed of many separate and independent functions. The user extends EMACS by adding or replacing functions, writing their definitions in the same language that was used to write the original EMACS system... this is the only method of extension which is practical in use.[3]

SWTM is an extensible software module that can be "plugged-in" to existing sites, that, with a basic API for the programmers, provides a highly extensible interface to control the look-and-feel for the entire website. Note that this interface specifically focuses on the look-and-feel of the site, and ignores all other aspects of a website that should themselves be extensible.[4]

1.3 Where SWTM fits in

SWTM is designed to manage templates for a site where:

1. The number of pages that need to use templates is large
2. The customer does not need full content-management capabilities
3. There is a potential to brand the entire site, or sections of it, where the branding would require an independent set of templates
4. The site is database-driven.

When the above conditions hold, SWTM will result in a website that accomplishes its goals while being easy to maintain.

1.4 High-Level System Design Overview

SWTM separates a document into three distinct pieces:

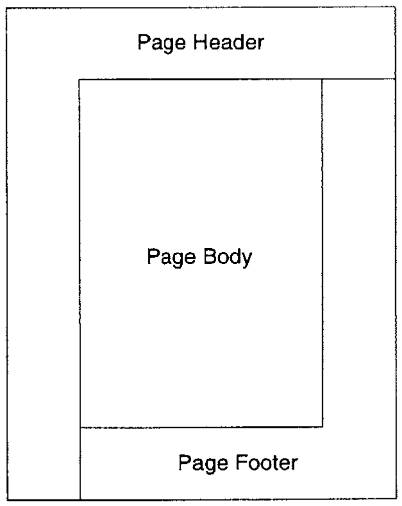


Figure 1: Division of a document

[3] Stallman, Richard, "EMACS, the Extensible, Customizable, Display Editor," Feb 11, 1998.
[4] The author believes the larger problem of designing a fully extensible website is too complex for the scope of this paper. Instead, we hope that by focusing our work initially on the design of an extensible templating system, we can motivate further research into systems designed to aid programmers in creating fully extensible websites.

Though this division of the page may seem obvious, its power is much more subtle. This division into three components is based on the way pages are designed using html. The only way to place elements in different locations on the page, e.g. a left menu bar, is to use html tables.[5] If we want to separate the contents of the page (the page body), from the graphics and navigation, we must separate the html needed to create the look and feel from that needed to display the page. Note that this division of a document will support almost all graphical layouts on the web today, from simple Yahoo-style menu bars to complex tab based systems with top, left, bottom, and right navigation.[6]

SWTM allows the graphics designer to specify the Page Header and Page Footer, leaving the programmer to design the page body. Note that designing the page body is largely dependant on functionality. That is, the page body is so intricately tied to the data model behind the entire web site that it is not feasible for a graphics designer to control the appearance of the page body. The basic idea here is that the programmer and graphics designer have different strengths:

- Programmers are great at extracting information from the database
- Graphics designers are great at creating graphics and navigation schemes.

As a real-life example of the need for such a system, one of the author's tasks while working on http://guidestar.org[7] was to support third-party sites who wanted to link to GuideStar's search engine. Figure 2 shows an example of one of the original GuideStar pages and Figure 3 shows a mockup of a desired third-party page (this one for America's Promise). The author chose to create the two pages entirely with Tcl procedures, and as a result, spent many hours changing style sheets, updating menu bars, and changing graphics.

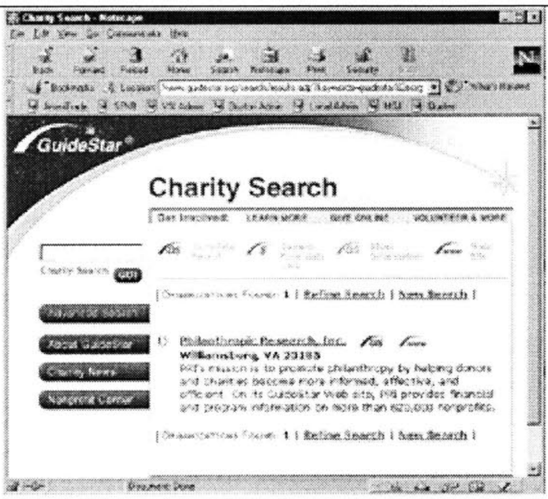


Figure 2: Original GuideStar Search Results Page

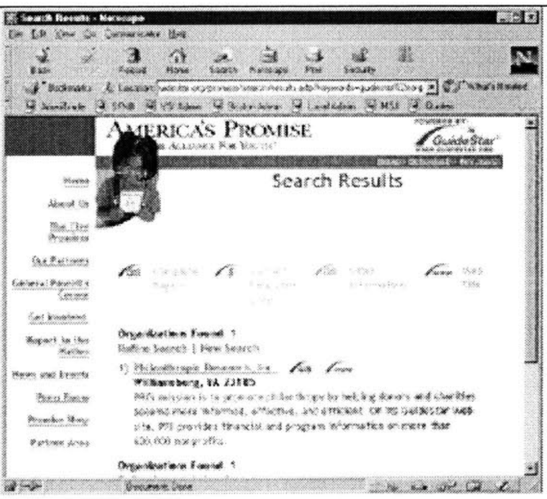


Figure 3: Mockup of Same Page for America's Promise

[5] Some people might say that you could use frames to accomplish the same affect. You probably could, but most people despise frames because they are impossible to bookmark. In any case, none of the top web sites uses frames, and we thus do not consider HTML frames as a viable alternative.
[6] Some examples of sites with complex navigation that still breaks down into a page header, page body, and page footer include http://mothernature.com, http://www.chowk.com.
[7] http://GuideStar.org is a client of ArsDigita that posts financial information about nonprofit organizations on the web with the goal of helping donors make better decisions as to where to donate their money.

1.5 Evaluating our success

The major goals of the site-wide template system include:

1. Separating the programmer from having to think about the graphical layout of each page in a way that makes it easy to maintain and change a set of templates.
2. Decreasing overall development time (or at least not increasing it!)
3. Ensuring relatively simple ports from legacy systems to ones that use SWTM.

1.5.1 Separation between programmer and graphics designer

The general concept of separating the programming of the web page's functionality from the layout of the page is the most challenging goal of this project. Many systems have been designed to try to create this separation as cleanly as possible, and all have failed (for more information, please refer to Section 2: Background).

Our goal here is to allow the programmer to work without having to think about the *general layout* of the page. That is, our goal is not to allow the programmer and graphics designer to work in complete isolation throughout the entire life of the website but rather to allow the programmer to work with the graphics designers only on the structure and flow of the content and functionality of each page. By doing so, we limit our focus to creating a system that can manage the graphics and navigation that surround the body of each page as shown in Figure 1.

We will measure our success with the following experiments:

- To what extent must the programmer be involved in creating templates?
- How long does it take to create an entirely new and different set of templates for use with the same content?
- How many things must be changed (e.g. number of lines of code, number of files, etc.) to make the new set of templates work properly?

1.5.2 Development time

Another of our primary goals is to ensure that we do not worsen the lives of either the programmer or the graphics designer through their use of SWTM. For SWTM to make sense as a tool, it must not increase the amount of time each party spends doing the same work they do today. Our hope is that this tool will actually decrease the time it takes to develop a page, but we'll settle for no change in the development time.

We will measure development time by simply measuring the amount of time it takes to add the look and feel to a web page using SWTM versus a traditional method of either cutting and pasting the html or modifying some procedure that returns the html. Note that we will run this experiment on a website that currently implements at least two different templates for a portion of its pages.

1.5.3 Ensuring simple upgrades/ports

Our final major goal is to ensure that it does not require too much programming time to install SWTM on an existing website. This is important as once site-wide templates are in use, we want them to be used on all pages, including those previously developed.

Porting a system is never easy, and we expect a significant time investment to install the template manager on top of an existing system. Our goal is to at least make the upgrade process simple and as error-free as possible.

To measure our success here, we will choose one section of an existing website built with ArsDigita's toolkit and compare the amount of time it takes to put that module under the control of SWTM versus a full content management system.

2 Background

Before designing any new system, it is helpful to look at what is currently available in the marketplace that is a possible replacement for the system we are trying to build. In the case of site-wide templates, we are interested in any tool or product that performs at least one of the following functions (Note that without loss of generality, we assume that graphics designers do not know anything about programming, but do have basic knowledge of html):

- **Markup languages:** Consists of a simple language designed to eliminate or hide the complexity of programming from graphics designers.
- **Reusable components:** Allows the definition of some sort of reusable block of text or code that can be modified by programmers and/or graphics designers. Once we have reusable components, we can easily design a system to implement top-down or bottom-up design of presentation templates.

In looking at examples that address some of the needs of SWTM, we hope to identify important concepts or tools to use in the design of our system.

2.1 Markup Languages

Tagging languages are designed, in general, to hide some complexity from the author of a document. The most common tagging languages are used to separate the presentation of text from the text itself.

2.1.1 HTML

> "HTML consists of standardized codes, or "tags", that are used to define the structure of information on a web page. HTML is used to prepare documents for the World Wide Web. A web page is a single unit of information, often called a document, that is available on the World Wide Web. HTML defines several aspects of a web page including heading levels, bold, italics, images, paragraph breaks and hypertext links to other resources."[8]

Basically, HTML separates the content of a web page from its presentation through a very basic set of tags. The simplicity of the language is largely responsible for the number of web pages that exist today. HTML is designed to represent only one web page at a time, and does not help us in defining site-wide templates.

2.1.2 ADP Templates

ADP templates are similar in concept to HTML with the major difference that tags can be defined on the fly and reused by other sites. If a group of users agrees to a particular ADP standard, this tagging language can be used to help that group of users share and present data on independent sites. The concept of extending the base language is very powerful and one that we include in the design of our system.

2.1.3 Scribe

Brian Reid developed Scribe (Document Specification Language and its Compiler) as one of the early experiments in automating the production of documents. Scribe was an example of a program designed around "the whole idea of creating documents at a higher level than the individual formatting commands."[9] In designing this system, Reid found that:

[8] Network Solutions, Inc., "What is HTML?", http://rrpac.upr.clu.edu:9090/~jcarroll/html/sld02.html
[9] Abelson, Hal, Email exchange with the author, Feb 14, 2000.

- The reality was that computer production of documents was not easy and the finished quality was often poor
- Formatting programs put too much control in the hands of the authors and did not separate the expertise of authoring text, designing formats, and imposing the formats onto the text.[10]

In summary, Reid tried to do too much with Scribe. The program excluded people, such as the formatting experts, from the process of creating a document. Without their expertise, it was nearly impossible to correctly format a document.

2.1.4 LaTeX

LaTeX is a markup language designed to separate the work of presenting a document from authoring its content. It is very similar in design to HTML but has a richer set of commands. LaTeX works very well for individual documents and users, but does not assist us in creating templates that can be applied and easily changed to hundreds or thousands of documents.

2.2 Reusable components

Many programs today already allow users to create site-wide templates. We look at both traditional, or single-user applications, and web applications, noting their strengths, weaknesses and the lessons learned from their design.

2.2.1 Desktop Word Processors

When opening a new document in a word processor such as Microsoft Word, users have the option to select a template in order to set the basic style sheet for the document. With non-web applications, such as a desktop word processor, the idea of reusable components is largely simplified because in most cases, the amount of customization that is needed and the number of users with disparate needs are small. Designing a website, however, is more akin to designing a suite of customizations that can be dynamically loaded as different users interact with the software.

2.2.2 Email Programs

Email programs allow users to define footers that are automatically attached to each email message sent. This saves the user from having to repeatedly enter the same information.

Email footers are another extremely simple example of a system of reusable components. Alice simply creates a file containing the signature she wants to repeat in every message and the software "cuts and pastes" that text into every mail message she composes.

2.2.3 Macros

Many desktop applications support macros that allow us to repeat commands infinitely without having to manually re-type the commands. Macros are similar to email footers in that a user saves a sequence of commands that can then be executed in batch. Once the macro is changed, it must be rerun on all of the data. There is no need for changes to a macro to automatically trigger the macro to be run again.

2.2.4 EMACS

EMACS is one of the great examples of a software program that can be infinitely customized to the needs of an individual user and is probably the nearest relative to our goal of creating site-wide templates. Some

[10] Reid, Brian, "20 Years of Abstract Markup. Any Progress?" Compaq Computer Corp, Nov 19, 1998, http://reid.org/~brian/markup98.html

of the lessons we can learn from the success of EMACS as related to reusable components for site-wide templates include:

- The ability "to accept and then execute new code while [EMACS] is running."[11] This feature allows users to not have to recompile the entire program to accept a new module.
- The use of global variables that are available after compilation. This allows modules/features added after compilation to reference the same variables.
- Dynamic binding that allows nested calls to see the last definition of any previously defined variable. This form of binding also allows us to define procedures that can accept a variable number of arguments as we really have no way of knowing in advance what the necessary arguments will be.[12]

The underlying theme to some of the lessons learned from EMACS is that the system must be designed from the start as an extensible system and must be modular so future needs can be cleanly integrated. We will have achieved our goal only when it is easy to extend and maintain our templating system.

2.2.5 Cascading Style Sheets (CSS)

Cascading Style Sheets are used to redefine properties associated with the look-and-feel of individual HTML elements. "Every element type as well as every occurrence of a specific element within that type can be declared a unique style, e.g. margins, positioning, color or size."[13] On the web, CSS is the only system that makes it easy to customize entire websites. CSS simply requires programmers to include a link to the style sheet in their pages and leaves the design of the style sheet to the graphics designers. Modifying the single style sheet can instantly change the display properties of every document in the website.

The major problem with CSS is that it is too limited in scope. CSS is designed to affect only the display properties of existing elements and does nothing to help us change the entire look and feel of a website and all of its documents. We expect that any templating system designed for the web should take full advantage of CSS rather than try to rebuild the functionality already provided.

2.2.6 Content Management Systems

Content Management Systems[14] are used to separate the management of content from programming tasks. These systems are usually complex and are designed with the customer in mind. One of the main features of a content management system is that a non-programmer can easily and remotely modify any piece of content on a website. Content managements systems are extremely powerful and work well when the client wants to control the way each piece of information appears on a website. In reality, the client is more concerned about the general look and feel of the site as a whole rather than changing a <BR> tag to a <P> tag. Because content management systems are so closely focused on the needs of the client, programmers often must change their entire style of programming to use a large, and sometimes complex, API. It is thus often easier to rewrite an existing system than to port it to a content management system, and, in many cases, content management systems are overkill.

[11] Stallman, Richard, "EMACS, the Extensible, Customizable, Display Editor," Feb 11, 1998, http://org.gnu.de/software/emacs/emacs-paper.html.
[12] Ibid.
[13] "CSS Frequently Asked Questions", http://www.hwg.org/resources/faqs/cssFAQ.html#css
[14] See, for example, http://www.vignette.com or http://arsdigita.com/doc/versioning.html

2.2.7 Page-by-page templates using ADP

Page-by-page templates using ADP are commonplace. The idea here is to ask the programmer to create several variables that hold the majority of the page contents, to design custom ADP tags on a per-client basis, and finally to offer clients the ability to rearrange the page contents using the predefined variables and simple ADP tags.[15] The variables are created in one file, and a separate file is created for each template used to present the file. This system works well when the number of pages that must use templates is small. If the entire site needs to use templates, this system will double the number of files that must be maintained. In addition, each time a new set of templates is designed, the number of files will increase linearly with respect to the original number of files in the system. This quickly results in a maintenance nightmare.

However, page-by-page templates offer two very real advantages:

1. Full control over the placement within the document of each piece of content
2. The ability to customize the layout of the page contents on a template-by-template basis.

These two features replicate some of the functionality of fully blown content management systems, and can be used, when necessary, to complement any templating system.

[15] See, for example, http://arsdigita.com/doc/style.html

3 Example Scenario

Before we begin discussing the design of the system, we outline the way in which programmers and graphics designers interact with the Site-Wide Template Manager (SWTM). The following example outlines the construction of a site-wide template for one of Volunteer Solutions' partners – Boston University (BU). VolunteerSolutions.org helps community service centers at universities put their databases of volunteers, nonprofit agencies, and opportunities on the web. As part of this service, VolunteerSolutions.org works with the university to integrate that university's look-and-feel.

3.1 General Process of Creating a Site-Wide Template

The overall process of creating site-wide templates can be summarized as:

1. Entering basic information into SWTM – Before any template is created, SWTM needs to know the name of the site-wide template and unique identifier for that template.

2. Identifying distinct nodes in the system – A node is a directory on the file system that has a unique look and feel. For example, on VolunteerSolutions.org, there are three primary nodes:
   a. `/general` – This directory contains files that contain general information including the site's privacy practices, legal policies, contact information, etc.
   b. `/agency` – This directory stores all files related to serving the needs of nonprofit agencies. Agencies use files within this node to post and update their information and to view statistics on their listings.
   c. `/volunteer` – This directory stores all files related to a volunteer browsing through the site, including the search engine, the volunteer's personal workspace, and agency listings for the volunteer to browse.

   Additional nodes might be needed for templates wishing to further customize the look and feel for:
   - Agencies who are logged in (`/agency/home`)
   - Agencies who are viewing general information (`/agency/general`)
   - Volunteers who are logged in (`/agency/home`)
   - Volunteers who are viewing general information (`/volunteer/general`)

3. Ensuring that all files in each node are setup to work with SWTM – Every file must use the SWTM API.

4. Creating a mockup html pages for each node – For each node that we identify as having a distinct look and feel, we need to create an html mockup from which to build the initial SWTM objects. Note that in most cases, the changes in look-and-feel between nodes will be minor – perhaps only a graphic, title, or menu bar needs to be changed. In this case, one core mockup and a detailed outline of what changes from node to node is sufficient.

5. Creating SWTM objects – We need to convert the mockups into SWTM objects. This process is straight forward and involves a lot of "cut-and-paste" work from the user's desktop to the web.

6. Assigning objects to nodes registered with the partner – The final step is to register objects with nodes for the given partner's template. Objects are registered through SWTM by assigning an object to a specific node, in one of two roles, either header or footer, and in a specified order.

The first step is trivial and amounts to filling out a web form – either the programmer or graphics designer can do this.

The programmer and graphics designer will generally work together to identify the nodes that need distinct templates. The programmer outlines the different site functions of the site, and the graphics designer decides whether the node associated with each function needs to have a custom template.

The programmer is solely responsible for the third step. Once the nodes have been identified, they must be under the control of SWTM. In some cases, the files in the node will need to be ported before they can be used.

The graphics designers are responsible for the last three steps, working mostly from their desktop computer and uploading their work when they reach milestones.

3.2 A Real-Life Example – Building the BU Site-Wide Template

To make our lives easier here, we introduce Paula, who is the lead programmer for VolunteerSolutions.org, and Gary, who is the graphics designer for the BU Community Service Center.

3.2.1 Step One – Gathering Basic Information

Paula and Gary agree to a unique identifier of "bu" for the BU site-wide template. Paula enters this information into SWTM, registering the `/bu` URL to serve pages using the BU templates.

3.2.2 Step Two – Identifying Nodes

Paula and Gary work together to identify the following nodes that need to have templates:

- `/agency`
- `/general`
- `/volunteer`
- `/volunteer/general`

They also decide that all four templates will be very similar, with only the menu bars and the top image depicting the name of the section changing.

3.2.3 Step Three – Ensuring that the files in each node use the SWTM API

Paula runs through the files in each node to be sure that all files have the appropriate SWTM calls. In our specific case, there is no work to do since all of the nodes were originally written to use the SWTM API. If there were some files that did not use the SWTM API, Paula would need to rewrite the appropriate portions of those files.

3.2.4 Step Four – Creating HTML Mockups

Gary begins by creating one core mockup that will be used by templates used in all the nodes. Before beginning, we identify the elements that will be changing between the nodes and how those changes will impact the template:

- The image depicting the name of the section will change – Changing the image will be up to SWTM and is straightforward, as simply changing a graphic does not impact the mockup html. Note that to successfully reuse one html template, all the images must be the same size.

- Each section will have its own menu bars – SWTM is designed to support multiple objects assigned to each node of a template. The core template thus cannot include any menu bars. We should create a separate mockup for each menu bar to later add to SWTM. In actuality, BU

mbryzek@alum.mit.edu

decided to use the standard VolunteerSolutions.org menu bars making the process of creating templates that much simpler.

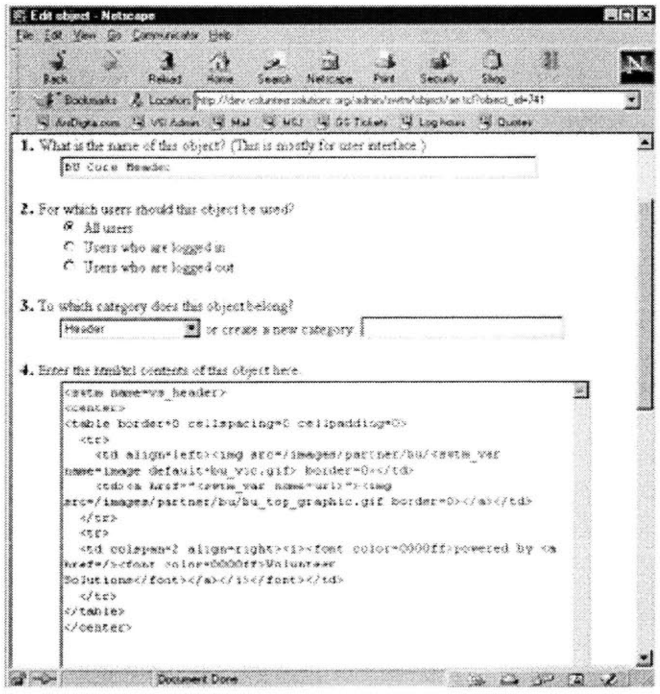
3.2.5 Step Five – Creating SWTM Objects

Once the mockups are done, Gary, in general will need to create one SWTM object for every mockup he has made. In our case, Gary has the following mockups:

- Core mockup to be used in all nodes
- The graphic to be used in each of the nodes

Note that since Gary has decided to use the VolunteerSolutions.org standard menu bars, those objects are already created and there is no need for a mockup.

Gary first creates the core object, which he calls "`BU Core Header`" by copying the html from his mockup into his web browser. (See Figure 4).


Edit object - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop
Bookmarks Location: http://dev.volunteersolutions.org/admin/swtm/object/ae.tcl?object_id=741
ArsDigita.com VS Admin Mail MSJ GS Tickets Log hours Quotes
Document Done

1. What is the name of this object? (This is mostly for user interface.)

   BU Core Header

2. For which users should this object be used?

   [✓ All users]
   Users who are logged in
   Users who are logged out

3. To which category does this object belong?

   Header or create a new category:

4. Enter the html/tcl contents of this object here.

```
<swtm name=vs_header>
<center>
<table border=0 cellspacing=0 cellpadding=0>
  <tr>
    <td align=left><img src=/images/partner/bu/<swtm_var
name=image default=bu_vic.gif> border=0></td>
    <td><a href="<swtm_var name=url>"><img
src=/images/partner/bu/bu_top_graphic.gif border=0></a></td>
  </tr>
  <tr>
  <td colspan=2 align=right><i><font color=0000ff>powered by <a
href=/><font color=0000ff>Volunteer
Solutions</font></a></i></font></td>
  </tr>
</table>
</center>
```

Figure 4: Creating the object "BU Core Header"

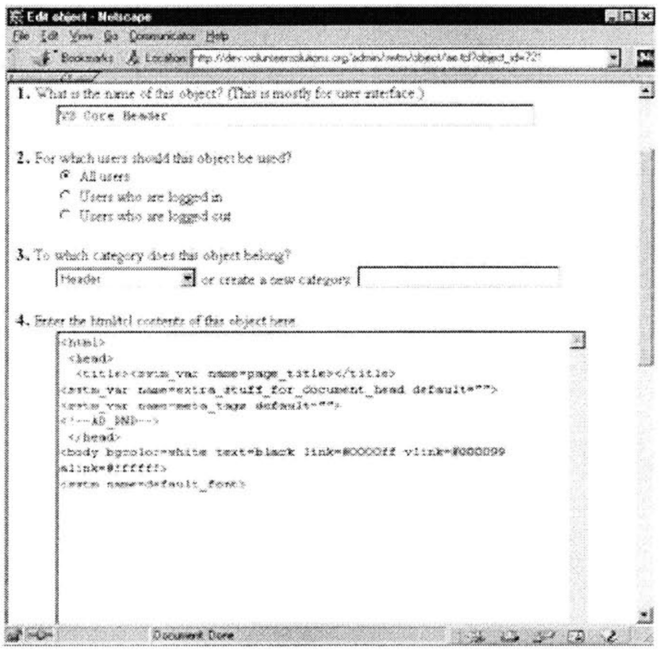
There are several key elements to note from the contents of this object:

- The first line, `<swtm name=vs_header>`, is a reference to a previously created object that inserts the basic `<html>` and `<head>` elements. Gary could have used his own html had he wanted, but it is easier to reuse objects. For reference, we include the definition of `vs_header` in Figure 5: Creating the object "vs_header".

- Gary replaces the name of the image to use for the graphic with this call:
  - `<swtm_var name=image default=bu_vic.gif>`

This allows us to later dynamically bind the value of the variable image to the appropriate image to use for a given section. If there is no binding for `image`, we use the default value.

- Though not captured in the screen shot, Gary names the object he creates "`bu_header.`" This name is how we are able to reuse objects and is unique across SWTM.



Edit object - Netscape
File Edit View Go Communicator Help
Bookmarks Location http://dev.volunteersolutions.org/admin/swtm/object/ae.tcl?object_id=22

1. What is the name of this object? (This is mostly for user interface.)
   VS Core Header

2. For which users should this object be used?
   [✓ All users]
   Users who are logged in
   Users who are logged out

3. To which category does this object belong?
   Header or create a new category

4. Enter the html/tcl contents of this object here

```
<html>
 <head>
  <title><swtm_var name=page_title></title>
<swtm_var name=extra_stuff_for_document_head default="">
<swtm_var name=meta_tags default="">
<!--AD_DND-->
 </head>
<body bgcolor=white text=black link=#0000ff vlink=#000099
alink=#ffffff>
<swtm name=default_font>
```

Figure 5: Creating the object "vs_header"

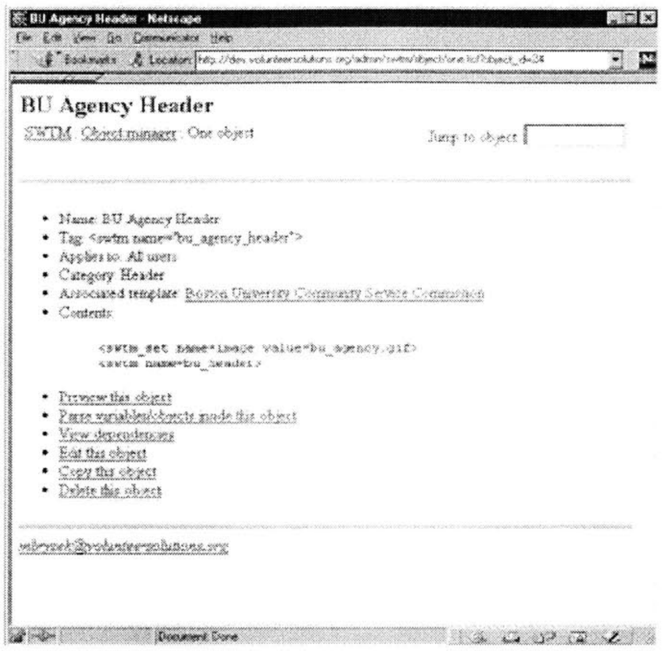
Next, Gary creates an object for the `/agency` node by dynamically binding the variable `image` to the name of the graphic he wants to use in this node, and including the `BU Core Header` object he just created. Figure 6 shows the code for this object, called "`BU Agency Header`." The interesting thing to note from this example is the way in which the Gary dynamically binds the `image` variables:

- `<swtm_set name=image value=bu_agency.gif>`

The `swtm_set` tag defines (or redefines!) the variable image so that the call to `BU Core Header` uses the right image.

Gary finishes by creating two additional objects:

1. `BU Volunteer Header` – Identical to BU Agency Header, with image bound to `bu_vic.gif`. This step ensures that even if Paula has previously bound the variable image to some value, `BU Core Header` will still find the right image.
2. `BU General Header` – Binds image to `bu_general.gif`


BU Agency Header

SWTM : Object manager : One object

Jump to object

- Name: BU Agency Header
- Tag: <swtm name="bu_agency_header">
- Applies to: All users
- Category: Header
- Associated template: Boston University Community Service Commission
- Contents:

```
<swtm_set name=image value=bu_agency.gif>
<swtm name=bu_header>
```

- Preview this object
- Parse variables/objects inside this object
- View dependencies
- Edit this object
- Copy this object
- Delete this object

mbryzek@volunteer-solutions.org

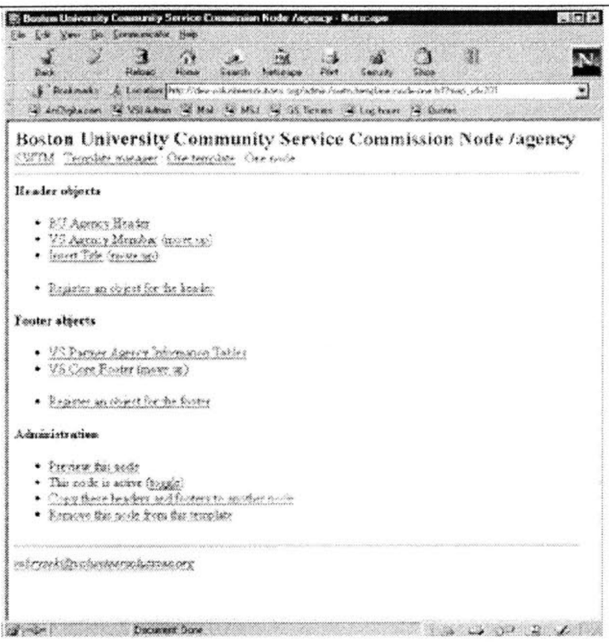
Figure 6: The definition of "BU Agency Header"

3.2.6 Step Six – Assigning Object to Nodes

The next step is for Gary to assign the objects he has created to each of the nodes through a series of simple web pages:

1. Gary goes to the SWTM template manager (e.g. `/admin/swtm/template/index.tcl`)
2. He selects the BU Template
3. He adds the four nodes previously selected to the BU Template
4. For each node, Gary simply selects the objects he wants to use for the header and footer.

Figure 7 shows the objects assigned to the `/agency` node and Figure 8 shows a preview of what the template for the node looks like. Note that in the preview we see that the variable `page_title` (coming from the call to `vs_header`) is undefined. This is a warning message that in this case is not a problem since Paula tells us that the `page_title` has been defined in every file, or left blank intentionally. When we turn off debugging mode, this type of error message would no longer appear.



Boston University Community Service Commission Node /agency

SWTM : Template manager : One template : One node

Header objects

- BU Agency Header
- VS Agency Member (move up)
- Insert Title (move up)

- Register an object for the header

Footer objects

- VS Partner Agency Information Tables
- VS Core Footer (move up)

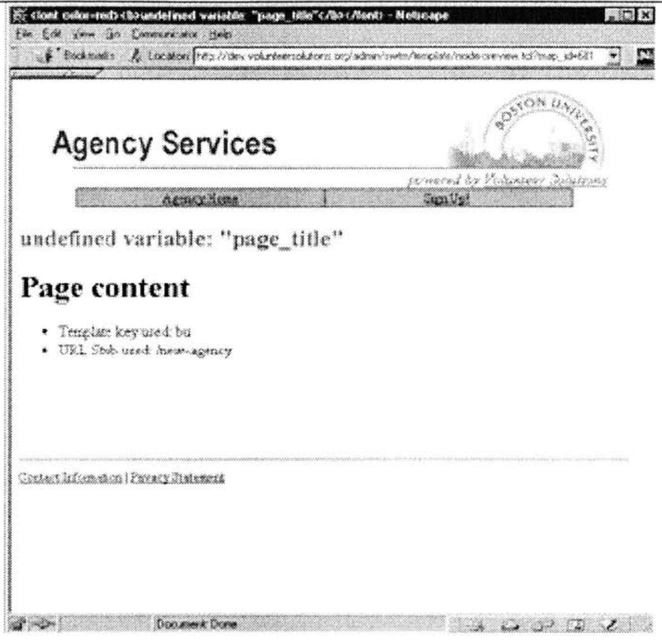
- Register an object for the footer

Administration

- Preview this node
- This node is active (toggle)
- Copy these headers and footers to another node
- Remove this node from this template

mbryzek@volunteersolutions.org

Figure 7: Objects assigned to the /agency node



Agency Services

powered by Volunteer Solutions

Agency Home | Sign Up!

undefined variable: "page_title"

Page content

- Template key used: bu
- URL Stub used: /new-agency

Contact Information | Privacy Statement

Figure 8: Preview of the /agency node

3.3 Summary

Most of the responsibility for creating site-wide templates lies with the graphics designer creating mockups for each node. Once the mockups are created, transferring to SWTM is relatively simple: cut-and-paste the html and use the three SWTM tags as necessary:

- `<swtm name=?>` to include another object
- `<swtm_set name=? value=?>` to dynamically bind a variable
- `<swtm_var name=?>` to retrieve the value of a variable

Over time, as the library of objects grows, the number of new objects that will need to be created for each new site-wide template will be reduced. We thus expect the first few templates to be the most painful.

4 Design of the Site-Wide Template Manager (SWTM)

This following section reiterates the primary design goals of SWTM and outlines the various considerations leading to the final system design.

4.1 Design Goals

We begin by outlining the major design goals for SWTM:

- Separate work of the programmers from that of the graphics designer
- Make it easy to create, maintain, and modify all of the different templates in use
- Decrease overall development time
- Allow for minimal work in upgrading legacy systems.

We also note that by successfully accomplishing our first two design goals, we will have accomplished our third design goal.

4.2 Design Considerations

4.2.1 Separating the work of the programmer and graphics designer

Our goal here is to allow the programmer and graphics designer to work as independently as possible. We start by identifying the tasks that each will be performing.

4.2.1.1 Identifying Job Tasks

We assume, without loss of generality, that the site owner has already specified the overall functionality of the website we are building. The programmer's job is to create all the pages in the system to accomplish the functions of the specification. For example, if we were building a site to provide advice to graduating law students on which law firm might be best for them, the programmer might build a search engine that lets students find a law firm by categorizing their interests along predefined criteria, such as the firm's size, location, and area of practice.[16]

The graphics designer, on the other hand, has two primary responsibilities. The first is to work with the programmer during the initial development of any page to ensure that the programmer's choice of user interface is appropriate. Note that this interaction between the two cannot and probably should not be replaced. It is important that the programmer receive enough feedback about each object in the system so that users will be able to successfully interact with the database. The second task of the graphics designer is to create the overall look-and-feel for the website which includes the general color scheme, graphics design, including icons and logos, and navigation schema.

In summary, we expect and encourage the programmer and graphics designer to work together to create an intuitive user interface within each page. However, we want to ensure that most of the work of the graphics designer remains separate from that of the programmer.

4.2.1.2 Achieving the Separation – Designing the SWTM API

To achieve this global separation, we introduce a very simple and generic API to specify the header and footer of a particular page:

- `swtm_header`: Inserts the page header

[16] See, for example, http://infirmation.com

- `swtm_footer`: Inserts the page footer

The graphics designer will be able to control what swtm_header returns based on the following state:

1. What page is currently being called? Note that we will identify a file by both its directory (or node) and its name. The node is used to retrieve graphical elements for the page and the filename is used to retrieve properties specific to that page.
2. What are the template properties of this page? (E.g., what is the page title, what section of the navigation bar do we highlight, etc.)
3. What site-wide template are we currently using? Each template will be assigned a unique identifier that is maintained either in the URL or in a cookie.

Everything else on the page belongs to the overall page body and is in the hands of the programmer.

Since we are using the TCL Scripting language as a test-bed for SWTM, we note that `swtm_header` does not take any arguments. Rather, we will create a separate mechanism for these procedures to look for variables defined in the calling environment (In a sense, we will implement a simpler, but sufficient, version of the dynamic binding of EMACS). Note that this is possible because Tcl gives us the ability to look into the running environment with commands like `info exists`.

To further enable the graphics designer to control the page headers and footers, we note that neither `swtm_header` nor `swtm_footer` actually generates html – rather, they figure out what procedures or objects must be loaded to create the html based on the system's current state.

4.2.1.3 Alternatives

In building a database driven site, there are a few alternatives to the proposed API to the SWTM. One option is to store the page body in one file, and create separate files containing the headers and footers for each template created. This option, though also simple, quickly leads to a maintenance nightmare as the number of files in the system will increase linearly with the number of templates created. We note that our SWTM API adds only a constant number of files to the system, if any at all.

Another option is to store each page and all the templates in the database, and at run-time, figure out which page in the database is being called and which template to use. This model is very attractive in that all of the content will reside in the database. However, unless we're going to offer content management type services, such as allowing a non-programmer to edit text in html pages through a web interface, storing each page in the database will make the development cycle longer, as the programmer will be forced to work with web forms, or some similar variant. Working inside the file system is simply faster and more convenient.

We do note, however, that Oracle plans to eventually integrate the file system with the database[17] so that users would interact with a file system as they do today, but the data would actually be stored in the database. When this solution becomes available, it will be a very powerful way to edit information that will then always be available.

4.2.2 Maintenance Component: Creating, Maintaining, and Modifying Templates

Creating, maintaining, and modifying templates is the most important, and most difficult, component of SWTM. Our goal is to allow non-programmers to create templates made up of smaller pieces that can be re-used by other templates, if necessary, can be easily modified through a web interface, and are powerful

[17] Dixon, Paul, Head of InterMedia Division at Oracle, Conversation with the author, Jan 2000.

enough to satisfy most graphic design requirements. We break up the design of the maintenance component into the following components:

- Creating an object-based approach to building templates
- Support of automatically generated ADP tags assigned to individual components
- Creation of nodes to logically group pages in the site together.

The reader is encouraged to look back at Section 3 to better understand the process of creating site-wide templates.

4.2.2.1 Object-based approach to building templates

The object-based concept to building templates is simple: every time a type of element is needed on a particular page, we create that object and ensure that it remains available to all existing and future templates. With every object we create, we also document its function. This makes creating future templates easier as we will be able to draw from a library of objects that have already been created, debugged, and documented.

Every object that is created can contain HTML, ADP generated from other objects, or Tcl. We include TCL as a supported language because there will inevitably be templates that cannot be created without programming. Rather than create a new language to deal with these types of templates, we simply include the programming language itself.[18]

Note that since each template object is now reusable, the graphics designer can easily reuse object previously created by the programmer. Thus, if a certain object requires some programming, it need only be created once and can be reused in the future without further interaction between the programmer and graphics designer.

We add a final component to the concept of our objects. Each object will store basic metadata, pre-defined by the programmer, to allow the graphics designer to easily select when that element is to be used. This allows the programmer to essentially capture the most basic if/then statements into a user interface for the designer. For example, we know in advance that menu bars change depending on whether or not a user is logged into a system. Every object should "know" if it is to be displayed based on the status of the current user's login status. Currently, the login metadata is the only flag our objects support.

Note that we are using the term object loosely here. All we mean is that SWTM will allow us to reuse all the components ever created, assist in documenting existing objects and provide some basic options to help graphics designers select some standard requirements that must be satisfied to display the object.

4.2.2.2 Automatically Generated ADP Tags

For every object we create, we want to assign it a unique, immutable name that can be used as an ADP tag. This ADP tag can then be used inside other objects, creating a bottom-up template design environment. Note that we do not allow the names of these objects to change else we would break all other references to the object.

[18] The decision to include support for TCL in the definition of an object also stems from the design of EMACS that emphasized that the system must support and execute new procedures defined after compilation. Each templating object can be thought of as a new procedure in the system and is indeed implemented as such. TCL code executing TCL code!

To support top-down template design, we add a feature that inserts a placeholder when an unknown ADP object is used. We also offer a toggle to turn off this feature, if desired, since placeholders essentially mask "template object not found" errors that we want to catch before moving to production. This idea of a debugging/development mode comes from the design of Multics Emacs with its "lisp-program-editing mode to facilitate the interactive development and debugging of extensions as they are being written"[19].

4.2.2.3 The node manager

The node manager is the module that associates directories with SWTM. Our primary goals here are to make it easy to associate metadata with every object in the system and to ensure that the additional information does not impact system performance.

A node is a container for files in the system that also stores information common to all its files. The main reason nodes are important is that grouping, in general, is the only way to create a scalable maintenance model of a website. It is much easier to think of a site as a tree of ten nodes than a tree of 1,000 files.

Files in a file system can be logically grouped together. For example, each subdirectory should contain files that perform tasks that are somehow related. We continue to use the idea of a directory on the file system as representing one node that contains its files. Though this may seem limiting at first, in terms of long-term maintenance, it is very important to keep the file system and database as closely connected as possible. Note also that using directories as nodes makes it easier to initially group files in the system to nodes in the database.

Additionally, every file stored in a node may contain optional information including:

- Page title (overriding what the programmer indicated in the file itself)
- Subsection (again to highlight a specific item on the menu bar)

The node manager must be flexible enough to accept new fields in the future and must be very fast, as it will potentially be called every time a user requests a page. Thus, it is important to cache the information contained in the node manager and in SWTM in general.[20]

4.2.3 Decrease overall development time

If we successfully create SWTM and separate the creation/maintenance of templates from the programmer's everyday job, we claim that overall development time will decrease. The basic idea here is that the programmer does not have to think about the template for the page currently being development. Instead, the programmer simply inserts calls to `swtm_header` and `swtm_footer` respectively around the contents of the page. The page will automatically be formatted exactly like the rest of the files in the node.

4.2.4 Allow for minimal work in upgrading legacy systems.

To place an existing system's templates under the control of the SWTM, the following items must be completed:

[19] Greenberg, Bernard S., "Multics Emacs: The History, Design, and Implementation," August 15, 1979, http://www.multicians.org/mepap.htm, Page 13.

[20] Caching the information stored about a node or file is extremely important. In an early experiment in October 1999, the author turned off caching on a live system. Within ten minutes, the system had reached capacity handling 400 hits per minute on a Sun E450 with four processors at 400mHz and 4gb of RAM. Reinstating caching quickly restored the system, which later reached capacity limits at approximately 1,600 hits per minute. Caching resulted in a 400% increase in system capacity.

- Templates must be created for every unique node in the system. We can identify a unique node in a legacy system as a set of pages that has a significantly different look and feel.
- All pages must be modified to replace whatever legacy code generated either the header or footer with calls to `swtm_header` or `swtm_footer`.

Zzzz – replace tedious

This process is very straight forward, though admittedly somewhat tedious. If the design of the legacy system already abstracted away some of the template components, then upgrading should be relatively simple (perhaps a short perl script could even do most of the work!). However, if the legacy system is programmed with erratic and inconsistent style, upgrading will be time-consuming and the system could probably use a re-write anyway.

4.3 Design Summary

SWTM will consist of the following pieces:

- Simple API
- Template manager – create/manage/maintain templates and the objects associated with each node in a template
- Object manager – create/maintain objects in the system
- Node manager – create/maintain nodes and their metadata
- File manager – manage attributes assigned to individual files
- Field manager – create/maintain fields used to store metadata in all other parts of the system

4.3.1 SWTM API

The SWTM API consists of two procedures that have no required arguments:

- `swtm_header` – queries SWTM Node Manager for all the objects needed to generate the header for the current node and template. Serves those objects in their specified order.
- `swtm_footer` – queries SWTM Node Manager for all the objects needed to generate the footer for the current node and template. Serves those objects in their specified order.

These are the only procedures required to use SWTM. However, we provide the following functions for convenience:

- `swtm_var <variable name>` - returns the value of the specified variable for the current template, node, and file.
- `swtm_parameter_default_template` – Returns the unique identifier for the default template in SWTM. The name of this identifier can be changed on a server-by-server basis.
- `swtm_template` - returns the unique identifier for the current template. The template identifier is stored in a cookie or is maintained in the URL. If there is no current template, this function returns with a call to `swtm_parameter_default_template`.
- `swtm_url <url>` - adds the current template identifier to the start of the url, if necessary. This is necessary when the site owner has decided not to use cookies to maintain template information as every absolute link must then contain the template identifier. Note that if cookies are not used to store the template information, every absolute url in the system must be inserted with a call to this function, else the template information will be lost.
- `swtm_return_template` - API call to replace the two calls to `swtm_header` and `swtm_footer`. This procedure looks for a variable called `page_content` in the calling

environment, and returns a string which is equivalent to sending the following sequence to the web browser:

1. `swtm_header`
2. value of `page_contents`
3. `swtm_footer`

4.3.2 SWTM Template Manager

The template manager is the main point in the system from which we derive all other functionality. The template manager

- Maintains all site-wide templates in the system
- Maintains all variables defined for a given template
- Maintains the objects, and their calling order, for each node that is assigned to a template

At a high-level, the template manager is the primary interface to SWTM. Users create templates, assign them identifiers and values for certain user-created fields, register nodes with the templates, and assign objects to generate the header and footer for each node. In this way, the template manager truly integrates all of the functionality of SWTM in one central location.

4.3.3 SWTM Object Manager

The goals of the object manager are:

1. Create an object-based approach to building templates
2. Include a simple user interface
3. Provide support for categorizing objects

The SWTM Object Manager must also include some kind of development environment. It should include a preview facility to show the user what the template currently looks like, and summary views to see what nodes use which objects and which objects are being used by what nodes. We expect most of the time users spend working with SWTM to be inside the Object Manager, especially before the library of objects has been developed.

4.3.3.1 The object-based approach

Real-life HTML templates are built up of many smaller blocks. The template manager must support this concept of very simple, object-oriented template design. Although we call this system object-oriented, it is important to note that we do not allow for inheritance of any kind between template objects.

To implement this system, we create two tables. The first stores information about the template object, including its source code (in HTML or ADP), unique identifier (to create the ADP tags (described in Section 4.3.3.2: Creating a simple user interface) and any additional properties stored for all objects (e.g. this object only applies to users who are logged in). The second table stores references to other objects in the system. This is crucial for two reasons:

1. Before removing an object, we must ensure that we do not break any other object in the system
2. It is extremely helpful to view all system dependencies from an individual object in the system.

4.3.3.2 Creating a simple user interface

We assume that our graphics designer already has experience with and knowledge of HTML. We require that the designers assign a unique identifier to every object that they create. These identifiers are wrapped into ADP tags that the designers can use in later templates. Note that ADP tags are unique across SWTM – a designer cannot assign an ADP tag to a template variable and to an object identifier.

Registering new ADP tags should be part of the API to any web server. To create a new tag with AolServer, the programmer simply associates the start and end ADP tags with a Tcl procedure. The procedure takes as arguments a string (whatever is between the start and end tags) and a set of key/value pairs (which are specified in the first ADP tag).[21] As an example, let us create a replacement for the HTML <h1> tag called `<my_h1>`. The user would register the tag `my_h1` with the procedure `process_my_h1_tag`. When used, e.g.

- `<my_h1 size=+2>Testing</my_h1>`,

the procedure `process_my_h1` would be called with the arguments `"Testing"` and the set `(<size=+2>)`.

To increase the readability and ease the maintenance of our user-entered code, we choose to provide a minimal set of tags that take an argument specifying the unique identifier of the object/variable.

4.3.3.3 Object Categories

The designer should have the option to create various object types. This is important in helping the designer quickly identify components that can be reused in developing new templates. For example, one object type may be "Top-level Navigation" that contains all the objects that already implement top-level navigation. This system of categorization is just a simple mechanism to categorize objects for easier future retrieval. Additionally, we allow each object to be associated with a particular template, indicating that this object is customized to the point at which it is no longer useful in a general sense.

4.3.4 SWTM Node Manager

The goal of the node manager is to simply maintain all of the nodes that are currently being used by SWTM. Before a template assigns objects to a node, the node must first be registered with the Node Manager.

We also note that our choice of representing each directory in the file system as a node allows us to traverse the tree of nodes in the same manner as the file system – by appending or deleting directories from the path.

4.3.5 SWTM File Manager

The goal of the file manager is to maintain metadata assigned to individual files. In general, the programmer will assign most of the metadata in the file's actual source code. However, it is desirable to provide the graphics designer with a way to override the programmer's choices. For example, the graphics designer may want to change the page title or to highlight a different section of a menu bar. The file manager would provide this functionality.

4.4 Other Design Considerations

4.4.1 Maintaining system performance

SWTM must be efficient. We provide a central method to cache any data in the database through two API calls that are closely tied to SQL:

[21] For more information, please refer to the `ns_register_adptag` function of the AolServer API at http://aolserver.com/doc.

- `swtm_memoize_list` – takes a sql query, and caches the result as a list of all the columns/rows returned.
- `swtm_memoize_one` – takes a sql query, and caches the first element returned

All other calls in SWTM must use these two functions to cache all results. Note that caching metadata for nodes and files makes sense since there are very few nodes and files. In addition, the amount of data we store for each node or file is limited. Note also that the system of caching we use is optimistic in that it caches all the information for a specified node, and its files, the first time any piece of information for that node is called. This takes advantage of spatial locality in the way users interact with a web site.

An alternative design would cache individual pieces of information as they are requested. We feel most of the data will eventually be requested once a user first accesses a node, thus making our implementation more efficient.

4.4.2 SWTM Library

We initially considered including a formal library that would serve as a central repository from which to find all other objects, templates, files, etc. in the system. However, as we implemented the design, it became clear that the library was truly useless. The functional division in SWTM along the lines of templates, nodes, objects, and files, along with the ability to associate any object or node with an individual template made it easy to find any needed element in the system. One enhancement to SWTM would provide a search function inside each of these functional divisions (e.g. a keyword search box that found all objects that mentioned the word "header").

5 Implementation

The implementation of SWTM can be broken into the following pieces:

1. User pages
2. Data model
3. API

We choose to start with the user pages to provide an overview of how a user actually interacts with SWTM, without spending too much time initially on any other aspect of the system. We then discuss the data model that supports the activities of the user. We finish by outlining the API provided to both programmers and graphics designers.

5.1 User pages

The user pages are designed to capture the main modules of SWTM:

- Template manager – create/manage/maintain templates and the objects associated with each node in a template
- Object manager – create/maintain objects in the system
- Node manager – create/maintain nodes and their metadata
- File manager – manage attributes assigned to individual files
- Field manager – create/maintain fields used to store metadata in all other parts of the system

Each module is implemented as a stand-alone subsystem, with links created between modules. The root directory for SWTM is identified by the procedure `swtm_url_stub`, which in our implementation returns `/admin/swtm`. This is the base directory from which users work and simply contains links to all the other modules.

5.1.1 Template Manager

The template manager is the central module in SWTM, from the point of view of functionality – almost all other modules reference templates either directly or indirectly. This module controls the actual generation of the look-and-feel for each node registered to each template. In a sense, the template manager is the interface between templates, nodes, objects, and the rest of the web site.

5.1.1.1 Core Template Information

Each template is identified by a unique `template_key` and must contain a `template_name`, which is a human understandable identifier for the template. Additionally, users can specify whether a template is currently active. An active template is registered with the web browser so that pages can be served appropriately when requests for the `template_key` are received (e.g. `http://servername.com/template_key/*`). An inactive template is useful when debugging to ensure that outside parties cannot access files using the specified `template_key`.

Registering the template key with the web browser is usually very straightforward. In AolServer, during server startup, we use the API call `ns_register_proc` to "tell" the server to redirect certain requests (see Figure 7). Note that we redirect all urls beginning with template keys in one of two ways, depending on whether or not we are using cookies to maintain the current template. If we are using cookies, we redirect requests for the template key to a procedure that takes the template key out of the url, inserts it into a cookie named `swtm_template`, and redirects the user to the same url, with the template key removed. If we are not using cookies, we simply locate the requested file on the file system, and serve it appropriately (e.g. Tcl files are served as Tcl, ADP files as ADP, etc.). The same functionality can be accomplished in Apache using ModRewrite to rewrite queries for pages beginning with `template_key` to the appropriate location.

```
foreach template [swtm_template_list] {
     if { [swtm_parameter_using_cookies] } {
         ns_register_proc GET /$template/* swtm_set_template_cookie
         ns_register_proc POST /$template/* swtm_set_template_cookie
     } else {
         ns_register_proc GET /$template/* swtm_serve_page
         ns_register_proc POST /$template/* swtm_serve_page
     }
}
```

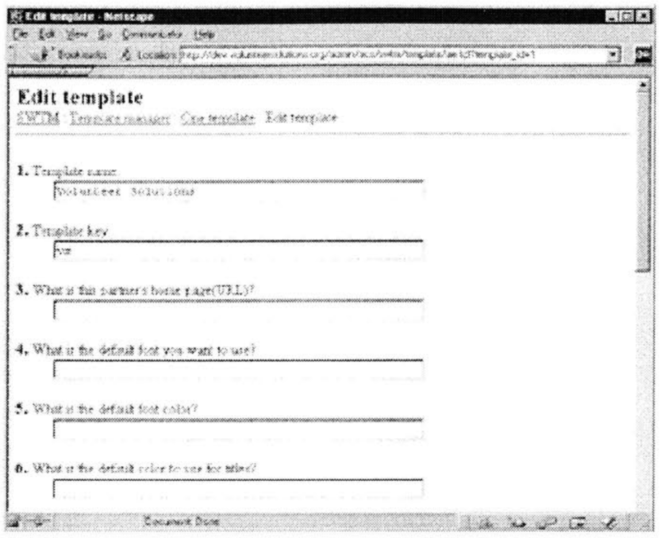
Figure 9: Procedure to register template keys with AolServer

5.1.1.2 Template Metadata

In addition to the basic information we store about each template, we need to allow users to dynamically add additional information to the template. This is crucial if we want to continue using SWTM as the number of templates increases, each adding potential new pieces of information that are incorporated in various pages on the site.

The template manager provides support to add new fields to the system, through a link to the Field Manager, and to provide values for each of the fields associated with all templates. Note that fields are globally visible to all templates, and there is currently no support for a field specific to one template. Through experience building templates for several sites,[22] we found that any field that is useful to one template will eventually be useful to future templates. There is no need or reason to associate one field with a specific template. Figure 10 shows the screen to edit template information, including values for several fields associated with templates.

[22] The author's primary templating experience comes from building multiple templates as guidestar.org and volunteersolutions.org both matured.


Edit template

SWTM Template manager One template Edit template

1. Template name

Volunteer Solutions

2. Template key

vs

3. What is this partner's home page(URL)?

4. What is the default font you want to use?

5. What is the default font color?

6. What is the default color to use for titles?

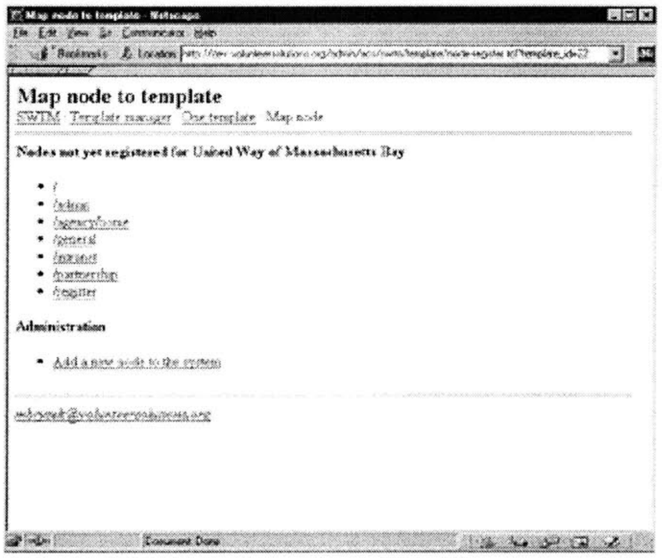
Document Done

Figure 10: Screen to edit template information and specify values for template fields

We also add an optional field that associates templates with user groups in the existing web site architecture. This is useful since many existing sites already make use of the concept of user groups. We may want each user group of a certain type to be associated with a template. The danger with associating templates to user groups is that only a small number, if any at all, of user groups will actually use templates. Thus, for user interface purposes, we set up the data model to look for user-groups of type "site-wide-template" when offering users the choice of selecting a group to associate with a template.

5.1.1.3 Registering Nodes with Templates

Another function of the template manager is to allow users to register nodes in the system with each template. Users can easily register nodes previously created in the node manager to each template by simply following a link to register a new node. Each registered node is activated by default, but can be deactivated. Figure 11 displays the screen to register a new node for a template.


Map node to template

SWTM Template manager One template Map node

Nodes not yet registered for United Way of Massachusetts Bay

- /
- /admin
- /agency/home
- /general
- /intranet
- /partnership
- /register

Administration

- Add a new node to the system

[illegible]

Document Done

Figure 11: Screen to register a new node with a template

5.1.1.4 Registering Objects with Nodes

The final major function of the template manager is to allow users to assign objects that define the header and footer for a specified template/node pair. Once the desired objects are created, the user can select the desired node to see all registered objects for both the header and footer and can register new objects for that node. This screen also allows users to preview what the current look-and-feel for the template/node pair looks like.

Note that the idea of registering multiple objects to represent the header or footer for a template/node pair could have been left out of the implementation of SWTM. The user could have created a new object that was a wrapper for all the objects needed to generate either the header or the footer. However, the author felt that it was simpler, both in terms of time and in minimizing the total number of objects in the system, to allow the user to register multiple objects explicitly for each template and node. Seeing the objects themselves makes it easier for the user to later reuse the same objects while specifying a new header and footer for a different node or template.

5.1.1.5 Caching Template Information

Performance is one of our primary concerns, and caching the template information is part of maintaining system performance. We want to make it quick to lookup the value of any template variable, including the basic information and the metadata specified through the user created fields. The need for caching this information led to the procedure `swtm_var`, described below, which caches the values of all the variables for a given template the first time any of the template variables are accessed.

Additionally, we need to cache the list of objects specifying the header and footer of each template/node pair. The procedure `swtm_object_tags_for_url` implements this caching. Note that unlike fields where we cache all values the first time any field is accessed, we cache objects for only the template and node we are currently accessing. Over time, the number of nodes in the system can grow quite large and it is not clear that users who touch one node will touch the others. In most well organized web sites, the author actually believes that users tend to migrate to one node where they spend the majority of their time.[23]

Note that the SWTM user pages do not use the caching feature, as it would make data updates transparent. Rather, all data accessed through the user pages forces a reload of the cache. This makes sense since the user expects changes made to show up immediately on the website.

5.1.2 Object Manager

The object manager is the module that accomplishes many of the most interesting and challenging objectives of SWTM:

- Creating a simple user interface

[23] The behavior of users with regard to the nodes they access when entering websites would be an interesting experiment that the author has yet to explore. A few examples, though, provide insight into the author's decision to cache individual nodes as they are accessed:

- cnn.com – users who access the WEATHER directly may not be interested at all in the CURRENT_EVENTS node
- photo.net – users often spend most, if not all, of their time in one of two nodes: BBOARD or PHOTO. Neither node indicates in any way that the user will reach the other in any short period of time
- volunteersolutions.org – depending on the type of user, all the user's time will be spent in either the VOLUNTEER or AGENCY node
- guidestar.org – most users spend all their time in the SEARCH node, never seeing the MEMBERS or HELP nodes.

- Implementing an object-based approach to developing site-wide templates

We begin with an overview of the implementation of an object, and then focus on the user interface.

5.1.2.1 What Are These Objects?

An object in SWTM is simply a string that contains HTML, ADP, and/or TCL. In addition to the source itself, each object stores several pieces of metadata:

- A name that is used to help the users identify the object and understand what it does.
- A description that is a more verbose explanation of what the object does.
- An optional flag indicating whether the object should be used for users who are either logged in or logged out. This is especially useful when creating menu bars that depend on a user's login state.
- A category to which the object belongs. Categories can be created dynamically, but common categories include General, Header, Footer, and Menu bar. The categories are used to help the user organize the objects in the system for faster future identification.
- Optional association with a template. Some objects are built for a specific site-wide template and cannot be reused in other templates. By associating these objects with a specific template, we can significantly simplify the user interface when registering objects with nodes.
- A flag indicating whether the object should be run through the Tcl interpreter. We expect that most graphics designers will never enable the Tcl interpreter, but we provide this feature as a way for programmers to create objects that contain Tcl code. Note that the user can also use ADP syntax to include Tcl code, making this flag a convenient, but unnecessary, feature.
- The key to uniquely identify the object and to incorporate the object in other templates. Once an object is created, its key can never be changed, though this functionality could eventually be incorporated.

Each object must also be able to "include" other objects to allow for object reuse. We attain this functionality through a special tag, `<swtm name=key>`, that dynamically inserts the object with the specified key (see section 5.3.2 for a more thorough explanation).

5.1.2.2 Creating a Simple User Interface

The web form to create objects is relatively straightforward. We expect graphics designers to already have html mockups of the objects they create, developed using their standard desktop tools. Designers need only name their objects, assign a unique key, and paste in the html source.

Once the object has been created, the user interface becomes more important. One of our goals is to create a friendly environment that aids in debugging. To this end, we provide several debugging functions.

5.1.2.2.1 Debugging mode

The developer can set the system in Debugging mode that inserts placeholders for referenced variables and objects that do not yet exist or that could not be found. This makes it easy for the user to identify remaining objects and variables that must be created. When in debugging mode, we return an error string, in html, describing the problem, which in the case of objects will usually be "variable or object is not defined". When debugging mode is turned off, we return the empty string rather than the error message. Production sites should turn off debugging mode to ensure that even if a page is missing a variable, it does not interfere with the web surfer's reading of a page. Note that when debugging mode is off, the error is still logged in the web server's error log.

5.1.2.2.2 Preview mode

Each object has a link to a preview that includes the object source, the html itself of the parsed object, and a "screen shot" of what the object actually looks like (where the screen shot is simply the parsed object placed inside a table with a small border).

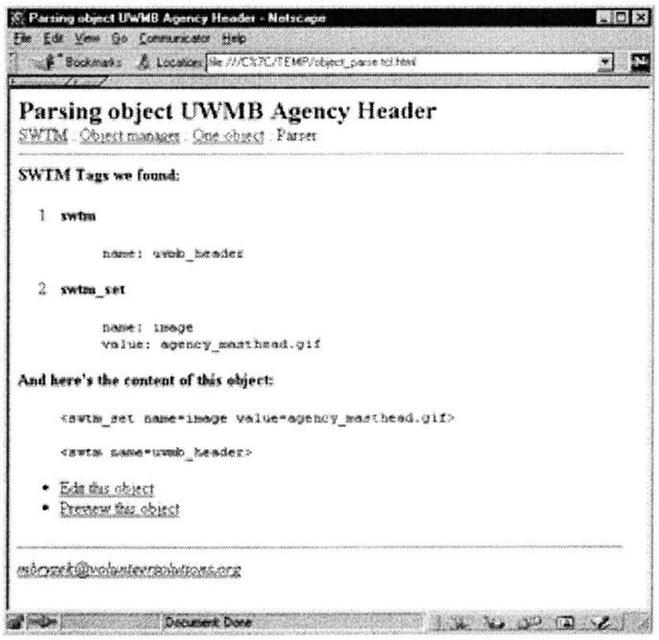
The preview shows the actual object itself, but currently has two major limitations:

- Broken html code could prevent the screen shot from appearing. The problem is that the html code may be broken on purpose, such as a header leaving an open <TD> tag to start the column that contains the page body. With a robust html parser/verifier tied into the preview option, we would be able to identify the broken html, highlight the possible problem, and add enough html code to allow the object to display correctly. We leave this modification to future versions of SWTM.

- Some objects will rely on variables being defined in either the file being accessed or a parent object that includes the currently viewed one. This results in additional variable not found errors, if the system is in debugging mode. Though somewhat annoying, these errors should be treated as warnings/reminders to ensure that the specified variable is later somehow specified.

5.1.2.2.3 Parsing an object

Since the SWTM API includes a few new tags (see Section 5.3), we provide a parsing function that displays the SWTM tags we found while parsing the object. This allows the user to quickly identify missing tags and verify that the correct objects and variables are being used. Figure 12 shows sample output from the parse function.


Parsing object UWMB Agency Header - Netscape
File Edit View Go Communicator Help
Bookmarks Location: file:///C%7C/TEMP/object_parse.tcl.html
Document: Done

Parsing object UWMB Agency Header

SWTM : Object manager : One object : Parser

SWTM Tags we found:

1. **swtm**

```
name: uwmb_header
```

2. **swtm_set**

```
name: image
value: agency_masthead.gif
```

And here's the content of this object:

```
<swtm_set name=image value=agency_masthead.gif>

<swtm name=uwmb_header>
```

- Edit this object
- Preview this object

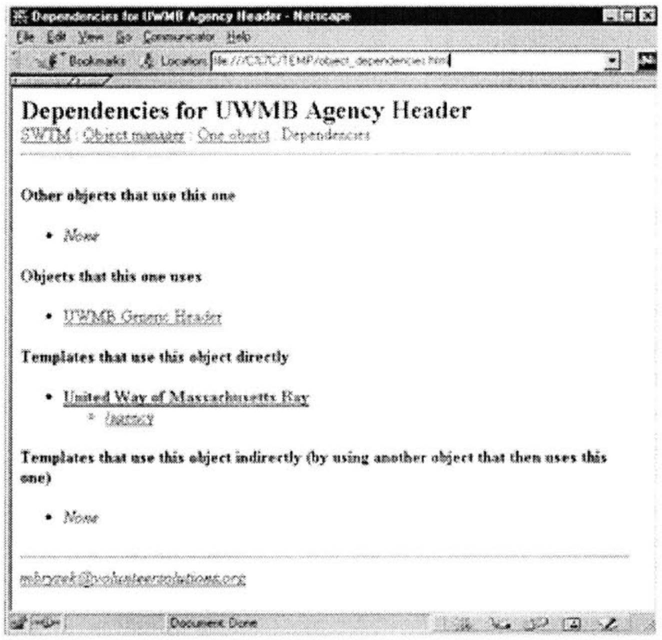
mbryzek@volunteersolutions.org

Figure 12: Parsing the "UWMB Agency Header" object

5.1.2.2.4 Object Dependencies

As the object library grows in terms of the number of objects, we need an easy way to view all dependencies for individual objects. Thus, we maintain all dependencies in the system in one central table that maps any tag to another. This works well for objects as one object includes another by using the second object's tag.

From this information, we can generate a page that shows all the object and template dependencies one object has (See Figure 13).


Dependencies for UWMB Agency Header - Netscape
File Edit View Go Communicator Help
Bookmarks Location file:///C:/TC/TEMP/object_dependencies.htm
Document: Done

Dependencies for UWMB Agency Header

SWTM : Object manager : One object : Dependencies

Other objects that use this one

- *None*

Objects that this one uses

- UWMB Generic Header

Templates that use this object directly

- United Way of Massachusetts Bay
  - agency

Templates that use this object indirectly (by using another object that then uses this one)

- *None*

mbryzek@volunteersolutions.org

Figure 13: Dependencies for one object

5.2 Data model

Before we delve into the SWTM API, we outline a few of the major components of the Data Model with the goal of making it easier to understand the implementation of the queries that the API uses. The full data model is attached as Appendix A: Oracle 8i Data Model. We should also note that just about every table in SWTM has an integer primary key – including the mapping tables discussed below.

5.2.1 Central Tables

The following tables make up most of the SWTM data model:

- `swtm_template` – stores basic information about all templates in the system
- `swtm_tags` – stores all of the ADP tags currently in use
- `swtm_type` – stores information about all the types we are using
- `swtm_field` – stores each field the user has added
- `swtm_object` – repository for all user created objects
- `swtm_node` – stores all the nodes registered with SWTM
- `swtm_file` – stores all metadata associated with individual files in the system

5.2.1.1 Storing templates - swtm_template table

The `swtm_template` table stores central information about each template, including:

- The template key used to uniquely identify each template
- The name of the template
- An optionally associated user group
- Information about whether the template should inherit the nodes and objects from a different template.
- Whether or not the template is currently active

The `swtm_template` is also used as a reference for mapping fields that the user has created.

5.2.1.2 Storing Tags – swtm_tags table

To minimize any possible confusion when interpreting an ADP tag, from the point of view of both SWTM and the users, we enforce the constraint that all tags are unique. The tag of an object or field uniquely identifies a row in either the `swtm_object` or `swtm_field` table. Note that it would be possible to allow both objects and fields to have identical ADP tags, as we always know the context in which the tag is being parsed, but we chose not to add this possible confusion to the system.

Each tag stores information about the table and row that is using the tag. This allows us to work backwards, if need be, from the tag to the referencing row, and also enforces the principle of system-wide unique tags.

5.2.1.3 Storing Types – swtm_type table

In SWTM, we introduce types as a way to group fields added to the system and objects assigned to roles in templates. Having this one table store all of the types in which we are interested simplifies the overall data model by reducing the number of tables we would have to replicate. In our implementation, both fields and objects, when assigned to roles, are assigned a type.

5.2.1.4 Storing Fields – swtm_field table

We need an easy way to keep track of all of the fields that the user adds to this system. The `swtm_field` table implements a basic metadata system that consists of the following information for each field:

- Its global type (through `swtm_type`)
- Its presentation type (e.g. this is a Boolean or text field)
- The text to display when asking the user to fill in data for the field
- The tag to use to retrieve the value of a field
- Whether or not the field is required, and if so, how to prompt the user
- An optional default value

Based on this information, we can dynamically generate a form to capture all the information the user has specified. Note that the use of `swtm_type` lets us use the same `swtm_field` table when obtaining data for new templates or for new files.

The field's presentation type is stored in a separate table, `swtm_field_types`, that currently only supports Boolean and text fields. The SWTM API consists of a Tcl procedure, `swtm_display_field`, which formats the field for display, inside an html form, based on the type. To add a new presentation type to the system, users need only add a row to `swtm_field_types` and modify the procedure `swtm_display_field`.

We store all user-entered data for a particular field in the `swtm_field_value` table. The values table is the only one that is not keyed by an integer primary key – rather, the primary key is a combination of the field with which the user-entered value is associated, and the referencing table/row for which the user entered the data.

5.2.1.5 Storing Objects – swtm_object table

The `swtm_object` table stores all the information about each object, including:

- To what category the object belongs
- Its name
- Its tag
- Its content
- A user entered description
- Flags describing what user state is required for this object to be active
- With what template, if any, the object is affiliated
- Basic auditing information, including who created the object and when, and who last modified it and when

To store information about the category to which an object belongs, we use a helper table, `swtm_object_category`, which simply stores a string describing the category. The user can dynamically create new categories while adding/editing objects.

5.2.1.6 Storing nodes – swtm_node table

For nodes, we only store the URL stubs that are part of SWTM. Every URL stub must be a directory in the file system, though currently nodes are not automatically removed when the underlying directory is removed. We considered adding this logic, but chose against it for two reasons:

1. Once a file system directory is removed, the node will never again be accessed, automatically deactivating that node.
2. It is not clear that we would want to remove the information regarding the objects associated with the node for the various templates.

We believe the automatic and natural deactivation of nodes is a better option. A possible enhancement could offer a feature to verify the validity of all the nodes, but this is left to future work (see Section 7.2.2 Better File System Integration).

5.2.1.7 Storing Metadata for files – swtm_file table

Similar to the `swtm_node` table, the `swtm_file` table simply stores the node to which the file belongs and the path to the file, relative to the node's URL stub. Files that are registered in `swtm_file` can then be associated with various fields that the user creates for files.

Note that associating files with nodes allows us to rename nodes without having to worry about losing file properties. However, when a new node is added or an existing node is deleted, we may end up in an unusual state. For example, if the file "`/volunteer/home/index.tcl`" belongs to the node "`/volunteer`," when we add the node "`/volunteer/home`," we need to decide whether or not to move the file to the new node. We chose not to move the file primarily to reduce confusion. If graphic designers add files to one node and later create new nodes, they would potentially be surprised if their previously added files were no longer registered with the same node.

5.2.2 Tying the Central Tables Together

SWTM relies on three general mapping tables that tie the rest of the information in the system together:

- swtm_object_object_map – Maps one tag to another as a way of saying "This tag uses that tag." This map only enforces the foreign key constraint on the from tag since a user working in a top-down fashion might create an object that uses an object that has yet to be built.

- swtm_node_map – Associates nodes with any other table/row in the system.

- swtm_node_object_map – Maps objects in a specified role, through the swtm_type table, to a mapping of a node.

The swtm_object_object_map table is currently modified whenever new objects are created or existing objects are edited (since only objects can include other objects). Each time the content of an object changes, we parse the content to pull out all calls to include other objects, and update this table. This table is used mainly for user-interface purposes in exposing the dependencies between objects.

The swtm_node_map table allows templates and files to be associated with a given node. We simply store the node_id and the referencing table and row. In addition, the node_map table stores a flag indicating whether or not the mapping is active.

The last mapping table associates objects, types and node mappings. Any object can be assigned into a role, as specified by the swtm_type table,[24] and associated with a row in the node_map table. This lets us easily assign objects to a specific node for a given template. Note that this type of data model also lends itself nicely to future extension. For example, the data model supports associating objects with files registered to given nodes, though we presently see no reason to add this functionality.

5.3 API

The SWTM API can be broken up along the lines of the interface given to programmers and that given to graphics designers. We start by outlining the programmer's API as the graphics designer's API is necessarily a subset of that provided to the programmers (since programmers will have access to whatever API we provide to graphics designers).

5.3.1 Programmer's API

We are going to outline the Programmer's API starting with system-wide parameters, then describing the interface to maintain information regarding the current template, and concluding with the actual serving of pages in the system.

5.3.1.1 System-wide Parameters

We augment our AolServer instance with a few parameters and provide procedure calls to easily return the value of the parameter (see Table 1).

Parameter	Procedure Call	Description
URLStub	swtm_parameter_url_stub *or* swtm_url_stub	Returns the path to SWTM, relative to the server root, with no trailing slash. (e.g. /admin/acs/swtm)

[24] We considered adding another swtm_role table, but found it unnecessary. Roles and types are very similar in the case of SWTM and an object's role can be thought of as the type of role the object plays in the current mapping.

`DefaultTemplate`	`swtm_parameter_default_template`	Returns the default template to use when we cannot identify the template based on the user's request or cookies
`UseCookiesP`	`swtm_parameter_using_cookies`	1 if we are using cookies to keep track of the template. 0 otherwise.
`DebuggingModeP`	`swtm_parameter_debugging_mode`	1 if we are in debugging mode. 0 otherwise.
`TemplateGroupType`	`swtm_parameter_template_group_type`	Returns the type of the user groups we want to associate with individual templates. Returns the empty string if no type has been selected
`TraceP`	`swtm_parameter_trace_p`	Returns 1 if we are in trace mode. 0 otherwise (Trace mode refers to the logging of all the procedures calls in SWTM).

Table 1: Overview of Server Parameters

5.3.1.2 Maintaining template information

SWTM offers two ways to maintain information regarding the current template:

1. **Cookie Method** – In a persistent cookie named "`swtm_template`"
2. **URL Method** - In the url of every page request.

In either case, we are storing the `template_key` for the current template, a unique and non-intrusive identifier from which all other information can be retrieved. The programmer always requests the current template by calling the `swtm_template` procedure, which first identifies the method being used to store the template key, and returns the template key (or default template if there is no current template key).

The major difference between the two methods for storing a template key is that with the cookie method, no links in existing pages need to be modified whereas with the URL method, every absolute link (e.g. starting with /) must be replaced with a call to `swtm_url` to ensure that the `template_key` is included in the link.[25] To partially remedy the problem, objects that are served are run through the swtm procedure `swtm_parse_hrefs` that automatically adds in the `template_key` if necessary. The programmer could easily modify the API to also run all page content that is served to the user through this same procedure, though we expect a such a global parsing of the `<a href>` tags produce erroneous links that will be hard to track down.

An additional concern when using the URL method is that pages that rely on asking the server for the URL of the current page could be fooled as the URL returned to this script will not actually point to the

[25] Note that the problem of maintaining the template key in all URL requests is similar to that with storing session identifiers in the URL as opposed to in cookies. Once the session ID has been established, every link must contain it, else the session information is lost. The advantage with our URL method is that relative links automatically inherit the template key.

file being served. SWTM provides an additional procedure, `swtm_conn_url_no_template`, which returns the URL without any initial template key.

It is also worth mentioning one other, somewhat subtle, pitfall with the URL method. Normally, when url stubs are registered with the web server to trigger certain procedures to be called, they are registered from the server's root (e.g. we would register the `/help` directory on guidestar.org by assigning `http://guidestar.org/help/*` to a procedure). This type of registering will not work when serving pages in the URL method. SWTM provides its own call to register procedures with the server, `swtm_register_proc`, to automatically register the specified directory for all template keys being used.

Although the URL method is more tedious to work with, it does provide several key benefits that in many cases cannot be ignored:

- When users bookmark a page, their bookmark will contain the template key so that even if users lose their cookies file, the template information will be preserved next time they come to the site.

- The server log will contain information regarding the amount of site activity each template key received. Without the template key in the URL, the amount of traffic for each template could not be determined.

We recommend using the cookie method as a starting point since it is simpler to implement (especially when porting a legacy system). However, the URL method is provided as an option when business demands require more tracking of usage across the templates.

Procedures:

```
swtm_template { }
   returns the value of the current template, or the default template if there is no
   current template.

swtm_url { { url "" } }
   If the current url begins with a leading slash, meaning it is an absolute link from
   the server's root, this proc will prepend the template variable to the url if we
   are NOT using cookies. Note that "/" is left as "/" (how else could you specify the
   root?) And an empty url is either returned as empty or returned as /template if
   we're not using cookies

swtm_parse_hrefs { string }
   Replaces all hrefs with calls to swtm_url to add in the template when necessary.
   Note - if we are using cookies, this procedure does nothing.

swtm_conn_url_no_template { {url "" } }
   Returns the current url (from url, if specified, or ns_conn, if we have a
   connection), minus any template identifier. Default return value is the current url

swtm_register_proc { from to {inherit 1} }
   Registers procs for all templates
```

5.3.1.3 Initializing the system

We now explain the SWTM API for AolServer to initialize the web server to recognize and appropriately serve requests for templates. These procedures are simple and could easily be ported to most other web servers. The key procedure is `swtm_initialize` that coordinates the registering of template keys to the appropriate processing procedures.

At server startup, we need to identify all of the active template keys in SWTM, and register them as url stubs with the web server. The call `swtm_template_list` caches and returns a list of active template keys. The next step is to redirect all requests to the template keys to a procedure to handle the request. There are two such procedures:

1. When using the cookie method, send requests to `swtm_set_template_cookie` that will identify the template key at the start of the url, set the `swtm_template` cookie to contain that key, and redirect to the requested url, with template key removed.

2. When using the URL method, send requests to swtm_serve_page that will identify the appropriate file on the file system and process it according to its type (Tcl, ADP, html, etc).

Procedures:
```
swtm_initialize {}
   Registers all the template url_stubs as one of two procs (depending on whether or
   not we're using cookies!)"

swtm_set_template_cookie {}
   Strips off the cookie that identifies a template from the first part of the url.
   Redirects to cookie-chain to set the cookie, ending at the page the user requested,
   minues the template identifier.

swtm_serve_page {}
   Function that reads the current filename from the url, and then parses/sources that
   file depending on its extension. This function strips off the template identifier
   before identifying the physical file, letting us implement the virtual url's based
   on those template id's.

swtm_template_list {}
   Returns a list of all the template identifiers. Caches the result.
```

5.3.1.4 Hooks to work with the template key

Since the template information is all keyed on the one string `template_key`, we provide several functions to convert back and forth between other identifiers that are often used while building pages with SWTM. Namely,

- Obtaining a `template_id` from its key
- Obtaining a `template_key` from its id (useful when working with database information since foreign keys use `template_id`)
- Obtaining the `group_id` for a given template
- Obtaining the template for a specified user id. Note that this relies on the user belonging to at least one group in the system that is tied to a template.

All of these procedures cache the information they retrieve to minimize database accesses for these common pieces of information. The caching also allows us to quickly shift between these identifiers to simplify database queries.

Procedures:
```
swtm_group_id_from_template { { template "" } }
   Returns the group id for the specified (or current) template.

swtm_template_id_from_key { template_key { force 0 } }
   Returns the template_id associated with the specified template key. Memoizes the
   result.
```

```
swtm_template_key_from_id { template_id { force 0 } }
   Returns the template_key associated with the specified template id. Memoizes the
   result.

swtm_template_from_user_id { db user_id }
   Returns the template user_id is associated with
```

5.3.1.5 Pseudo Dynamic Binding in Tcl

One of the key technical problems in building an extensible system is to allow dynamic binding of variables to ensure that changes made to one procedure do not affect others. The basic idea is to allow each called procedure to:

- Take an optional number of arguments that can change dynamically
- Allow each called procedure to inherit the most recently defined value of a variable.

In Tcl, we implement this through a simple procedure call, `swtm_dynamic_binding`, which binds a key to its value in the calling environment. This is possible in Tcl because of its ability to dynamically bind local variables to a variable in the calling environment. To retrieve the most recently defined value of a variable, we use the `swtm_upvar` call to search through the calling stack to find the last definition of a variable, returning that value.

To implement dynamic binding, we explicitly force all procedures to be used with SWTM templates to take no required arguments. Instead, parameters are set in the calling environment, and the called procedure looks for the argument it needs by using `swtm_upvar`.

Procedures:

```
swtm_upvar { var { default_value "_swtm_undefined_value" } }
   Implements dynamic binding of variables by returning the value of the variable
   named var in the lowest found level. If the value is not found, returns
   default_value (or [swtm_undefined_value]).

swtm_dynamic_binding { key value }
   Dynamically binds key to value 2-3 levels up in the calling stack.
```

5.3.1.6 Performance

SWTM provides two procedures devoted to caching information from the database:

- swtm_memoize_list – stores the rows and columns returned as a list in memory
- swtm_memoize_one – returns the first column from the sql query

Neither of these functions requires a database handle, meaning that some pages in the system will only hit the database once after server startup, after which even a database crash would not break the page (though a web server restart would, as the cache would be cleared).

Procedures:

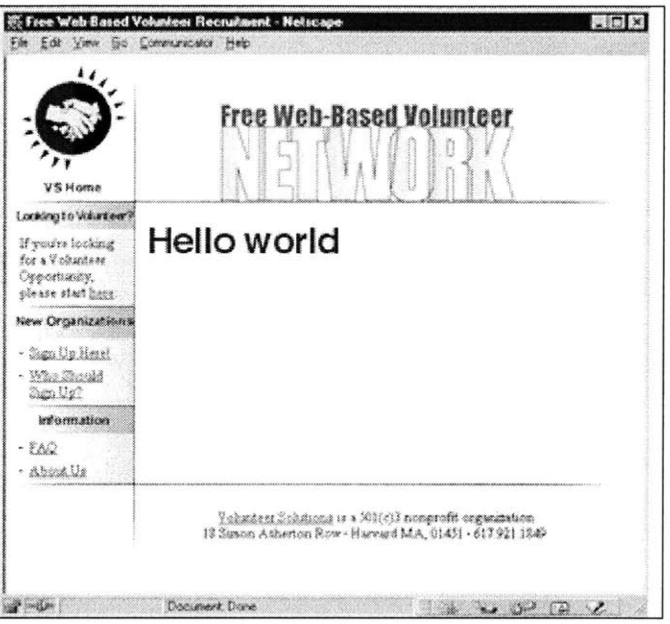
```
swtm_memoize_list { sql_query { force 0 } {also_memoize_as ""} }
   Allows you to memoize database queries without having to grab a db handle first. If
   the query you specified is not in the cache, this proc grabs a db handle, and
   memoizes a list, separated by [_swtm_default_divider] inside the cache, of the
   results. Your calling proc can then process this list as normally.

swtm_memoize_one { sql { force 0 } { also_memoize_as "" } }
   wrapper for swtm_memoize_list that returns the first value from the sql query.
```

5.3.1.7 Serving pages

Once a template has been set up with nodes and objects for that node, writing pages is very simple. The example in Table 2 shows the Tcl code for a web page that simply displays "Hello world" and the presentation of the page. Note that we defined `page_title` in the calling environment, allowing the API calls `swtm_header` and `swtm_footer` to determine which objects must be processed to serve the current page (or the API call `swtm_return_template` in the second Tcl script example in Table 2).

```
Set page_title "Hello world"

ReturnHeaders
ns_write "
[swtm_header]
[swtm_footer]
"
```

```
Set page_title "Hello world"

ns_return 200 text/html \
[swtm_return_template]
```


Free Web-Based Volunteer Recruitment - Netscape

File Edit View Go Communicator Help

VS Home

Looking to Volunteer?

If you're looking for a Volunteer Opportunity, please start here.

New Organizations

- Sign Up Here!
- Who Should Sign Up?

Information

- FAQ
- About Us

Free Web-Based Volunteer NETWORK

Hello world

Volunteer Solutions is a 501(c)3 nonprofit organization
18 Simon Atherton Row - Harvard MA, 01451 - 617.921.1849

Document: Done

Table 2: Simple "Hello world" page (2 ways to write the Tcl page) and the output (served by SWTM)

The API calls do all the work figuring out which objects to use to serve the current page. Both `swtm_header` and `swtm_footer` are very similar in execution, differing only in the type of objects that each returns (`swtm_header` returns objects registered as headers for the current node, and `swtm_footer` returns objects registered as footers).

The call to `swtm_header` proceeds as follows:

1. Call `swtm_object_tags_for_url` to obtain a list of all the objects, in their specified calling order, that generate the header for this template
   a. Obtain the current template using `swtm_template`
   b. Obtain the current url from the web server
      i. Look up the url in the node manager to find the `node_id`
      ii. If there is no `node_id`, find the node id of the parent directory.
      iii. If no node could be found for the current template, and we have reached the root directory, change the template to the system default and begin again with the originally requested url.
   c. Once we have the current node and template, simply look up all the ADP tags for the objects that need to be called.
2. For each object tag, serve that object using the procedure call `swtm_serve_object`
3. Return the concatenated result of all the objects served.

The process for `swtm_footer` is identical. Note that most of the work in serving pages can be divided up into two pieces:

1. Figuring out which template and node to use – we do this through recursive lookups as outlined above, and cache the result so the next time this node is requested, the lookup is fast.

2. Serving each of the registered objects – each object is served with a call to `swtm_serve_object`. This procedure parses each object in the following manner:

    a. Lookup and cache the basic information for the object (its content, required user state (e.g. only for logged in users), and whether or not this object is a Tcl program).

    b. If we're using the URL method to maintain template information, insert the template into every `<a href>` tags that references the server root directory.

    c. Parse the content using AolServer's ADP parser – This parser will recursively parse all other objects in the body of our current object through a registered tag designed to allow objects to include other objects. (See Section 5.3.2: Graphics Designer's API).

    d. Evaluate the object with our Tcl parser, if necessary.

Procedures:
```
swtm_header { { template "" } { url_stub "" } { force 0 } }
    Generates the header for the current or specified template and url_stub. If force
    is set to 1, we skip the cache.

swtm_footer { { template "" } { url_stub "" } { force 0 } }
    Generates the footer for the current or specified template and url_stub. If force
    is set to 1, we skip the cache.

swtm_return_template { }
    Adds the partner header and footer around the string page_content (or page_body)
    that is defined in the calling environment.

swtm_serve_object { tag { template "" } { force 0 } }
    Takes in a tag of an object and returns the html output for that object. This is
    the main procedures used in generating templates for nodes. Note that this
    procedures parses the swtm_set tag to implement dynamic binding. Arguments: *
    template: If non-empty, we'll use it for the template * force: If force is set to
    1, we will ignore caching

swtm_serve_var { tag }
    Simulate adp registered procedure for swtm_var
```

5.3.2 Graphics Designer's API

For the graphics designer, we create the following tags to make it easy for the designer to work with SWTM:

- `<swtm_header` *`options`*`>`*`page_title`*`</swtm_header>` - a wrapper for the procedure call `swtm_header` that dynamically binds the `page_title` and any key/value pairs specified in *`options`* (e.g. size=1). Note that both *`page_title`* and *`options`* are optional arguments.

- `<swtm_footer` *`options`*`></swtm_footer>` - a wrapper for the procedure call `swtm_footer` that dynamically binds and any key/value pairs specified in the optional argument *`options`* (e.g. size=1).

- `<swtm name=object_key>` - Includes the SWTM object with key `object_key`. This tag is registered to the procedure `swtm_parse_tag_swtm` that pulls out the object name and serves the specified object with a call to `swtm_serve_object`.

- `<swtm_var name=var_name` *`default=default_value`*`>` - Looks up the variable named `var_name` with a call to `swtm_serve_var` that looks for the variable in several different places: (Note that *`default_value`* is an optional argument, and if specified, this value overrides any messages provided by the debugging mode.)

    1. Looks up the value of `var_name` for the current file being processed.
    2. Calls `swtm_upvar` to find the last binding of the variable
    3. If the variable is still not found, returns `default_value`, if specified, an error message if in debugging mode, or the empty string.

- `<swtm_set name=`*`var_name`* `value=`*`var_value`*`>` - Dynamically binds a variable named *`var_name`* to the specified value.

A graphics designer could have produced the Hello world example page in Table 2 by writing the following two lines in a file saved in the same directory:

```
<swtm_header>Hello world</swtm_header>
<swtm_footer></swtm_footer>
```

Finally, we note the following objects loaded are used very frequently in all types of objects and in cascading style sheets:

- `<swtm name=default_font size="-1">` (or `[swtm_default_font "size=-1"]` for programmers) – This command inserts an html font tag using the specified properties and the `default_font_face` and `default_font_size` fields for the current template.
- `<swtm name=title_font size="-1">` (or `[swtm_title_font "size=-1"]` for programmers) – This command inserts an html font tag using the specified properties and the `title_font_face` and `title_font_size` fields for the current template.

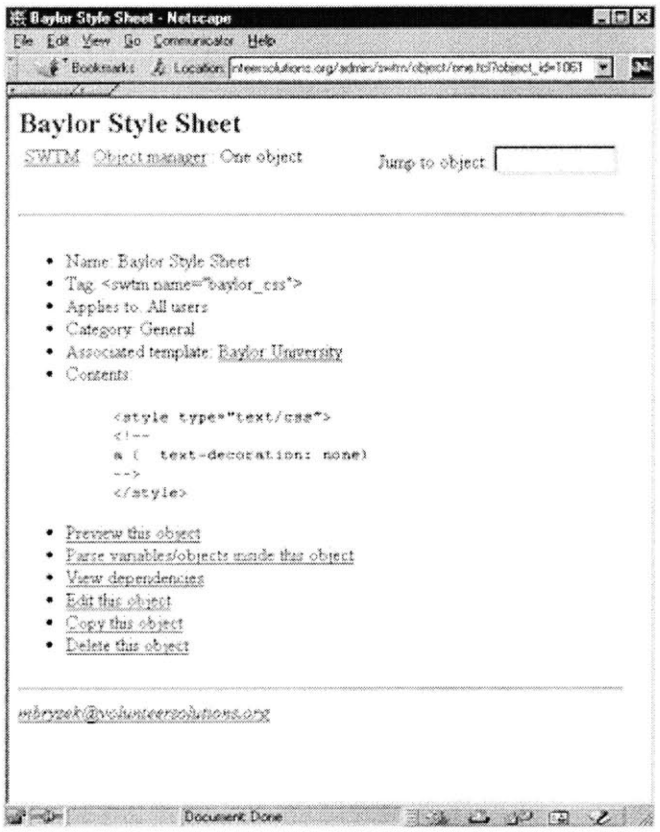
Procedures:

```
swtm_var { var {template ""} {force 0} }
   Caches and returns the value of the specified variable for the current template
   (unless specified). If force is 1, we reset the cache for the template, and then
   return the value of the variable. Returns [swtm_undefined_value] if there is no
   such variable, or template

swtm_var_or_default { var { template "" } }
   Returns the value of var for the specified (or current) template. If the value is
   empty, looks for the same var in the default template.

swtm_default_font { { props ""  } }
   Wrapper for swtm_parse_tag_default_font

swtm_title_font { { props ""  } }
   Wrapper for swtm_parse_tag_title_font
```

5.3.3 Supporting Cascading Style Sheets

Initially, we considered adding a separate module to automatically create a cascading style sheet from whatever fields were specified. However, as we implemented SWTM, we realized that it was much simpler to create, maintain, and extend style sheets by simply creating an object that contained the style sheet. Other objects that needed to display style sheets can simply include the CSS object.

This is an extremely simple solution – there is no additional user interface and no additional code required. The graphics designer simply uses the core tools to enable CSS. Figure 14 shows a simple style sheet used in the site-wide template for Baylor University at VolunteerSolutions.org.


Baylor Style Sheet - Netscape
File Edit View Go Communicator Help
Bookmarks Location: nteersolutions.org/admin/swtm/object/one.tcl?object_id=1061
Jump to object
Document: Done

Baylor Style Sheet

SWTM : Object manager : One object

- Name: Baylor Style Sheet
- Tag: <swtm name="baylor_css">
- Applies to: All users
- Category: General
- Associated template: Baylor University
- Contents:

```
<style type="text/css">
<!--
a {  text-decoration: none}
-->
</style>
```

- Preview this object
- Parse variables/objects inside this object
- View dependencies
- Edit this object
- Copy this object
- Delete this object

mbryzek@volunteersolutions.org

Figure 14: A Simple Style Sheet Object for Baylor University

5.4 Implementation Summary

The SWTM implementation is built upon a general data model that we expect to extend as the needs for various types of new templates becomes clearer. We have focused on creating an API that is rich enough to support all foreseeable needs of a programmer building a site with SWTM while supporting a very simple, and small (3 tags!) API for graphics designers.

6 Evaluation - Did We Accomplish Our Goals?

We began this paper by outlining the following major goals for SWTM:

1. Separating the majority of the tasks between the programmer and graphics designer
2. Maintaining, and hopefully decreasing, overall development time
3. Ensuring relatively simple ports from legacy systems to ones that use SWTM.

In this section, we evaluate how SWTM satisfies each of these goals. As a basis for evaluating our work:

- We fully implemented SWTM at VolunteerSolutions.org, maintaining approximately 20 site-wide templates
- Ported an Online Bookmarks[26] module
- Ported the user pages of an Online Education System to SWTM

6.1.1 Separating the Work

One of the distinguishing characteristics between SWTM and content management systems is that we are only concerned with the tasks of building and maintaining the headers and footers, as opposed to all elements and content in each web page. SWTM achieves the separation by introducing a simple API consisting of two basic calls, `swtm_header` and `swtm_footer`, that generate the appropriate template based on the page being called. The API is simple and increases the legibility of web pages as there is much less, if any, html associated with look and feel. Instead, almost all of the code for each file is directly related to performing some specific function. This makes the separation a success from the point of view of a programmer.

However, to make the separation a true success, we must be able to rely on the graphics designer to work independently with SWTM. To this end, we introduced the SWTM Object Manager as the primary playground for graphics designers. This web based interface allows the designers to continue working with the tools they use today to generate html mockups for websites, synchronizing their changes with the online system whenever necessary.

In practice, we found that designing a complex template without the assistance of a programmer is either impossible or overly tedious. Some of the most common elements in a website are tough to design without control logic. For example, many sites use menu bars that dynamically change based on some state. There are two ways to create these menu bars:

1. Create a unique html mockup of each menu bar and associate one menu bar with each file
2. Create a short script that loops through the options for the menu bar, and highlights the appropriate section, or sections, based on some state (e.g. a variable named `section`).

The second option is much more favorable in terms of future maintenance of the menu bar. However, SWTM does not provide the graphics designer with any assistance in creating such dynamic menu bars. Note that this is a problem common to all templating interfaces, and not just to SWTM.

Instead of trying to "solve" this problem by introducing a new ADP tag or by adding an attribute to an existing html tag, we integrated the TCL scripting language with SWTM objects and provided support for object reuse. Our goal here is not to isolate the programmer completely from the process, but as much as is helpful. Once a programmer creates an object, such as a menu bar, it is saved in the Object Manager.

[26] For more information about the module we used, please refer to http://arsdigita.com/doc/bookmarks.html.

The next time that menu bar needs to be used, the programmer does not have to be involved. Additionally, we provide an online tool for a graphics designer to specify the metadata (e.g. what section to highlight) through the web, allowing the designer to at least modify the objects being created.

There are a couple objective measures to how well we accomplished our goal of separating the work:

1. Percentage of objects that required a programmer's assistance to create
2. Amount of object reuse in the system

6.1.1.1 Objects that required a programmer's assistance to create

The number of objects that require a programmer's assistance is only a rough measure of the level of separation of the work between the programmer and graphics designer. This number will surely be impacted by the type of templates being developed. If the majority of objects need programmer assistance, then we truly have not accomplished our goal.

We implemented 20 site-wide templates for VolunteerSolutions.org, creating 76 objects. Of those, approximately 20 objects required some sort of programming – from simple if/then statements to calls to other Tcl procedures. We note, however, that all of the objects that required programming were created while building the first four templates – all other templates made use of the pre-defined components.

The following types of objects were easy for graphics designer to create independently:

- Images based navigation bars with image maps
- Cascading Style Sheets
- JavaScripts
- General, static, menu bars

The following types of objects were difficult and generally required the programmer's assistance:

- Text-based navigation bar that highlights current section
- Objects that behave differently depending on the URL
- Objects that append additional html based on the existence of variables

Overall, we found that after creating a core set of dynamically changing objects, the graphics designer was able to easily reuse those objects in building new templates.

6.1.1.2 Overall Object Reuse

Object reuse is perhaps the best measure of our ability to remove the programmer from the graphics design process. We can measure object reuse in two ways:

1. The number of objects that use other objects, leading to a hierarchical approach to building templates
2. The number of objects that are used in multiple nodes.

6.1.1.2.1 Hierarchical approach to building templates

On VolunteerSolutions.org, we found that 54 objects were directly reused by other objects. This number, however, is a lower limit. To better understand what this number means, we look at the following example of object reuse:

- One object stores the CSS for a site

- Another object stores the JavaScript
- We build a header object that includes both the CSS and JavaScript objects and some additional html
- Four other objects use the header object

The total number of directly reused objects in this example would be six – The header uses the CSS and JavaScript objects, and four other objects use the header object. In reality, the number of reused objects is twelve – the CSS, JavaScript, and Header objects are each reused four times.

In summary, out of 76 objects, 54 were reused directly. This means that at least 70% of all the objects we create are reused by other objects.

6.1.1.2.2 Reusing objects in nodes

SWTM also supports object reuse at the node level. We registered 77 nodes with the 20 VolunteerSolutions.org site-wide templates, and registered 304 objects with those 77 nodes. This means that on average each of the 76 objects was registered approximately four times with a node.

6.1.1.3 Summary

The level of object reuse in 20 nontrivial VolunteerSolutions.org templates is extremely high, supporting our initial design goal of building a simple, object-oriented approach to creating templates. Part of the reason for this high level of reuse can be attributed to a type of self-fulfilling prophecy – after the first few templates have been created, the designer knows which objects are readily available and can create future mockups to take advantage of the growing library of objects to create new templates.

When actually building these templates, the programmer was involved in creating objects for only the first four of them. The last sixteen templates were created from:

- HTML Mockups
- Previously created objects
- The three SWTM tags provided to Graphics Designers

We thus claim that SWTM accomplishes, at least to a first-degree, the separation of work between the programmer and graphics designer in creating site-wide templates.

6.2 Maintaining Development Time

For SWTM to make sense as a tool, it must not increase the amount of time either the graphics designer or the programmer spends doing the same work they do today. We hope that working with SWTM actually decreases development time, but we would settle for no change in development time because of the added benefits of increased reliability and simpler maintenance.

There are a few aspects to measuring development time:

- How long does it take to create a template with SWTM?
- How long does it take to code a new page?
- How hard is it to change the template in a non-trivial way?

6.2.1.1 Creating a new template with SWTM

The general process of creating a new template is the same regardless of tools being used:

1. HTML mockups are created
2. Parties agree to the state that each page maintains (e.g. the section of the menu bar to highlight)
3. Templates are added to the web site.

With SWTM, the process is no different. However, SWTM provides two real advantages:

1. All objects are stored in the Object Manager and can be reused.
2. The programmers API is constant – because of dynamic binding, the procedure calls in the individual files never need to be changed.

If the template requires a new piece of state to be associated with certain files, there are two options:

1. If there are relatively few pages that are affected, the new state can be assigned through the File Manager
2. If the changes are global, the programmer needs to revisit every affected file to set the state.

Regardless of the system to maintain templates – whether it be straight html or fully blown content management systems, these file-by-file changes will have to be made. SWTM is no different. In practice, we found that the amount of state that each file needs to maintain is both small and relatively constant, and if it does change, there is no additional work imposed by SWTM that did not exist before hand.

On VolunteerSolutions.org, once the core set of objects was created, each new site-wide template took approximately 30 minutes to put together from the html mockup. That is hardly any time at all when you consider that each template affects over 100 files.

6.2.1.2 Coding a new page

Once the templates are created, coding a new page is extremely simple. However, this is true of most templating systems – once the html mockup is complete and somehow made part of the system, nothing more needs to be done.

Note, however, that many systems of including templates in files may require more lines of code to be written. ArsDigita standard coding style is one example of this. Table 3 contrasts the two styles and shows that using SWTM actually reduces the number of lines of code from five to two.

ArsDigita Style	SWTM Style

`ReturnHeaders`
`ns_write "[ad_header $page_title]`
`<h2>$page_title</h2>`
`$context_bar`
`<hr>`
	ReturnHeaders
	ns_write [swtm_header]

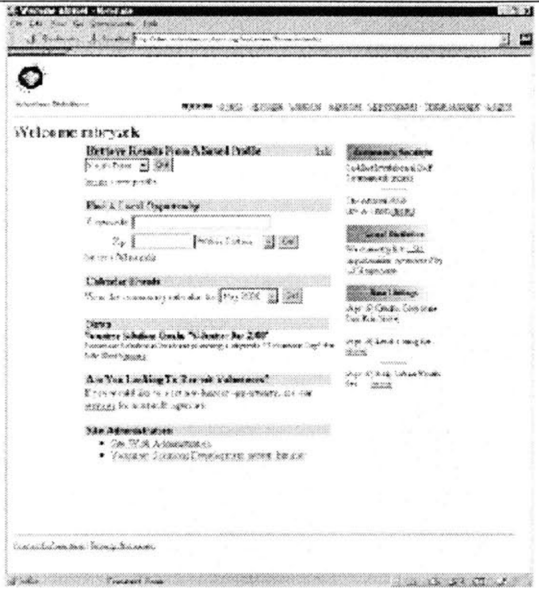
Table 3: Comparison of ArsDigita Coding Style with SWTM Style

The only claim that we can truly make about the amount of time it takes to code a new page while using SWTM is that SWTM requires at most the same amount of time as whatever was in place before.

6.2.1.3 How hard is it to change the template in a non-trivial way?

Changing a template in a non-trivial way is akin to creating a new site-wide template and depends on the amount of new state (e.g. a new variable defining a subsection of a menu bar) and the number of new objects that the programmer must create. To get a grasp on this issue of creating non-trivial new templates, we implemented, for VolunteerSolutions.org, a complex template for SouthWest Texas State

University (See Figure 15 and Figure 16). In this case, the graphics designer was able to completely implement the SouthWest Texas State template, with no programming, in less than half an hour. The interesting point here is that although the template looks completely different, and contains much more complex html, the existing objects for elements such as menu bars allowed the graphics designer to independently make the non-trivial change.



Welcome mbryzek

Retrieve Results From A Saved Profile

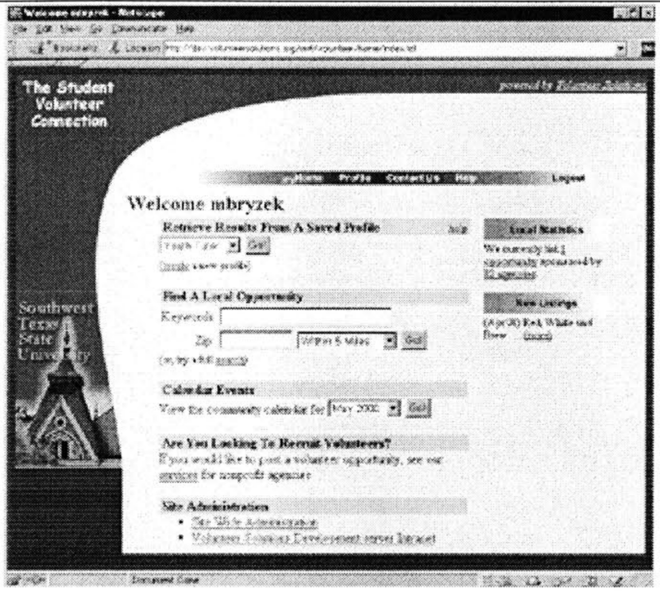
Find A Local Opportunity

Calendar Events

Are You Looking To Recruit Volunteers?

Site Administration

Figure 15: Volunteer Solutions Template for /volunteer node



The Student Volunteer Connection

Welcome mbryzek

Retrieve Results From A Saved Profile

Local Statistics

Find A Local Opportunity

Keywords

Zip

New Listings

Calendar Events

View the community calendar for May 2000 Go

Are You Looking To Recruit Volunteers?

Site Administration

Figure 16: SouthWest Texas State Template for /volunteer node

6.3 *Ensuring simple upgrades/ports*

Our final major goal is to ensure that it does not require too much programming time to install SWTM on an existing website. This is important as once site-wide templates are in use, we want them to be used on all pages, including those previously developed.

6.3.1 Porting the Bookmarks module

To measure our success in this area, we first ported the user pages of a simple ACS Bookmark module to use SWTM. This consisted of porting approximately 20 Tcl pages consisting of about 2,000 lines. Overall:

- The port took less than half an hour
- We modified approximately 125 lines of existing code
- All the modifications were simple – a global find and replace perl script took care of most of the work.

Table 4 shows an example of the type of changes we made to each of the files in the Bookmarks module. The entire process was simple and error-free, though admittedly tedious in that each file had to be manually modified.

Original file (`create-folder.tcl`)	SWTM port (`create-folder.tcl`)

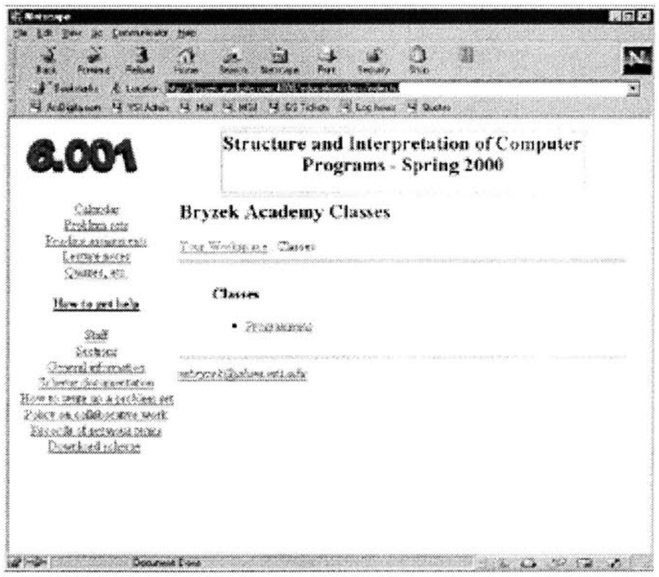
`set title "Create Folder"`
 	
`set html "[ad_header "$title"]`
`<h2>$title</h2>`
 	
`[ad_context_bar_ws "$return_url \`
`  [ad_parameter SystemName bm] " "$title"]`
 	
`<hr>"`
	`set page_title "Create Folder"`
	`set context_bar [ad_context_bar_ws \`
	`  "$return_url [ad_parameter SystemName bm] " "$page_title"]`
	
	`set html [swtm_header]`

Table 4: Example of the type of code we replaced while porting the Bookmarks module to SWTM

6.3.2 Installing SWTM on the Sloan Education Network[27]

We were initially planning to implement SWTM on top of a class run using SWTM. However, there are no classes currently using the Sloan Education Network, and thus most of the examples we presented in this paper came from VolunteerSolutions.org.

We ported the core user pages for the Sloan Education Network (SEN), but did not port other modules used by SEN (e.g. bulletin board, news, etc) that are currently being ported to some kind of templating system by ArsDigita.[28] The core user pages were again simple to port, requiring about a half hour of work. Figure 17 shows a sample template created from the existing MIT Course 6.001 website.


6.001

Structure and Interpretation of Computer Programs - Spring 2000

Bryzek Academy Classes

Classes

Figure 17: Sample SWTM template implemented on central class page for SEN

The key points to take away from the port of the user pages of SEN are:

- Integrating SWTM with the existing pages was simple and fast.
- Once integrated, all other templates, including those created for the other sites presented as examples in this paper, could be dynamically loaded.

[27] Many thanks to Randy Graebner and Aileen Tang who were concurrently developing this system!

[28] The actual templating system that will be used is still being debated. However, once these modules have been rewritten to make use of any set of organized templates, it will be much simpler to either port them to SWTM or integrate SWTM with the templating system being used. In their current state, these modules have no common structure and porting them to any templating system involves a near full re-write, which is beyond the scope of this paper.

The second point is the most important – in less than an hour, we were able to port the user interface for a system consisting of over 20,000 lines of code to SWTM.

6.4 Summary

SWTM accomplishes our goals through:

- Object reuse and dynamic binding allows for easy implementation of new templates and for simple modification of existing templates
- A simple programmer's API ensures that programmers do not need to spend any time thinking about the templates while writing individual user pages
- Support for simple upgrades of existing modules.

Although we were unable to completely separate the tasks of the programmer and graphics designer, we have come a long way to ensure that the two work independently, and have done so with minimal interference to either's design environment.

Finally, we note that some of the work that still requires the interaction of both the programmer and graphics designer is an ongoing problem in web design that we did not set out to solve. Our goal was to separate the majority of the work between the two, and to this end, SWTM is a success.

7 Future Work

As with any system, SWTM is the first version of a templating system that can head in many directions. Our goal was to separate a document into three parts: the header, the contents, and the footer. The future work we describe here broadly outlines several directions/enhancements that we believe will further the utility of SWTM.

7.1 Current SWTM Limits

In implementing the current version of SWTM, we identified several limits that might be well worth removing in future implementations.

- **Support for a root templates directory:** Currently, every new template is assigned a template key that is then registered at the level of the server root (e.g. `http://guidestar.org/template_key`). For a site with many templates, it is foreseeable that one of the template keys will conflict with an existing directory on the system. An alternative design would force all templates to be relative to a specific directory (e.g. `http://guidestar.org/templates/template_key`) rather than from the server root. This may be a desirable option for sites with many templates.

- **Better support for the debugging mode:** Currently, the debugging mode is toggled on or off on the server level, which can become inconvenient when many people are using the site. An enhanced debugging mode would:
    - Allow authorized users to dynamically toggle debugging mode on a user-by-user basis
    - Include a more advanced html parser to provide better feedback on poorly constructed or improperly formatted objects and templates.

- **Node level variables:** We support values associated with fields at the file and template level. It would be nice to allow users to associate these fields at the node level as well (e.g. all pages at the /volunteer node default to a `page_title` of "Volunteer Services").

- **Support for template copying:** On some sites, the differences between some templates may be so small that it is easier to program the logic for the different templates and simply associate new keys with the same nodes and objects. One example of this situation is a site like yahoo.com that sets up site-wide templates for major cities that differ only in the image displaying the city name (e.g. http://boston.yahoo.com). In this case, assigning a template key that doubled as the name of the image to display would allow one template to support all the nodes and objects for all major cities. To make this work, we would have to add a "copy template" feature.

- **Better display of object dependencies:** We currently display only limited object dependencies – just other objects and templates either directly accessed or accessed through another object directly related to the initial object. A useful debugging tool would show all of the dependencies, regardless of depth.

- **Increased number of presentation types:** SWTM only supports two presentation types: Boolean and text. More types, such as integer, character, long text, or super long text, would allow the user to cater the presentation of new fields more closely to the type of field it is. It might be useful to incorporate a fully blown metadata system[29] to manage SWTM fields, though it is not yet clear that such flexibility is needed.

- **Changing keys of objects/fields:** Currently there is no way to change the key associated with an object or field. Since we already track dependencies between objects, it is feasible that a future version will allow the key to be updated.

7.2 Future Enhancements

7.2.1 Usability

One of the major obstacles to the adoption of SWTM is the ability of non-programmers to successfully interact with the system. Though we have made great progress in this initial version, the non-programmer will still struggle in creating some very common site components. For example, SWTM does not provide an IF statement for the graphics designer to use, making it difficult for a graphics designer to create an object for a menu bar that highlights one part of the bar based on the current file or node being accessed.

The challenge with future usability enhancements is to balance the need for certain features with the simplicity of the language that graphics designers use. The way to proceed along these lines is to work with several graphics designers to implement templates based on SWTM, and then to re-analyze the work produced and to understand the major stumbling blocks. The common struggles will point to features that must be implemented to enhance usability.

7.2.2 Better File System Integration

Currently, SWTM editors must work through a browser interface to create or to modify objects. For a site with a large number of templates, this can quickly become tedious. New functionality should be built that:

1. Allows the graphics designer to upload a new version of an object from their own computer

2. Provides the designer with some sort of local copy of a set of objects (e.g. all the objects that this template and node depend on).

This type of functionality would make it easier to create objects as the developer can work locally with third-party software packages and later synchronize work.

Another useful enhancement would be a "file system verifier" to check for nodes whose underlying directories no longer exist and for files which have been removed. This system could offer the user feedback on missing nodes and files, and offer the option to remove them from SWTM completely.

7.2.3 Module Search

A nice feature would allow the user to use keyword searches within SWTM modules. An example of this enhancement would be a keyword search through all of the objects, allowing the designer to find all objects that mention the word "header." This would also be useful with templates and fields.

7.2.4 Versioning

One of the major drawbacks to the current system is that there is no version history of an object. Rather, once an object is modified and refreshed in the cache (e.g. through the preview function), it will become active on the site.

[29] ArsDigita built such a system to support the knowledge management requirements of Siemens.

What is needed is a system of versioning that allows users to:

1. Save all versions of all objects
2. Retrieve any version of an object
3. Mark a particular version as the currently active one

We can provide versioning in one of two ways:

1. Implement a simple web-based versioning procedure
2. Tie the web interface to existing version-control software (e.g. CVS)

From the two options, working with a third party program would be simpler to implement, more reliable, and more powerful. Thus, we expect a future version to cleanly tie the web-editing process to a program such as CVS.

8 Conclusion

There are several different options available today for creating templates for a site, ranging from simple include files to complex content management systems. SWTM fills a niche somewhere in between these two systems, addressing the problem of managing templates to generate the overall look-and-feel of a site while ignoring issues concerning the placement and appearance of individual elements within the page.

We began the project with the following goals:

- Separate work of the programmers from that of the graphics designer
- Make it easy to create, maintain, and modify all of the different templates in use
- Decrease overall development time
- Allow for minimal work in upgrading legacy systems.

The first goal proved to be the most challenging in that many templates in use today require some programming logic to be successful. SWTM does not provide any mechanism to enable the graphics designer to create this logic and continues to rely on the programmer to create objects that require some sort of programming.

SWTM successfully accomplishes the second goal by creating an object-based approach to building html templates that is intuitive to graphics designers. Objects can dynamically bind variables (through a simple ADP tag), include other objects, and look up the value of any variables. Additionally, all objects are stored in a central Object Manager that allows the graphics designer to choose from a library of previously created objects. Thus, when a graphics designer requires assistance from a programmer to create a new object, that object will be stored and easily reused in future templates.

Through a simple programmer's API, SWTM usually decreases development time in terms of creating new pages. The API abstracts the templates being used, and furthers the treatment of individual web pages as objects with state.

Finally, we ported a couple modules of differing complexity to SWTM – the process is simple, though somewhat tedious, when porting a well-designed module (modules written in an inconsistent fashion are extremely difficult to port to SWTM).

In practice, SWTM truly takes advantage of object reuse, both on the level of new objects making use of previous objects and of templates using multiple objects to generate the look and feel for a portion of a site. Graphics designers can work independently, and programmers need not worry about the templates.

As discussed in Section 7:Future Work, there is still a lot of work to be done. The true test will come when a graphics designer is single-handedly managing hundreds of site-wide templates for a web site. Only at that time will we have truly accomplished our goals.

9 Appendix A: Oracle 8i Data Model

```
-- define a table that everything else references...
-- these are the identifiers for the templates themselves!
create sequence swtm_template_template_id_seq start with 100;
create table swtm_template (
       template_id          integer
                            constraint st_template_id_pk primary key,
       -- a human understandable name for this template - for UI
       template_name        varchar(250)
                            constraint st_template_name_nn not null,
       -- the unique identifier for this template
       template_key         varchar(50)
                            constraint st_template_key_nn not null
                            constraint st_template_key_un unique,
       -- a lot of times, we will want to associate a user group with a template
       -- we assume that the user will use a group_type of "site-wide template"
       -- from which to select the groups. Note that the user group must first be
       -- created before the user can associate it with a template.
        -- note that a user group can have at most one template associated
        -- with it
       group_id             constraint st_group_id_fk references user_groups
                                 constraint st_group_id_un unique,
       -- we allow "symlinks" between templates - that is, one template can use all of
       -- the objects specified for another template
       copy_from_template_id      constraint st_copy_from_template_id_fk references
swtm_template,
       -- is this node currently active?
       active_p             char(1) default('f')
                            constraint st_active_p_ck check(active_p in ('t','f'))
);


-- a centralized table to simply store all the adp tags we create.
-- This is important as it keeps the tags unique across the entire system
create sequence swtm_tags_tag_id_seq start with 100;
create table swtm_tags (
       tag_id                        integer
                            constraint st_map_id_pk primary key,
       -- each tag must be uniquely mapped to another object
       -- this is the ADP tag that our users will use as they create objects
       -- and programmers use in referencing template variables
       tag                  varchar(50)
                            constraint st_tag_un unique
                            constraint st_tag_nn not null,
       on_which_table                varchar(50)
                            constraint st_on_which_table_nn not null,
       on_what_id           integer
                            constraint st_on_what_id_nn not null,
       -- because each tag is unique, we only allow each row in a table to be mapped
       -- once. You may want to relax this constraint to allow for multiple tags to
       -- refer to the same table and row, but I don't think this is a really  good
       -- idea
       constraint st_which_table_what_id_un unique(on_which_table, on_what_id)
);


-- a table to put every object we create into a category
create sequence swtm_category_category_id_seq start with 100;
create table swtm_object_category (
       category_id          integer
                            constraint soc_category_id_pk primary key,
	category		varchar(500)
				constraint soc_category_nn not null
);


-- this table stores all of the template objects we create
create sequence swtm_object_object_id_seq start with 100;
create table swtm_object (
	object_id		integer
				constraint so_object_id_pk primary key,
	-- into what category can we stick this object?
	category_id		constraint so_category_id_fk references
swtm_object_category
				constraint so_category_id_nn not null,
	-- what is the name of this object - should be human understandable
	object_name		varchar(250)
				constraint so_object_name_un unique
				constraint so_object_name_nn not null,
	-- what ADP tag do we use?
	tag_id			constraint so_tag_id_fk references swtm_tags
				constraint so_tag_id_un unique
				constraint so_tag_id_nn not null,
	-- what's the actual content? This is an adp...
	content_adp		clob
				constraint so_content_nn not null,
	-- a human readable description of what the thing does
	description		clob,
	-- does this object apply to all users, logged_in users, or logged-out users?
	required_user_state varchar(20) default('all')
				constraint so_required_user_state_ck
check(required_user_state in ('all','logged-in','logged-out')),
	-- do we subst the content of this object?
	tcl_template_p		char(1) default('f')
				constraint so_tcl_template_p_ck check(tcl_template_p in
('t','f')),
	-- does this object belong to only one template or do we
	-- share it with the entire system?
	template_id		constraint so_template_id_fk references swtm_template,
	-- some columns to tracke the object creator and dates
	creation_date		date
				constraint so_creation_date_nn not null,
	creation_user_id	constraint so_creation_user_id_fk references users
				constraint so_creation_user_id_nn not null,
	last_modified_date	date default sysdate,
	last_modified_user_id		constraint so_last_modified_user_id_fk references
users
				constraint so_last_modified_user_id_nn not null,
	last_modified_ip	varchar(50) constraint so_last_modified_ip_nn not null
);


-- we keep track of which objects reference which other objects
-- note that one object can have at most 1 reference in this table to another
-- object because we don't really care about the number of references
create table swtm_object_object_map (
	from_object_tag		varchar(50) constraint soom_from_object_tag_nn not
null,
	to_object_tag		varchar(50) constraint soom_to_object_id_nn not null,
	constraint soom_from_to_object_tag_pk primary key(from_object_tag,
to_object_tag),
	constraint soom_from_to_not_equal_ck check(from_object_tag <> to_object_tag)
);
```

```
-- we need a way to say we only care about headers and footers
create sequence swtm_type_type_id_seq start with 100;
create table swtm_type (
       type_id              integer
                            constraint st_type_id_pk primary key,
       type_key             varchar(50)
                            constraint st_short_description_un unique
                            constraint st_short_description_nn not null,
       explanation          varchar(2000)
                            constraint st_explanation_nn not null
);



-- types of fields - text/boolean/etc.
create sequence swtm_field_types_type_id_seq start with 100;
create table swtm_field_types (
       type_id                    integer
                            constraint sft_type_id_pk primary key,
       pretty_text          varchar(100)
                            constraint sft_pretty_text_nn not null,
       type_key             varchar(50)
                            constraint sft_type_key_un unique
                            constraint sft_type_key_nn not null
);


-- we need an easy way to associate multiple fields with each node
-- create a table to store all the fields for which we want to ask
create sequence swtm_field_field_id_seq start with 100;
create table swtm_field (
       field_id             integer
                            constraint sf_field_id_pk primary key,
       type_id                    constraint sf_type_id_fk references swtm_type
                            constraint sf_type_id_nn not null,
       -- text to display when asking for people to enter this field
       display_text         varchar(1000)
                            constraint sf_field_text_nn not null,
       -- unique template_key for an ADP tag to insert this data
       -- into the page
       tag_id               constraint sf_swtm_tag_fk references swtm_tags
                            constraint sf_swtm_tag_un unique
                            constraint sf_swtm_tag_nn not null,
       required_p           char(1) default('f')
                            constraint sf_required_p_ck check(required_p in ('t','f')),
       field_type_id        constraint sf_field_type_fk references swtm_field_types
                            constraint sf_field_type_nn not null,
       -- what to display if they forget to enter this field and it's required?
       display_if_not_entered    varchar(1000),
       constraint sf_required_display_ck check( required_p='f' or
display_if_not_entered is not null ),
       -- what's the default value for this field?
       default_value        varchar(1000)
);


-- a place to store information for all fields
create table swtm_field_value (
       -- this is the value for which field?
       field_id             integer
                            constraint sfv_field_id_fk references swtm_field,
        -- which row does this field value refer to?
        on_which_table              varchar(50)
                             constraint sfv_on_which_table_nn not null,
        on_what_id           integer
                             constraint sfv_on_what_id_nn not null,
        -- the value itself
        value                varchar(1000),
        constraint sfv_table_what_id_field_id_pk primary key (on_which_table,
on_what_id, field_id)
);



-- we need a table to store all the nodes in the system
-- and metadata on those nodes. We use the url stub with a
-- leading slash and no trailing slash to
--   a. uniquely define a node
--   b. let us implement inheritance on the filesystem (through caching,
--      we can afford the performance hit on this one...)
create sequence swtm_node_node_id_seq start with 100;
create table swtm_node (
        node_id                     integer
                             constraint swtm_node_node_id_pk primary key,
        -- what's the url_stub of this node (relative to server root)
        url_stub             varchar(200)
                             constraint swtm_node_url_stub_un unique
                             constraint swtm_node_url_stub_nn not null
);


-- we need a table to store meta-information for any file.
-- we simply create a table to store files, just like nodes,
-- and then use the abstract field/value tables
-- note that files are also system-wide... This is probably
-- desirable as most templates share file information. If not,
-- then you better use unique variables in the objects you
-- create for the templates that need different field/values!
create sequence swtm_file_file_id_seq start with 100;
create table swtm_file (
        file_id                     integer
                             constraint swtm_file_file_id_pk primary key,
        node_id                     constraint swtm_file_file_id_fk references swtm_node
                             constraint swtm_file_file_id_nn not null,
        -- what's the path to this file, including the filename itself
        -- and relative to the node url stub. Includes leading slash
        file_name            varchar(400)
                             constraint swtm_file_file_name_nn not null,
        constraint swtm_file_node_file_un unique(node_id, file_name)
);


-- we need to map nodes to templates so that we maintain unique nodes
-- and unique templates, with unique pairings
create sequence swtm_node_map_id_seq start with 100;
create table swtm_node_map (
        map_id                      integer
                             constraint snm_map_id_pk primary key,
        on_which_table              varchar(50)
                             constraint snm_on_which_table_nn not null,
        on_what_id           integer
                             constraint snm_on_what_id_nn not null,
        node_id              integer
                             constraint snm_node_id_fk references swtm_node,
        constraint snm_table_id_node_un unique(on_which_table, on_what_id, node_id),
        -- is this mapping currently active?
        active_p            char(1) default('f')
                            constraint snm_active_p_ck check(active_p in ('t','f'))
);


-- we need to map each template/node pair to an object and object type...
-- note that there is no restriction saying an object cannot appear twice
create sequence swtm_node_object_map_id_seq start with 100;
create table swtm_node_object_map (
        node_object_map_id  integer
                            constraint snom_node_object_map_id_pk primary key,
        map_id              integer
                            constraint snom_map_id_nn not null
                            constraint snom_map_id_fk references swtm_node_map,
        object_id           integer
                            constraint snom_object_id_nn not null
                            constraint snom_object_id_fk references swtm_object,
        -- What "role" is the object playing here? That is, in regards to its being
        -- used for this node mapping, what is it?
        object_type_id              constraint snom_object_type_id_nn not null
                            constraint snom_object_type_id_fk references swtm_type,
        -- if we have multiple objects defined for a given node mapping, we probably
        -- want a way to indicate the order in which the objects should be applied
        calling_order       integer default 1
);
```

10 Appendix B: Loading Initial Templates

This section lists the SQL scripts to load a default template into SWTM.

```
-- Define a user group type of "site-wide template."
-- this is more for user-inteface concerns because we don't want to
-- offer a select list of 1,000 groups!
BEGIN
   insert into user_group_types
     (group_type, pretty_name, pretty_plural, approval_policy,
default_new_member_policy, group_module_administration)
   values
     ('site-wide template', 'Site-wide template', 'Site-wide templates', 'closed',
'closed', 'none');
END;
/
show errors;


-- create a few initial categories
insert into swtm_object_category
(category_id, category)
values
(1, 'General');

insert into swtm_object_category
(category_id, category)
values
(2, 'Menubar');

insert into swtm_object_category
(category_id, category)
values
(3, 'Headers');

insert into swtm_object_category
(category_id, category)
values
(4, 'Footers');

insert into swtm_object_category
(category_id, category)
values
(5, 'Miscellaneous');

insert into swtm_object_category
(category_id, category)
values
(6, 'Style Sheets');

insert into swtm_object_category
(category_id, category)
values
(7, 'JavaScript');



-- populate the types now
insert into swtm_type
(type_id, type_key, explanation)
values
(1, 'header', 'Header objects are used to create the templates for the page header');
```

```
insert into swtm_type
(type_id, type_key, explanation)
values
(2, 'footer', 'Footer objects are used to create the templates for the page footer');

insert into swtm_type
(type_id, type_key, explanation)
values
(3, 'template_field', 'Template fields are used to specify additional information for
each template');

insert into swtm_type
(type_id, type_key, explanation)
values
(4, 'object_field', 'Object fields are used to specify additional information for any
new template object that is created');

insert into swtm_type
(type_id, type_key, explanation)
values
(5, 'file_field', 'File fields are used to specify additional information for each
file');


-- Populate several common field types
insert into swtm_field_types
(type_id, pretty_text, type_key)
values
(1, 'Boolean (i.e. true/false)', 'boolean');

insert into swtm_field_types
(type_id, pretty_text, type_key)
values
(2, 'Text (e.g. any text description, field, or other information)', 'text');


-- SETUP A DEFAULT TEMPLATE WITH SOME BASIC PROPERTIES

-- create an initial, default template
insert into swtm_template
(template_id, template_name, template_key, active_p)
values
(1, 'ArsDigita Default Site-Wide Template', 'ad', 't');

-- create the root node
insert into swtm_node
(node_id, url_stub)
values
(1, '/');


-- Create two simple objects to wrap around api calls to generate ArsDigita look and
feel
insert into swtm_tags
(tag_id, tag, on_which_table, on_what_id)
values
(1, 'ad_header', 'swtm_object', '1');

insert into swtm_tags
(tag_id, tag, on_which_table, on_what_id)
values
(2, 'ad_footer', 'swtm_object', '2');
```

```
insert into swtm_tags
(tag_id, tag, on_which_table, on_what_id)
values
(3, 'insert_default_font', 'swtm_object', '3');

insert into swtm_tags
(tag_id, tag, on_which_table, on_what_id)
values
(4, 'insert_title_font', 'swtm_object', '4');

insert into swtm_tags
(tag_id, tag, on_which_table, on_what_id)
values
(5, 'default_font_color', 'swtm_field', '1');

insert into swtm_tags
(tag_id, tag, on_which_table, on_what_id)
values
(6, 'default_font_face', 'swtm_field', '2');

insert into swtm_tags
(tag_id, tag, on_which_table, on_what_id)
values
(7, 'default_title_color', 'swtm_field', '3');

insert into swtm_tags
(tag_id, tag, on_which_table, on_what_id)
values
(8, 'default_title_font', 'swtm_field', '4');



insert into swtm_object
(object_id, category_id, object_name, tag_id, content_adp, description, creation_date,
creation_user_id, last_modified_date, last_modified_user_id, last_modified_ip)
values
(1, 3, 'ArsDigita Default Header', 1, '<%=[swtm_generic_header]%>', 'Wrapper for the
default ArsDigita header - call to ad_header followed by the page title, context bar,
and horizontal rule', sysdate, 1, sysdate, 1, '0.0.0.0');

insert into swtm_object
(object_id, category_id, object_name, tag_id, content_adp, description, creation_date,
creation_user_id, last_modified_date, last_modified_user_id, last_modified_ip)
values
(2, 4, 'ArsDigita Default Footer', 2, '<%=[swtm_generic_footer]%>', 'Wrapper for the
default ArsDigita footer', sysdate, 1, sysdate, 1, '0.0.0.0');

insert into swtm_object
(object_id, category_id, object_name, tag_id, content_adp, description, creation_date,
creation_user_id, last_modified_date, last_modified_user_id, last_modified_ip)
values
(3, 1, 'Insert Default Font', 3, '<%=[swtm_parse_tag_default_font]%>', 'Inserts the
default font html tag specified in the template', sysdate, 1, sysdate, 1, '0.0.0.0');

insert into swtm_object
(object_id, category_id, object_name, tag_id, content_adp, description, creation_date,
creation_user_id, last_modified_date, last_modified_user_id, last_modified_ip)
values
(4, 1, 'Insert Title Font', 4, '<%=[swtm_parse_tag_title_font]%>', 'Inserts the title
font html tag specified in the template', sysdate, 1, sysdate, 1, '0.0.0.0');
```

```
--  Create the fields for default and title font
insert into swtm_field
(field_id, type_id, display_text, tag_id, required_p, field_type_id)
values
(1, 3, 'What is the default font color?', 5, 'f', 2);

insert into swtm_field
(field_id, type_id, display_text, tag_id, required_p, field_type_id)
values
(2, 3, 'What is the default font face?', 6, 'f', 2);

insert into swtm_field
(field_id, type_id, display_text, tag_id, required_p, field_type_id)
values
(3, 3, 'What is the title font color?', 7, 'f', 2);

insert into swtm_field
(field_id, type_id, display_text, tag_id, required_p, field_type_id)
values
(4, 3, 'What is the title font face?', 8, 'f', 2);



-- map the root node to the default template
insert into swtm_node_map
(map_id, node_id, on_which_table, on_what_id, active_p)
values
(1,1,'swtm_template',1,'t');


-- map our objects to the default template in their specified roles
insert into swtm_node_object_map
(node_object_map_id, map_id, object_id, object_type_id)
values
(1,1,1,1);

insert into swtm_node_object_map
(node_object_map_id, map_id, object_id, object_type_id)
values
(2,1,2,2);
```

11 Bibliography

1. Abelson, Hal, Professor of Computer Science and Engineering, Massachusetts Institute of Technology, Email exchange and conversations with the author, January 2000 to May 2000.

2. ArsDigita Community System Documentation, http://www.arsdigita.com/doc, specifically http://arsdigita.com/doc/versioning.html, http://arsdigita.com/doc/style.html, and http://arsdigita.com/doc/bookmarks.html.

3. Dixon, Paul, Head of InterMedia Division at Oracle, Conversation with the author, January 2000.

4. Goldstein, Karl, "Dynamic Publishing," http://karl.arsdigita.com/doc/publish/.

5. Graebner Randy and Aileen Tang, MIT Thesis on "Sloan Education Network."

6. Greenberg, Bernard S., "Multics Emacs: The History, Design, and Implementation," August 15, 1979, http://www.multicians.org/mepap.htm.

7. Greenspun, Philip, *Philip and Alex's Guide to Web Publishing*, April 1999, Morgan Kaufmann Publishers; ISBN: 1558605347.

8. Johnson, Rhonda, *United Way of Massachusetts Bay*, conversations with the author, November 1999 to May 2000.

9. Network Solutions, Inc., "What is HTML?", http://rrpac.upr.clu.edu:9090/~jcarroll/html/sld02.html.

10. Raggett, Dave, "Adding a touch of style," February 17, 2000, http://www.w3.org/MarkUp/Guide/Style.

11. Reid, Brian, "20 Years of Abstract Markup. Any Progress?" Compaq Computer Corp, Nov 19, 1998, http://reid.org/~brian/markup98.html.

12. Simmons, Wendy, GuideStar.org, Conversations with the author from January 2000 to May 2000.

13. Stallman, Richard, "EMACS, the Extensible, Customizable, Display Editor," February 11, 1998, http://org.gnu.de/software/emacs/emacs-paper.html.

14. Taggart Rip, GuideStar.org, Conversations with the author from September 1999 to May 2000

15. Tufte, Edward R., *Visual Explanations : Images and Quantities, Evidence and Narrative,* March 1997, Graphics Press; ISBN: 0961392126.

16. Valdés, Ray, "A Plateful of Templates," *WebTechniques*, May 2000, Volume 5, Issue 5.

17. Vimuktanon, Atisaya, GuideStar.org, Conversations with the author from September 1999 to January 2000.

18. Wium, Håkon and Bert Bos, "Cascading Style Sheets – Designing for the Web," http://www.awlonline.com/cseng/titles/0-201-41998-X/liebos/.

19. World Wide Web Consortium, "Web Style Sheets", http://www.w3.org/Style/.

20. "CSS Frequently Asked Questions", http://www.hwg.org/resources/faqs/cssFAQ.html#css.