Project Code: MXC-XXXX

World Wide Web Application Layout Tool

A Major Qualifying Project Report

Submitted to the Faculty

of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the Degree of Bachelor of Science

in Computer Science

by

Eric Willisson

August 25, 2011

Project Adviser, Professor Michael Ciaraldi

Abstract

The World Wide Web makes it possible for programs to be run on computers without needing to be installed or classically downloaded, but the interfaces remain bound by text-centric page layout languages. This project attempts to bring fully-featured GUI (Graphical User Interface) technology to the hands of Web developers by interpreting the output of Glade, a GUI design program, as HTML which can be published on the Web.

Acknowledgments

I would like to thank Professor Michael Ciaraldi for providing being a truly helpful adviser, for a project I have wished for since entering WPI.

I would also like to thank my parents, Wendy Rowe and Pace Willisson, for getting me started on Computer Science in the first place, and for giving the support I needed at the very end to finish.

Finally, I would like to thank my wonderful fiancée, Rhiannon Chiacchiaro, who made the final days of the project bearable.

Contents

List of Figures

1 Introduction

Computer programs are the foundation of the Information Age. A computer is defined as a machine designed to carry out a sequence of logical operations, where the specific sequence may be readily changed. The problem of how to load a particular sequence of operations, or program, onto a specific computer has existed as long as computers themselves have, and has led to many attempted solutions. In the present, a program which is intended to run on as many different computers as possible must be written to handle different operating systems, processor architecture, and even hardware peripherals.

The Internet has provided an alternative method for computations. Web browsers provide relatively uniform environments for programs to be run in, as well as connections back to servers, which allow complicated computations to be carried out in a known system, with only the output sent to a unique computer. The Hyper Text Transfer Protocol (HTTP) provides a standard protocol for communication between the client and server, while the Hyper Text Markup Language (HTML) augmented by Cascading Style Sheets (CSS) define platform-independent display standards. Finally, the JavaScript scripting language allows programs to be executed on the client computer. These tools were originally written to support the paradigm of static "pages" of content on the World Wide Web. As the Internet has grown and evolved, Web pages now may support full applications, with complicated user interfaces.

Although there are now many examples of complete applications written using HTML, CSS, and JavaScript, such as Google Documents, the languages are not particularly well-suited to writing applications as opposed to pages of text and images. A number of "web authoring" tools have begun to be developed, allowing non-programmers to create full pages. However, these tend to either be focused on writing text pages as well, or on certain types of web design, such as writing Adobe Flash content.

This project aimed to create a program to aid in using HTML, CSS, and JavaScript to build a complete Graphical User Interface (GUI) to a program written either in JavaScript or in any language on the hosting web server. It should allow both programmers and artists who are unfamiliar with programming to design the interface, and allow subsequent updates to the interface without

needing changes to the rest of the program.

The project is built to interpret the output of an existing tool, the Glade User Interface Designer. Glade[1] is an open source program written for designing GUIs for the GIMP Tool Kit (GTK+) library[2]. It is based around placing "widgets," user interface elements, in a canvas, and exporting a file which represents the layout which has been designed. Glade is a stable and mature program, which makes it a good choice for the basis of this new project.

This paper covers the need for the project and the method in which it was implemented. A brief history of the creation of computer programs and the ways they have been delivered to users explains the unique position of the Internet and the World Wide Web in programming today. The design decisions which had to be made are explained, including the reasons behind each choice of technology. Potential work for the future is outlined at the conclusion of the report.

2 Background

2.1 Programming Languages

A computer program is, at its most basic, a series of instructions which, when executed sequentially, perform an algorithm using a computer's hardware. The types of algorithms being implemented have varied greatly over the decades since the first von Neumann computers were built, but the purposes and methods have remained fairly unchanged.

The only form of commands a modern computer can execute are binary machine instructions, which are strings of zeroes and ones. These strings directly instruct the processor on which operations to carry out. However, this language is extremely difficult for humans to understand and write. In addition, the exact meaning of the machine instructions can differ from computer to computer, depending on the CPU architecture and the other software installed on the computer. Machine code now exists primarily as the language programs are translated into by other programs, while humans write in "higher-level" languages.

Many high-level languages, such as FORTRAN, C, and C++, work by being compiled. Compilation is the process in which one or more files written in a human-readable language are translated into machine code by a program called a compiler. The human-readable files, called the source code, can be copied to any computer which has a suitable compiler installed, allowing specifics of the computer's architecture to be handled by the compiler rather than the programmer.

There are several drawbacks to relying entirely on compiled languages. The first is that not all specifics of a computer can be fully abstracted. There is often considerably more effort which must be put in to writing "platform independent" code to be compiled, rather than code which can only be trusted to run on one type of computer. In addition, the compilation process can be slow, even on fast computers. Finally, the executable files containing machine code are significantly larger than the source code.

In order to circumvent some of the drawbacks of compiled languages, computer programmers developed interpreted languages. These languages work by having an interpreter, often a compiled program, ported to as many different types of computers and operating systems as possible. The

interpreter can read in human-readable source code files, and directly execute them, line by line. An interpreter is often much slower than a compiled program, but has much greater flexibility. Most of the problems involving different types of computers are handled by the interpreter, allowing the programmer to focus on writing the program's logic and not spend effort on ensuring it will run on different types of computers.

A third type of language exists, based on interpreted bytecode. These languages, including Sun®Oracle®Java, have a program similar to an interpreter, but instead of directly reading source code, they interpret "bytecode," which has been compiled with a special non-platform-specific program. However, these languages tend to effectively behave as either compiled languages, as in the case of Java, or interpreted languages, such as Python. For this reason, they do not have much effect on the problem this project intends to solve.

2.2 Programming in the World Wide Web

In 1995, JavaScript was first packaged with the web browser Netscape Navigator as a platform-independent interpreted language which could be used, as its name implies, to provide scripting for Java applets, as well as manipulating the HTML of displayed web pages.[3] It went through many revisions, issues of conflicting standards, and difficulty being taken seriously by established programmers. Despite all of these obstacles, however, it has emerged as one of the dominant languages in use today.

JavaScript's main draws come from the ubiquity of its platform. JavaScript interpreters can be found in almost every web browser, making it available on nearly all desktop, notebook, and tablet computers, and even on an increasing number of cellular phones. Its popularity has led to great effort being put into optimizing the speed of executing its source code, and the current version's libraries include functions to make asynchronous calls back to the server hosting the web page, allowing for procedures beyond the scope of an individual web page to be carried out, and their values reported back to be acted on by the JavaScript program. JavaScript has become a language suitable for implementing complete applications, while being executable on a wide variety of platforms.

2.3 Drawbacks of JavaScript

"Web applications," such as Google's Gmail, Google Documents, the game "entangled" by Entanglement, and a great many others, demonstrate that JavaScript is not only capable of supporting complete programs, it is now being widely used for this purpose. However, there are certain disadvantages to this chosen platform.

JavaScript does not handle display on its own. Normally, it provides logic, procedural instructions, and reactions to events, generated by a web page laid out with a combination of HTML and CSS. The development of these two markup languages has driven the majority of the use of the World Wide Web as we see it today. As a result, the early perceptions of how the Web would be used have shaped both HTML and CSS, as well as the protocol used to transmit data back and forth between Web servers and their clients, Web browsers. This protocol, HTTP, is stateless, meaning any information which needs to remain the same between transfers, such as session ID numbers, user names, and configuration options, must be included in each data transfer. HTML and CSS were written from the perspective of describing the layout of text-centric pages, with elements which are either structured as inline text or block elements the width of the entire page, such as titles. The order of elements in a rendered HTML page is based on the way English is read, with small elements proceeding from left to right and larger elements proceeding down. While these techniques are logical for page-based web sites, they are often based on assumptions which do not hold for GUI-based design.

GUIs have been developed independently from the Web, and have many assumptions of their own. Most languages define one or more GUI libraries, to aid programmers in creating windows, buttons, labels, text fields, and other widgets. Each library can define its own methods for handling how elements are laid out, but many conform to box-based sequential layouts, table-based positional layouts, or edge-based floating layouts. All of these are distinctly different from the particular choices made in specifying how HTML elements are rendered. However, they are often preferable when a GUI is being designed. Additionally, there are tools written for layout methods specific to a number of GUI libraries. These tools allow non-programmers to develop the visual aspects of GUIs without needing to know anything about the languages involved. They allow a distinct separation between the part of a program describing what should be shown to the user, and the part of the program which carries out operations depending on the user's interactions with its

interface. Similar intentions can be applied to Web applications laid out with HTML and CSS and programmed with JavaScript, but the tools so far have been much less advanced.

The majority of HTML authoring tools are based around the same paradigm as HTML itself. Adobe®Dreamweaver®, Microsoft®Word's Export to HTML option, and other editors all are structured with the intention of laying pages out for display. They usually support the various input elements which are used in creating GUIs, but achieving the desired layout can be extremely difficult, if it is possible at all.

2.4 Present Tools and Methods

It has become clear that despite the drawbacks of using HTML, CSS, and JavaScript over HTTP as an application framework, this is now the easiest way to write applications which will be accessible by the maximum number of people with the minimum amount of effort necessary on their parts before they can begin to use it. In the competitive world of software engineering, a low barrier to entry is extremely important. As awareness of computer viruses has spread, users have become less willing to install unknown software on their computers, adding to the value of writing for an interpreter already installed on their machine. In 2011, with new standards HTML5 and CSS3 being developed to incorporate more modern GUI features into standard Web design, JavaScript appears to be one of the most effective languages for writing applications.

The remaining choices to be made hinge on how to ensure that programmers have the best possible tools for creating applications using this framework. There are several other, similar methods, which allow more traditional means of programming. Java applets, Adobe®Flash pages, Microsoft®Silverlight documents, and other, less common, systems allow for programs to be written which are accessed through Web pages while giving access to more complete GUI programming tools. However, they all suffer from similar flaws: require special plugins to be installed on the client computers, which do not always conform to known standards, and tend to have much higher computer resource requirements. In addition, they do not communicate well with the browser itself, and so do not allow functions such as text searches, bookmarking, or text-to-speech synthesizers for disabled users. Tools which allow for standard HTML, CSS, and JavaScript to be employed to make fully functional applications add greatly to the usability of the World Wide Web.

2.5 Normal Usage of Glade

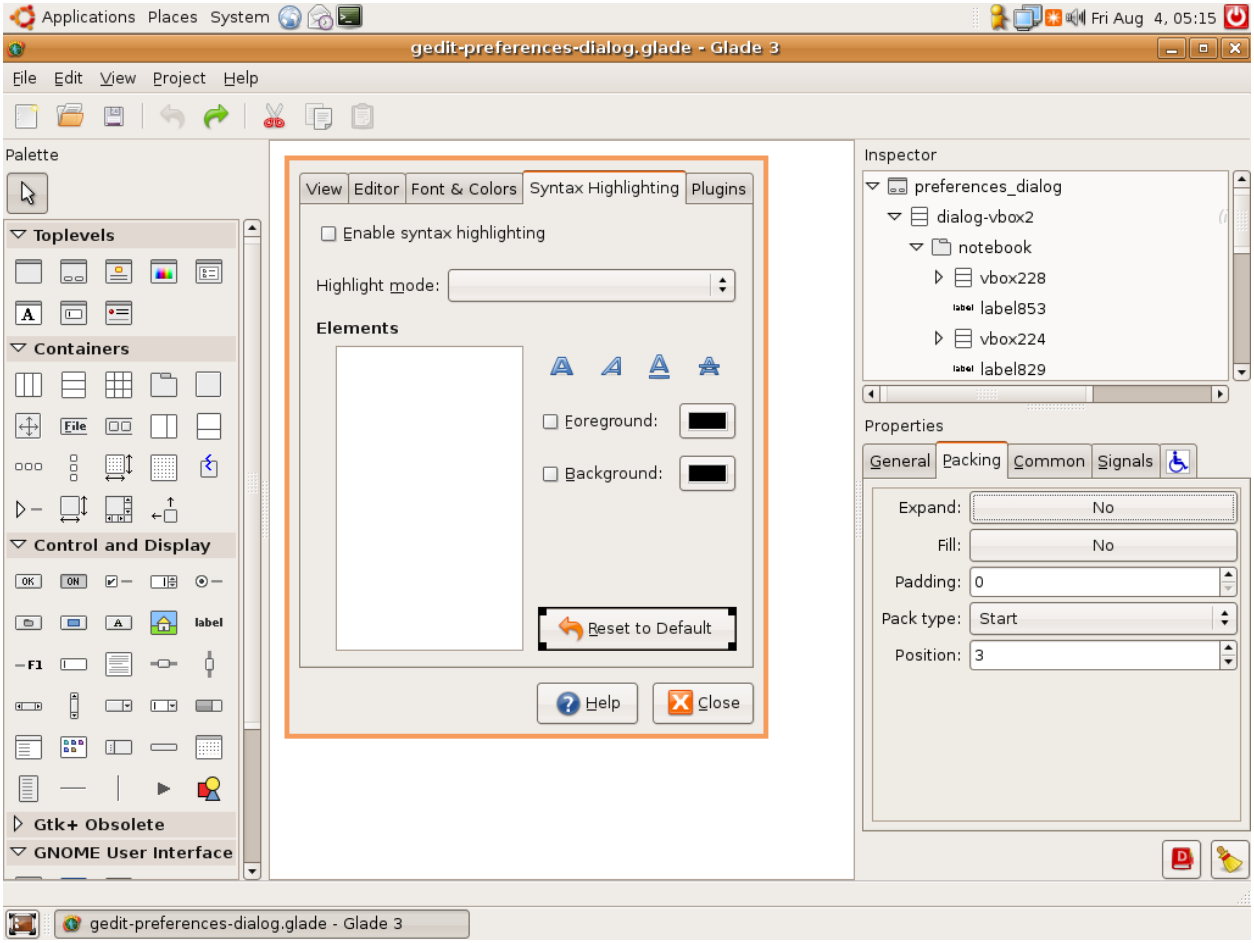

Figure 2.1: Glade window[1]

The Glade User Interface Designer was written to improve the development of GUIs for the GTK+ toolkit. In its current form, it is language-independent, allowing GUIs designed through its interface to be used in any language which supports GTK+.

During normal usage, Glade is launched by a designer. A window opens with a menu, a side panel with buttons representing all the possible widgets, property information about the current file, and a canvas area where widgets can be added to the UI in progress (Figure 2.1.) The interface is designed by selecting containing elements and widget elements, and clicking where they are supposed to appear. The displayed UI looks similar to how it would appear in a program, but not entirely the same. In addition to adding elements, the designer may customize them, by setting style details, and linking signals. One of the tabs in the main window allows strings to be inputted to associate function names with event signal which may be sent to a widget, such as “clicked,”

"keyup," or "delete-event." When the interface has been completed, it is saved through the File menu, as is commonly seen. The file ends with ".glade" by default, and contains GtkBuilder XML data.

To actually use the .glade file, a programmer must write code in a language with GTK+ bindings. C, C++, Java, Python, Perl, and many others all support the toolkit. A specialized function loads the .glade file when given its name, and creates the GUI window which has been designed. Any signals given function names to call in the file are linked, causing events in the window manager to call the functions written by the programmer. This system allows the designer working in Glade to specify which input events the program will need to handle, and the programmer to write the logic triggered when these events occur. The completed program must be distributed with the .glade file, and small changes to that file can be made without recompiling the program itself. Glade allows GUI programming to be separated into two parts, visual design work and program logic.

3 Methodology

3.1 Requirements

The idea for the project originated several years ago, giving the requirements significant time to be developed. When the project was started officially, a set of requirements was decided on to direct the development. Alternatives were considered, but the majority of the requirements remained the same throughout the project.

Many of these requirements were heavily informed by the design decisions, discussed in section 3.5.

3.1.1 Itemized Requirements

The requirements settled on for the project are as follows.

- The program must reproduce an interface designed in Glade as accurately as possible in HTML.
- The program should make it as easy as possible for additional functionality to be added to the pages it outputs.
- The program's outputted HTML must render correctly on as many different browsers as is feasible.
- The program should be straightforward to execute.

3.2 User Stories

In order to help specify the design of the program, several user stories were written. Writing user stories is one of the current methods employed in determining how to write programs. Mike Cohn describes that "a user story describes functionality that will be valuable to either a user or purchaser of a system or software." User stories are intended to be composed of three aspects:

- a written description of the story used for planning and as a reminder

- conversations about the story that serve to flesh out the details of the story
- tests that convey and document details that can be used to determine when a story is complete[4]

Certain user stories had little detail, and described the effects the program would have on the development cycle of Web pages from initial conception to end user views. These were grouped as "high level" stories. Other user stories more specifically explained interactions with the program itself, and were grouped as "Web Developer specific" user stories. This distinction helped conceptualize what kind of tool was being written, as well as how it would work.

3.2.1 High level

- As a Web user, I want to navigate to a Web page with a high-quality GUI.
- As a Web developer, I want to take a GtkBuilder file created by saving a GUI design in Glade, and convert it into a set of HTML, CSS, and JavaScript files which will create a high-quality GUI which can be viewed on the Web.
- As a Glade developer, I want to create a GUI using Glade which can be viewed on the Web.

3.2.2 Web Developer specifics

Here, "user" is not specified, because it has the same meaning in all of the stories. A user in these stories refers to a web developer who intends to use the program to help create a fully functional web page with an interactive GUI.

- As a user, I want to make as few manual changes as possible to the inputted GtkBuilder file and outputted HTML, CSS, and JavaScript files.
- As a user, I want to be able to specify any additional information needed on the Web but not in Glade while executing the translation program.
- As a user, I want to be able to make manual changes without undue difficult to the generated Web pages.
- As a user, I want to be able to run the entire translation process from the command line, a Web site, and local GUI-based program.

3.3 Use Cases

In addition to user stories, the more standard form of project structuring, use cases, were written. Like the user stories, some of these were logically considered "high-level" use cases, while the rest detail a user's actual interaction with the program more specifically. Only these more specific ones were expanded on to create fully-detailed use cases, as the more general ones serve their purpose best by leaving most design details vague.

The Actors listed in the following use cases fall into three categories: Glade developer, Web developer, and Web user. A Glade developer refers to a person who is familiar with using Glade to create user interfaces. This person is expected to understand the various GTK+ widgets and the ways in which they can be customized, as well as the different types of signals which can be linked. A Glade developer does not need to know any programming language, and can transfer the created GtkBuilder files to a Web developer in order to have it be displayed in a Web page.

A Web developer, in these cases, is the type of user who will be expected to directly use the program which has been written for this project. A Web developer should be a person who is familiar with HTML, CSS, JavaScript, and has access to a Web server such as Apache.[5] This user does not need to be familiar with the usage of Glade, and can receive the GtkBuilder files for their Web pages from a Glade developer.

A Web user is anyone who uses a Web browser to access a Web page over the Internet by following a link or specifying a URL. A Web user does not have any required knowledge, and should not interact with a page generated with the help of this project any differently from any other page available on the Web.

3.3.1 Use Case: Create GUI [High-level case]

Actors:

- Glade developer

Description:

The Glade developer creates a GUI in Glade and saves it, creating a GtkBuilder file.

Normal Flow:

1. The user creates a GUI layout in Glade.
2. The user saves the layout, creating a file with the ".glade" extension containing XML conforming to the GtkBuilder schema.

3.3.2 Use Case: Create Web GUI [High-level case]

Actors:

- Web developer

Description:

The Web developer takes a GtkBuilder file describing a GUI and runs it through a program which translates it into HTML, CSS, and JavaScript which can be placed in the Web server.

Normal Flow:

1. The user receives a GtkBuilder file from a Glade developer.
2. The user runs the translation program with the file as the input.
3. The user enters any additional information needed for the Web page into the program.
4. The user writes any additional Web pages and code needed to complete the functionality of the Web page.
5. The user ensures that the Web page is viewable by their target viewers.

3.3.3 Use Case: Access Web page [High-level case]

Actors:

- Web user

Description:

The Web user accesses a Web page and sees the GUI the Web developer programmed.

Normal Flow:

1. The user accesses a Web page, through a link or directly entering a URL.
2. The user is able to view and interact with a GUI reminiscent of one that would be seen on the user's desktop.

3.3.4 Use Case: Create Simple Input Web Page

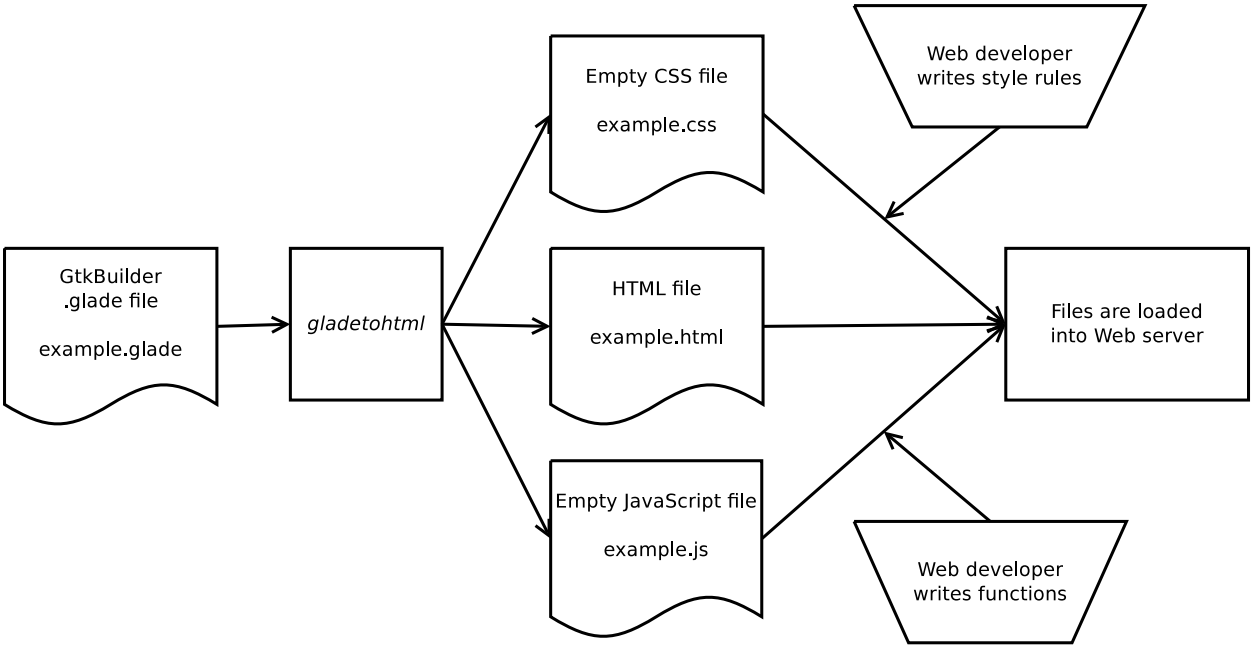

Figure 3.1: Simple Web Page Flow

Description:

The user wishes to create a relatively simple web page, which prompts for personal information that can be submitted for further processing.

Normal Flow:

1. The user has a GtkBuilder file written by Glade describing the desired GUI.
2. The user executes the translation program, inputting the additional data of the web page.
3. The translation program produces an HTML file, and empty files for custom CSS and JavaScript, with names derived from the GtkBuilder filename if not otherwise specified.

4. The user writes the function to handle submission in JavaScript, and puts the function in the page's custom JavaScript.

5. The user publishes the completed page.

Alternative Flow:

1. The user has a GtkBuilder file written by Glade describing the desired GUI.

2. The file has an error in its XML, possibly caused by manual editing or corruption during a file transfer.

3. The user executes the translation program, inputting the additional data of the web page.

4. The translation program detects an error, and outputs it to the user, specifying the position in the file where the error was encountered. The program then halts.

5. The user may attempt to fix the GtkBuilder file, or acquire a new one, and return to the beginning of the use case.

3.3.5 Use Case: Alter CSS or JavaScript of Simple Input Web Page

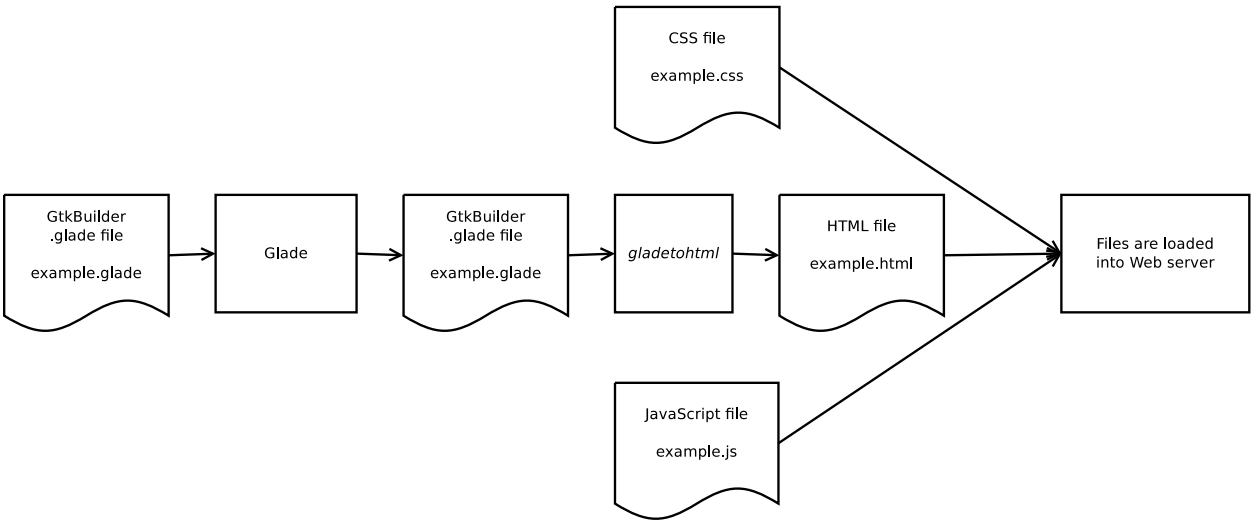

Figure 3.2: Alter Web Page Flow

Description:

The user wishes to alter the CSS or JavaScript of a relatively simple web page, described above, which has already been created.

Normal Flow:

1. The user has a GtkBuilder file written by Glade describing the desired GUI.
2. The user executes the translation program, inputting the additional data of the web page.
3. The user publishes the completed page.

Note:

There is no need for the user to rewrite the JavaScript. While the HTML is overwritten, the user-added JavaScript and CSS are in custom files and are not altered, as is shown in Figure 3.2.

3.3.6 Use Case: Alter GUI of Simple Input Web Page

Description:

The user acquires a new GtkBuilder file for a relatively simple web page, described above, which has already been created.

Normal Flow:

1. The user has the new GtkBuilder file written by Glade describing the desired GUI.
2. The user executes the translation program, inputting the additional data of the web page.
3. The user publishes the completed page.

Note:

There is no need to rewrite the JavaScript. While the HTML is overwritten, the JavaScript and CSS are in custom files and are not altered.

3.4 Detailed Use Case: Creating a complete web page in the command line

Actors:

1. Web developer

Description:

The user acquires a GtkBuilder file describing a GUI.

Normal Flow:

1. The user has a GtkBuilder file written by Glade describing the desired GUI.
2. The user executes the command-line translation program "gladetohtml," and specifies any of the following command-line arguments:
    - *input* - the GtkBuilder file, or a '-' to indicate a pipe from stdin.
    - *output* - the HTML file name, or a '-' to indicate a pipe to stdout. If not specified, the file name will be derived by removing ".glade" from the Glade file's name and appending ".html". The JavaScript and CSS file names are always the same as the HTML file name, with the extension altered to ".js" and ".css" respectfully.
    - *title* - the title of the HTML page, to be placed in the <title></title> tag. If omitted, a filler string will be written instead.
    - *gtkcss* - the path as will be seen by the HTML page to the file containing CSS used by all pages translated from Glade, to cause the HTML elements to look more like GTK+ widgets. If omitted, it will be assumed to be using the file "gtk.css" in the local directory.
    - *jquery* - the path as will be seen by the HTML page to the file containing the jQuery JavaScript library, which is required for the webshims to function correctly. If omitted, will use the URL `http://code.jquery.com/jquery-latest.min.js`
    - *webshims* - the path as will be seen by the HTML page to the directory containing the webshims. Webshims are JavaScript functions which provide backwards compatibility for HTML5 elements. If omitted, it will use the directory packaged with the program,

`webshim/`. The primary webshims used in the program can be retrieved from `http://afarkas.github.com/webshim/demos/`. The latest available version will be packaged with the program. If the directory is not included, the generated page may not render correctly on some older browsers.

- *signal-xml* - the list of signal handlers will be outputted as raw XML instead of a formatted list.
- *help* - Print a description of these items, and any other usage information which is needed.

3. The program writes out the HTML file, and ensures that the CSS and JavaScript file (names described above) exist. It does not overwrite any data found in the CSS or JavaScript files.

4. The user writes any JavaScript necessary to the Web page in the new JavaScript file. If any signal handlers were defined in Glade, the function names specified there should be used as the names of JavaScript functions, which will be called when the corresponding events are triggered.

5. If necessary, the user moves the HTML, CSS, and JavaScript files to the location they will need to reside in to be accessible through the web server.

The command-line arguments are specified according to the standard Linux method, and are interpreted by the getopt library. They may be inputted in any order, separated by spaces, and prefixed with two dashes and the option name, followed by a space and the string meant to be used in the program. For example, to indicate that the title of a Web page should be "Hello, world," the user would type `--title "Hello, world"`. The "help" option does not have any additional data to specify, and is not followed by anything. Finally, the input file may be specified as a final command-line option, with only the filename. Some possible invocations of the program are as follows:

- `gladetohtml --help`

  This would print an informative message.

- `gladetohtml --title "Hello, world" hello.glade`

  This would use the file `hello.glade` to generate `hello.html`, with a title "Hello, world." It would also generate empty files `hello.css` and `hello.js`.

- `gladetohtml --title "Hello, world" --webshims "/scripts/js/webshims" --input "hello.glade" --output "index.html"`

  This would use the file `hello.glade` to generate `index.html` with the title "Hello, world." It would also generate empty files `index.css` and `index.js`. The generated page would expect to find the webshims script files in the `/scripts/js/webshims/` subdirectory of the Web server's document root.

There will also be shorter abbreviations of the command-line arguments as are standard for Linux programs, beginning with a single dash followed by the first letter of the full options. These are provided for convenience, although when seen on the command line they will make the intention the program was executed with less apparent.

Note:

The flow for using *gladetohtml* in a GUI or through a Web page should be very similar, and is not significant enough to warrant separate use cases.

3.5 Design Decisions

The development of the *gladetohtml* program was an ongoing process through most of the actual programming. Many directions and methods had to be considered, and some could only be discarded after actual testing was attempted. Choices which were not employed are documented here with the reasons they were decided against.

3.5.1 Glade

One of the first decisions to be made was what GUI layout framework to use. Glade was chosen. From its own website, "Glade is a RAD [Rapid Application Development] tool to enable quick & easy development of user interfaces for the GTK+ toolkit and the GNOME desktop environment." Glade is unusual in that it does not generate specific code to represent GUI layouts. Instead, it writes XML according to a schema called GtkBuilder. The XML file contains all of the information necessary for a program to render a complete UI in GTK+, including types of elements, positioning, event handlers, and style information. Normally, the GtkBuilder file is read in by a function in a programming language with GTK+ bindings, such as C, Python, Perl, or Java. However, it was

realized that this project could adapt the same techniques to translate the GtkBuilder XML into HTML, CSS, and JavaScript. This allows Glade to be used for design, without a human having to handle the complexities of UI layout in HTML.

No other obvious candidates existed besides Glade for the intended purpose. Most GUI generating tools write code for specific languages, which would greatly increase the complexity needed for the translation program. While Glade was written specifically for GTK+, the GtkBuilder format is general enough that it could be extended for use in other platforms. This project is the first instance of such an extension which was able to be found, but other attempts may follow in the future.

3.5.2 Python

The language Python was chosen due to its ease of use, existing familiarity, its extensive libraries, and the fact that its interpreter has been ported to many platforms. The program does not need to be able to run multiple times per second, so the speed of a compiled language was not necessary.

3.5.3 Execution Time

One major decision which had to be made was at what point the translation should happen. When dealing with the Web, there are three main places and times a program can be executed: loading the server, serving requests, and in the web browser.

Loading the server is done ahead of time. Files are put in the paths which a web server will examine. Programs run at this time can be manually executed, to translate existing files into ones which should be browse-able from the Web. The *gladetohtml* program, if run while loading the server, would have the greatest flexibility in configuration options, as well as the fastest responses, because static files are easy for web servers to deliver to browsers. However, it requires intervention from the user any time the Glade file is altered. If the original GtkBuilder file is changed, *gladetohtml* must be executed on it to generate new HTML, or else the old file will continue to be served.

When serving requests, programs can be executed through the Common Gateway Interface (CGI). CGI scripts are commonly used to generate dynamic web pages, such as outputs of databases. Every time the URL indicating the CGI script is accessed, the script is executed again, and the outputted HTML is sent to the web browser. *gladetohtml* could be set up as a CGI script, allowing changes to the GtkBuilder files to instantly be reflected by new page loads by web browsers.

However, there would be a considerable cost to speed, as the program must run every time the page is accessed. In addition, configuration would be more difficult. The program would run without user intervention, so any options would have to be specified in a special configuration file.

Web browsers themselves usually include an XSLT parser, which can allow an arbitrary XML file to be displayed according to the rules in the XSLT style sheet. W3schools lists the major browsers and the versions of each which support this display option. Mozilla Firefox version 3, Google Chrome version 1, Opera version 9, Apple Safari version 3, and Microsoft Internet Explorer version 6 all are compatible with the official W3C XSL Recommendation, and can safely be expected to reliably display XML with an XSLT style sheet[6]. However, certain mobile browsers, such as the native browser found in Google's Android operating system for mobile phones, do not support XSLT[7] The file `gladetohtml.xsl` could be loaded on the web server, allowing users to directly load URLs for GtkBuilder files from Glade, which would then be translated according to the style sheet. This option would, like the CGI script method, allow updates in Glade to be instantly reflected in the rendered page, but suffers from similar, but greater, difficulties in configuration.

After careful consideration, the first option, in which the program is executed to load static HTML files on to the web server, was chosen. Configuration is highly important, and avoiding CGI scripts makes the program possible to use on web servers with high-volume traffic. In-browser XSLT would reduce customization options, and would not render on many mobile devices.

3.5.4 Interface

Once the choice of execution time had been made, the interface for the program itself had to be decided. For simplicity and power, the immediate choice was a command-line executable. Following the guidelines presented in Eric S. Raymond's *The Art of Unix Programming* allowed a program to be written that needed little effort for the interface itself, while ensuring that it could support many upgrades by acting as a "backend" for more complicated programs. A GUI could be implemented by programming the visual window itself, with spaces for options and a "Run" button. The options would cause command-line arguments to be filled, and the "Run" button would execute the program as if from a command line directly. The Unix philosophy identified by Raymond is "Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface." The program was written with this ideal in mind, allowing for future interfaces to require minimal alterations to the core code.

3.5.5 Accessibility

The standards for HTML and CSS are currently being updated. In 2014, the World Wide Web Consortium (W3C) is expected to officially approve HTML5 and CSS3 as the latest versions of these languages. Already, a number of features associated with them are being implemented in most modern Web browsers. Some of the features closely match aspects of GTK+ which would make translating for *gladetohtml* much simpler. However, not all users on the Web can be expected to have browsers which support these latest technologies.

In order to allow for the use of advanced HTML5 and CSS3 features while still meeting the specified requirement that the generated pages be as accessible as possible, the JavaScript library "Webshims" is being employed. A "shim" in this context is a program that handles incompatible commands and translates them into a form which can be used by a computer. In this case, Webshims detects when a web browser is not capable of displaying HTML5 elements, and uses JavaScript to replace them with HTML that the browser will render correctly. Webshims has multiple components, and requires several libraries to run, so the directory `webshim/` contains the necessary files as well as an initialization script which will be automatically called by the generated page. The remaining required library, jQuery, may need to be updated by a web developer, and so can be specified when `gladetohtml` is executed. While not guaranteed to work, Webshims allow for greater accessibility while still relying on newer technologies.

3.5.6 Structure

Initially, the project was structured under the assumption that Python itself would do the translation from GtkBuilder to HTML. Python has XML input and output libraries, which made it possible to detect tags, interpret how they would be rendered, and generate the corresponding HTML. A network of functions, referenced by keys in a hash table, was written to allow the type of GTK+ widget being specified in GtkBuilder to select which function should compute the HTML. This method allowed code for each widget to be separated from the rest, aiding in the design process.

While writing the pure Python version, further research uncovered XSLT, the XSL Transformations language. XSLT was written to carry out the exact operations being written in Python for this project: transforming one XML document into another. It quickly became clear that it would be much more effective to write an XSLT style sheet describing the translations from GtkBuilder

to HTML, and use Python only to apply the style sheet and execute any other auxiliary commands such as creating the custom CSS and JavaScript files.

The program XALAN, published by the Apache Software Foundation, takes XML and XSLT files as inputs and outputs new, translated XML[8]. After making the decision to switch to using XSLT to describe the actual transformations, the Python program was altered to call XALAN during the running of "gladetohtml," allowing a single XSLT file to be written as a major section of effort going in to the project, independent of extra features added in Python.

Writing the XSLT style sheet itself was a large undertaking. The first challenge was interpreting how GtkBuilder actually describes a GUI, as the documentation, primarily at `http://developer.gnome.org/gtk/stable/GtkBuilder.html#BUILDER-UI`, is not very descriptive. Several simple GUIs were created to analyze their structure.

The initial construction of the XSLT file was based around a template that would identify the root of the GtkBuilder document, and output a basic HTML structure. In the body of the HTML document, an XSLT command reads sequentially through the rest of the GtkBuilder document, causing smaller templates for individual Gtk+ widgets to be executed. This retained the separation of each widget into a small block of XML, similar to the separate functions in the original, pure Python implementation.

One issue which had to be considered was how to handle customizations to the generated web pages. By the nature of the program, the HTML file is overwritten each time *gladetohtml* is run. The solution settled on was to have the generated HTML include references to a CSS and JavaScript file, which the program would create if they did not exist, but otherwise not alter. A user could fill these files in, creating changes in the function and display of the page, without having to make the alterations every time a new Glade file is loaded. Changes to the HTML file itself would have to be repeated, but this should be unusual enough not to be highly necessary. If the functionality is desired, a script could be written to apply *gladetohtml* and then make further edits to the HTML.

Two additional aspects had to be included in order to meet all the necessary requirements of the program. The first was a method to input the HTML page's title, variable pathnames, and any other optional data which might be needed. The first method looked at was to use special XSLT tags to create comments in the HTML, which could then be read by the Python program and replace with the correct values. A string at the beginning of the comment could be identified by a regular expression, and the remainder of the comment would be treated as a key to a hash table

containing options set by the user, such as "TITLE". On further consideration, this method was altered slightly, to just put in direct XML tags in their own namespace, which could be detected by Python, without need for comments which would not show up in other XML parsers. As a result, the final string signifying the location that the title chosen by the user should be written into the HTML is "<title><g2h:extern>TITLE</g2h:extern></title>." Inside attributes, where tags cannot be placed, the syntax $\extern[...] was chosen, allowing the hash table key to be specified in between the square brackets. As with the XML tags, a regular expression in Python is able to identify the string to replace. In both of these cases, the control string exists briefly in the Python string containing the full page, until a function replaces them with the values which Python has calculated. When the HTML file is written out, it only contains the actual strings which should appear in the Web page.

The second change to the initial construction of the XSLT file was the inclusion of second and third passes through the GtkBuilder document. The first pass which was written translates each object into corresponding HTML. However, this pass did not have any way to handle the CSS or JavaScript specific to each element. A separate pass through the file for both CSS and JavaScript allowed for different data to be extracted from the same object declarations, creating specific CSS rules concerning width, borders, and other style information, and event handlers to correspond to signals declared in JavaScript. The HTML file is now built up sequentially by the XSLT parser. The three passes are diagrammed in figures 3.3, 3.4, and 3.5, with the final result in Figure 3.6. The actual execution is handled by XALAN, and is not touched by the program itself. In `gladetohtml`, the full HTML file, after all three passes, is returned as a string which is stored in a variable and then manipulated before being saved to a file on disk.

Each pass required its own challenges to overcome. The initial method of interpreting how properties should be translated to CSS rules consisted of one template, which contained a large <xsl:choose> statement, similar to a switch() statement in many other languages. Which CSS rule should be written was determined in the <xsl:choose> statement by examining the GTK+ class of the current object. It was eventually realized that this method circumvented XSLT's natural element-identifying procedures. The single template was rewritten into one for each property, similarly to how the templates identifying GTK+ widgets were set up. For maximum control over the layout, each element was given an HTML tag with a class name derived from the GTK+ widget class, and an identifier derived from the object's own "id=" property. This extra tag allowed

for the “expand” and “fill” properties to be correctly implemented. The tag was given a class named “gladecontainer” to allow special defaults to be set for all of these elements as well. This class was recognized as being necessary when a discrepancy in the horizontal alignment of elements between GTK+ and HTML was noticed. A final section of the CSS rules contains a special template to handle any properties which cannot be detected normally. In particular, detecting the lack of a “visible” property, which indicates a widget should be hidden, had to be placed in this last template.

Writing the JavaScript pass used many of the techniques learned while writing the CSS pass. The same initial <xsl:choose> method was employed until the more efficient one using XSLT’s own styles was recognized. JavaScript required an extra step because the functions mentioned in Glade might not be implemented in the Web page itself. A try/catch block was added around all signal handlers to ensure that ReferenceErrors triggered by naming a function which has not been written would not interrupt the rest of the JavaScript’s execution. The JavaScript pass also required a final section, similar to the one written for the CSS pass. However, its function was to include code necessary for some HTML objects to display correctly. The GtkLinkButton had to have a “click” event registered which would send the user to the URL it references, and the GtkHScale widget, as well as other variable-number interfaces, had to have correct display code programmed in.

The final necessary component was a special loop detecting all of the signal handlers defined by the Glade file. This loop was used to write an extra block of XML data into the bottom of the web page, between the close of the <body> tag and the <html> tag. The data is identified by a surrounding <g2h:signals> tag, allowing the Python program to identify it. Python extracts the data from the HTML file and removes it from the XML tree, and prints a message to the command line listing each of the signals found, the object each signal is bound to, and the name of the handler function which the signal will call when triggered. The names may then be used to write functions in the custom JavaScript files to enable the web GUI to respond to user interaction. An option to request that `gladetohtml` print the raw XML data instead of a formatted list was included to help programs built on top of `gladetohtml` make use of the data.

3.5.7 Omissions

Due to time constraints and the difficulty of programming, certain features in Glade were omitted from the final program. A number of GTK+ widgets were not able to be implemented, and would

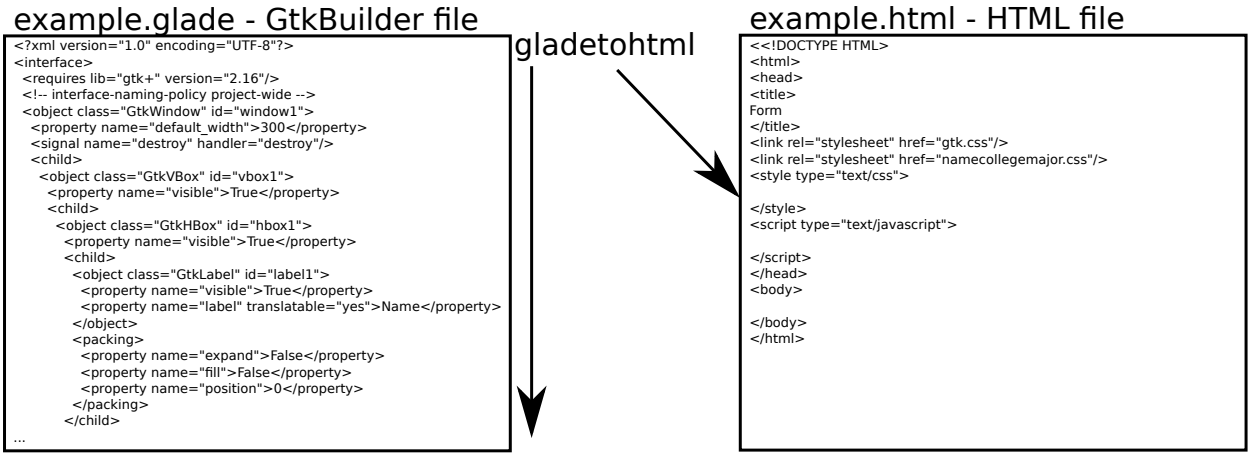

Figure 3.3: First pass over GtkBuilder file, filling in CSS rules.

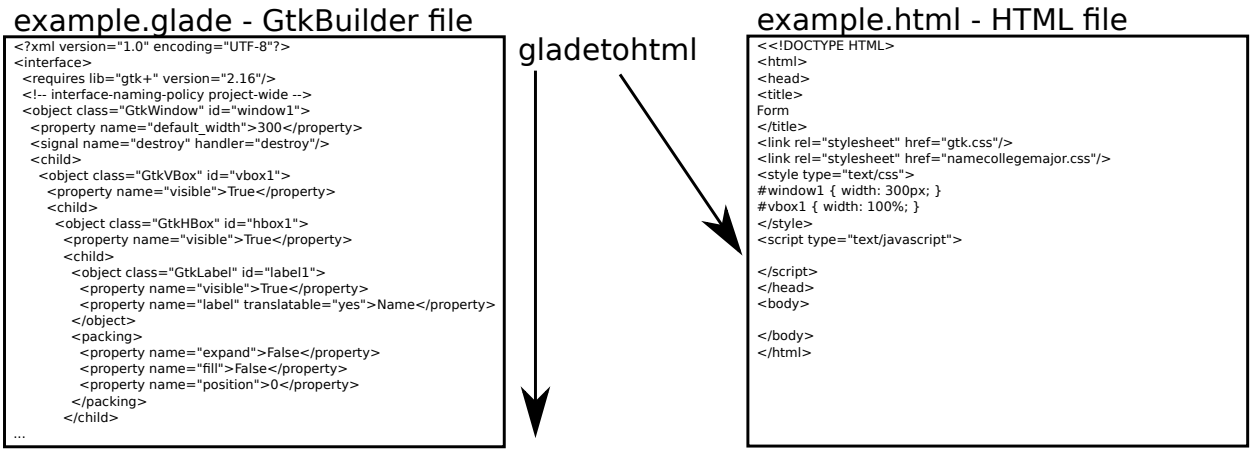

Figure 3.4: Second pass over GtkBuilder file, filling in JavaScript declarations.

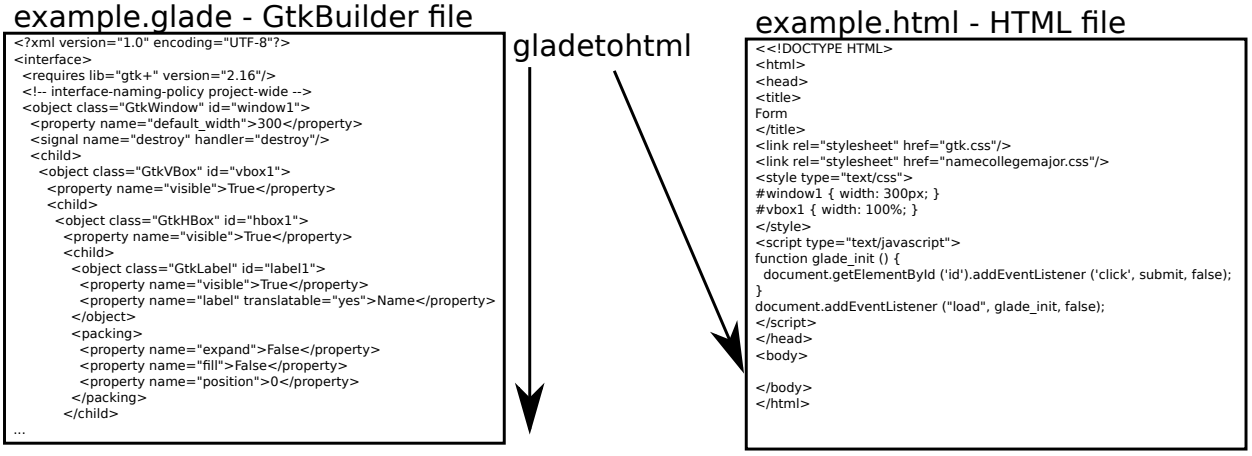

Figure 3.5: Third pass over GtkBuilder file, filling in actual HTML declarations.

example.html - HTML file

```
<<!DOCTYPE HTML>
<html>
<head>
<title>
Form
</title>
<link rel="stylesheet" href="gtk.css"/>
<link rel="stylesheet" href="namecollegemajor.css"/>
<style type="text/css">
#window1 { width: 300px; }
#vbox1 { width: 100%; }
</style>
<script type="text/javascript">
function glade_init () {
  document.getElementById ('id').addEventListener ('click', submit, false);
}
document.addEventListener ("load", glade_init, false);
</script>
</head>
<body>
<div class="GtkWindow" id="window1">
...
</div>
</body>
</html>
```

Figure 3.6: Final state of completed file.

not appear in a page generated from a file which includes them. The only "toplevel window" implemented is a standard GtkWindow, with various forms of dialog boxes left out. Including these would primarily require deciding what exactly they mean in the context of a web browser, where "popups" are generally avoided.

The only container widgets which were able to be implemented were the horizontally- and vertically-aligned boxes. These take sequential widgets and lay them out from left to right or top to bottom. They are the most important containers by far, and the majority of the rest could be implemented as they are needed, focusing primarily on the CSS needed for them to display correctly.

The majority of the functional widgets have been written, with the exception of several which present exceptional difficulty and will need to be added into the system in the future. Pull-down

selection menus in particular were unable to be completed. Additionally, a few widgets, such as the file selector, look entirely different in HTML from their appearance in standard GTK+. HTML and CSS do not include methods to customize these objects. It may be possible to rewrite their functionality from scratch using a combination of HTML, CSS, and JavaScript, but that was considered beyond the scope of this project.

GTK+ defines many more event signals than JavaScript uses. The ones which have direct correlations were included, but certain others were left out. The "backspace" event was omitted because in a Web browser, this event is a key press event with a particular key being signals. The "delete-event" signal will never occur in a Web page, and was not implemented. Other signals falling into either of these categories were also omitted. A full list of the supported signals can be found in Appendix B.

4 Results

The project can tentatively be called a success. Whether a given tool will achieve widespread use can never be predicted ahead of time on the Internet. Linux, the operating system which powers a large portion of the sites available on the Web[9] was originally "just a hobby, won't be big and professional like gnu."[10] However, some projects expected to be huge successes quickly fade into obscurity. "... HyTime was not some rinky-dink academic research project; it was an ISO standard. It was approved for military use. It was Big Business. And you can read about it yourself... on this HTML page, in your web browser."[11] This project has driven the creation of a program which may be used in the writing of other Web pages, which would indicate that it has succeeded in helping to advance one possible solution to the problem of distributing software.

This project also showed success on a smaller scale. The process of writing user stories and use cases in order to plan out the program is rarely necessary during a single-programmer endeavor, but its use here allowed for greater practice with proper software engineering methods. The steps leading to the choices made for the program were highly informative, as is often the case with complicated computer science projects.

The increasing importance of HTML5, CSS3, and related technologies was demonstrated by this project. A number of features were only possible to implement due to the existence of the drafts of these standards. Although the standards are not yet supported by all Web browsers, they provide a consistent style for what will be used in the future, allowing Webshims and similar libraries to be written to enable backwards compatibility, secure in the knowledge that the libraries will slowly become less necessary. This project would not have been possible several years ago, or would have required major design decisions which could not be guaranteed to be used successfully.

4.1 Difficulties Encountered

The primary sources of difficulties in the completion of this project were personal causes, poor documentation, and differing display paradigms. The personal causes centered around eagerness to deliver more completed steps rather than receiving feedback on all of the intermediary sections. This is a pitfall for many engineers which can only be overcome with disciple and practice.

The documentation of the GtkBuilder schema proved to be a major time sink. The choices about what should be recorded in the file and what should be omitted and left as a default were often the precise opposite of HTML and CSS. For example, an HTML element has the minimum width possible by default, while a GTK+ widget has the maximum width possible unless otherwise specified. Additionally, GtkBuilder is not well-documented, so the many default options must be discovered by trial and error, enabling unusual settings in Glade and checking to see what differences appear in the .glade file.

Including webshims to allow them to be included by a Web developer with the minimum amount of difficulty proved to be a challenge due to the complicated nature of the library. There are enough internal references to other files that it was necessary to include an entire directory instead of a single file as hoped.

The final difficulty was caused by one of the project's main reasons to exist. GTK+ and HTML have fundamentally different perspectives on the manner in which layout and spacing are handled. HTML provides no method to line up arbitrary edges of elements, while GTK+ has many different alignment options. This is most strongly illustrated by GTK+'s GtkTable widget, which handles layout most similarly to an HTML <table>. Tables in HTML are avoided for use in layout, as they do not adapt well to variable-sized displays. However, certain features from HTML5 and CSS3, including the "display: box" property, were able to simulate much of what was needed.

It is possible that, when the program is published, work could be attempted with the developers of GtkBuilder to better document and implement any options which may have been missed. As HTML and CSS are further developed, they may begin to support more of the element spacing commonly used in desktop GUIs. Finally, lessons learned from this project will be certain to aid the developer with programs in the future.

5 Conclusion

`gladetohtml` is now a completed, usable program. There are more features which will be added, but the core functionality is present and usable. It will be posted to news sites, the Glade mailing list, and other locations on the Web in an attempt to spread knowledge and use of the tool. The source code and documentation will be available as open source software on Github.com, where any other developers who are interested will have the opportunity to add their own contributions.

There are still several major areas to expand on the functionality of `gladetohtml`. To start with, the widgets and signals which have not been implemented should be examined. Many present their own unique challenges, but could add greatly to the options a designer has available. Additionally, a GUI and Web interface were imagined in the original design. As explained above, the command-line program was written according to the standards suggested in *The Art of Unix Programming*, which will aid in writing additional interfaces. Finally, the program does not support any of the new features from the recently released Glade 3.10.0, which could be analyzed and included by any programmer who desires them.

One final potential area for further work is on the styles themselves. The file `gtk.css` included with the program attempts to cause HTML objects to appear as close as possible to GTK+ widgets viewed on Ubuntu 10.10 with the standard "Ambiance" theme in GNOME. However, the option to alter the path loaded by pages generated was included in the program. Other styles, corresponding to alternative GNOME themes or even entirely new ideas, could be written by a Web developer skilled in CSS.

The World Wide Web is now the home of many unique ideas. This project successfully created a program to aid in the realization of some potential ideas, leading to even more functions, services, and tools to be available over the Internet.

Appendix A

Usage Manual

This brief manual provides a walkthrough on creating a simple Web page, beginning with Glade and ending with the JavaScript which will need to be written. It also lists the features of Glade currently supported by `gladetohtml`.

1. Begin by running Glade. It can be executed from the command line with the command `glade`, or can be found in the program menu. In GNOME, it can be found under Applications ->Programming. If Glade is not installed, you can download it from `http://glade.gnome.org`, or find it in your operating system's software package manager, if it has one.

   You should see a window which looks something like this appear:

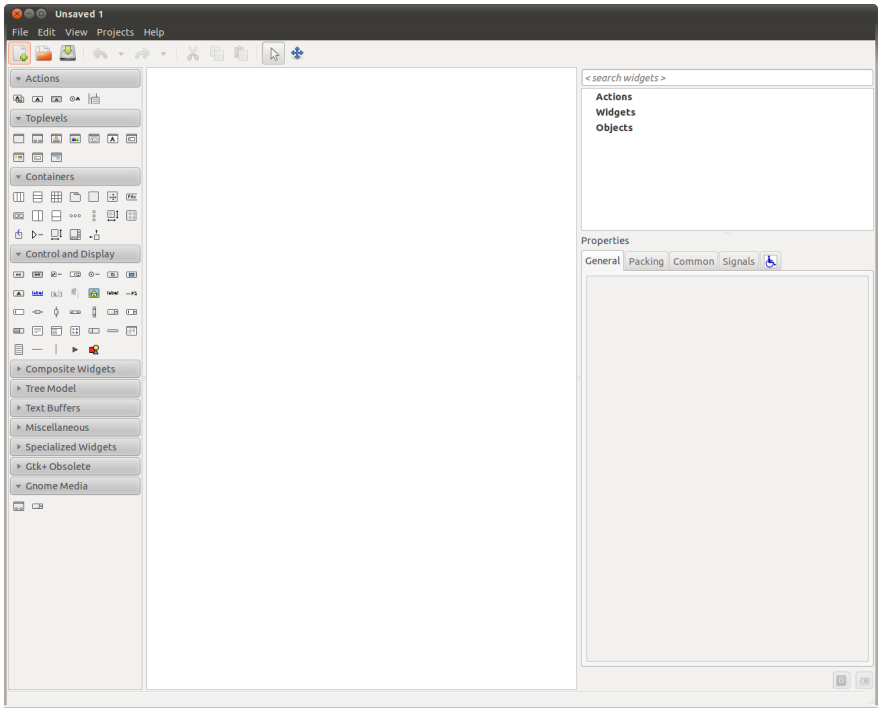

Figure 5.1: New Glade window.

2. Now, create the main window where all the elements will be found. Click on the first button, labeled “Window”, under the “Toplevels” list. A gray window with white lines inside, outlined in orange, should appear.

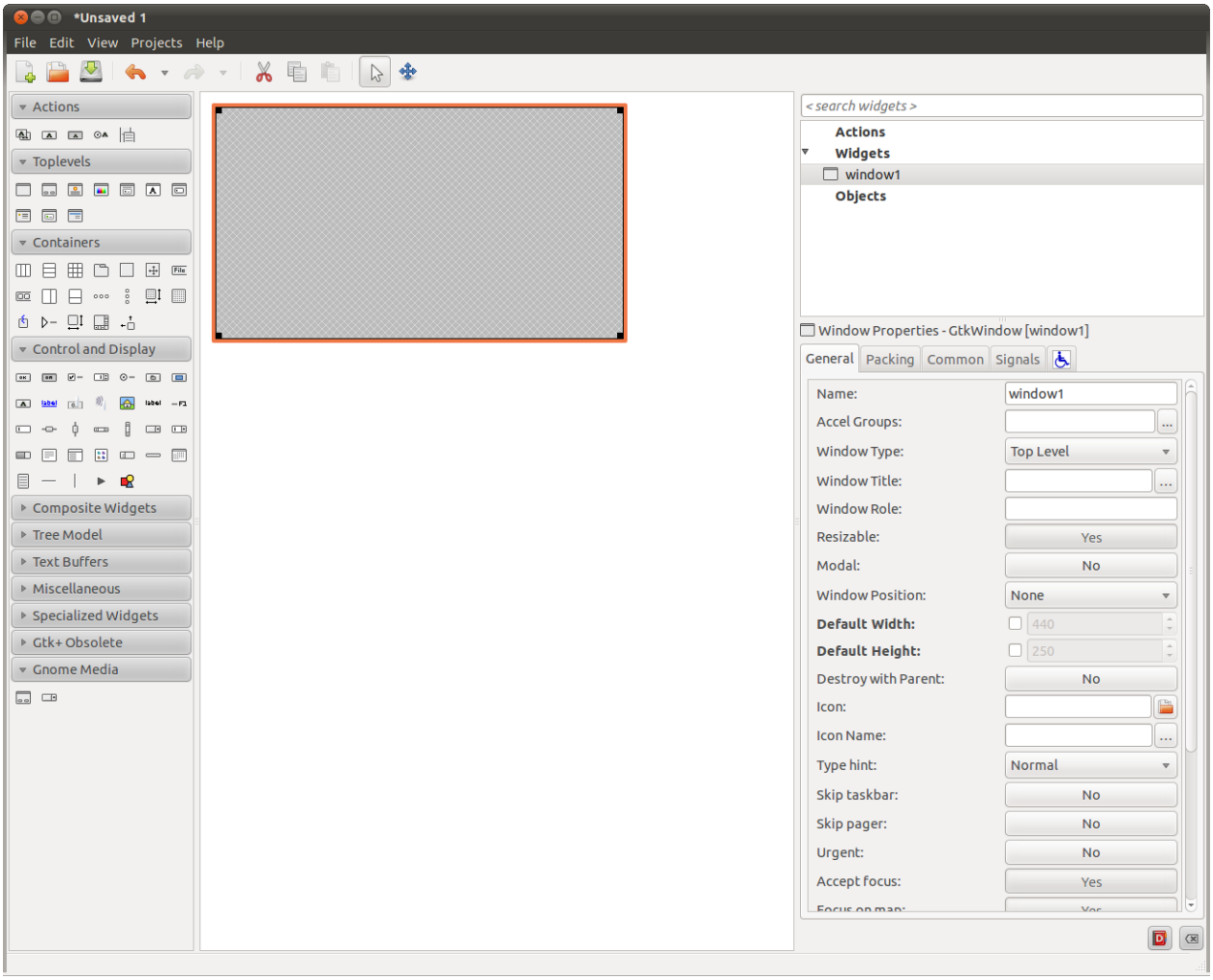

Figure 5.2: Main window placed.

3. To give the Web page a title, find the “Window Title:” property under the General tab of “Window Properties” on the right side of the Glade window. Click in the text entry and type “Name Submission Form.” Then, check the “Default Width” checkbox and set it to 440.

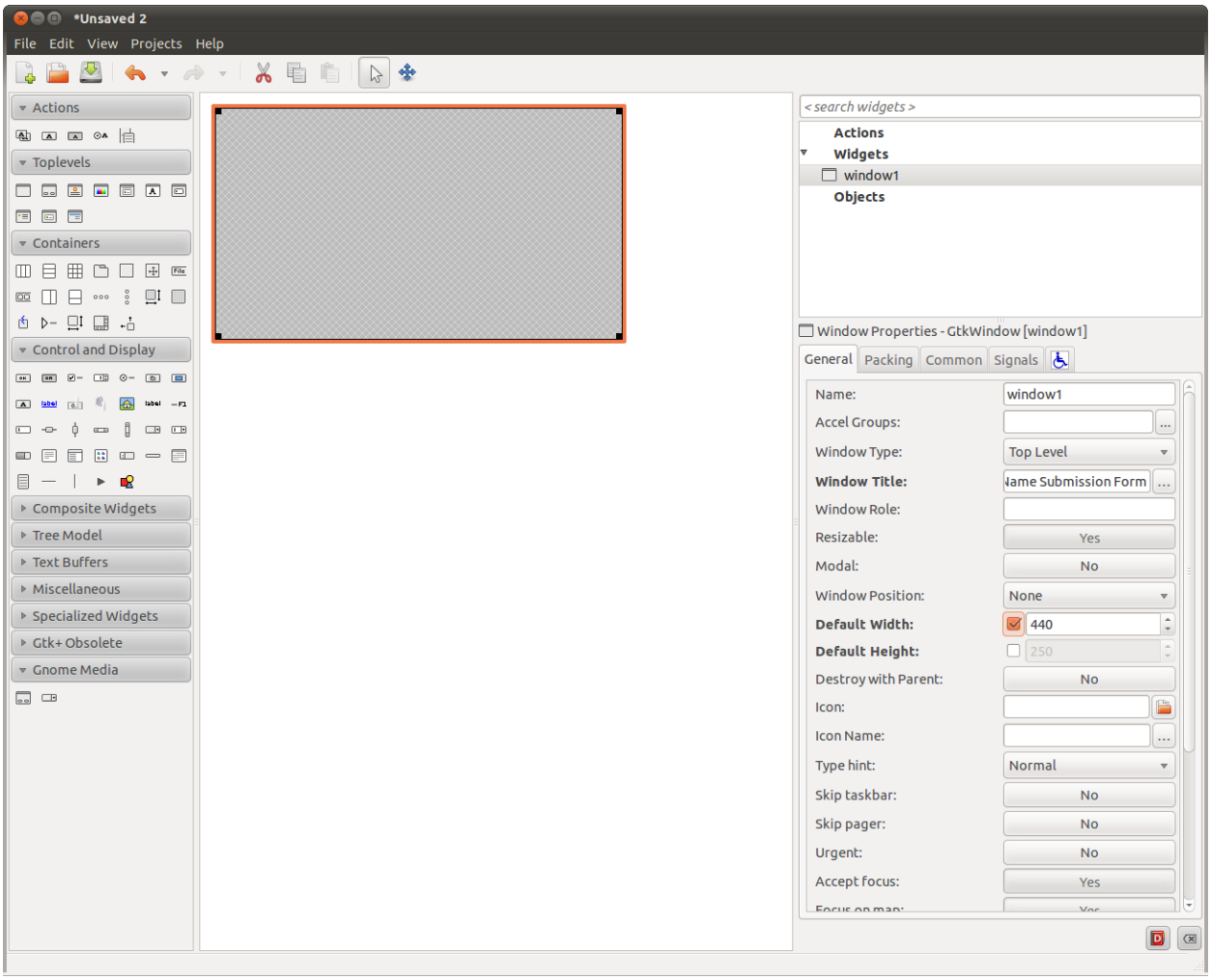

Figure 5.3: Window title set.

4. The main window can only contain one widget, so a “container” widget which can hold more should be placed. Click on the Vertical Box button, the second in the “Containers” list, and click in the gray main window. A popup will prompt you for the number of items you want. Set this to 2 and press “Ok.”

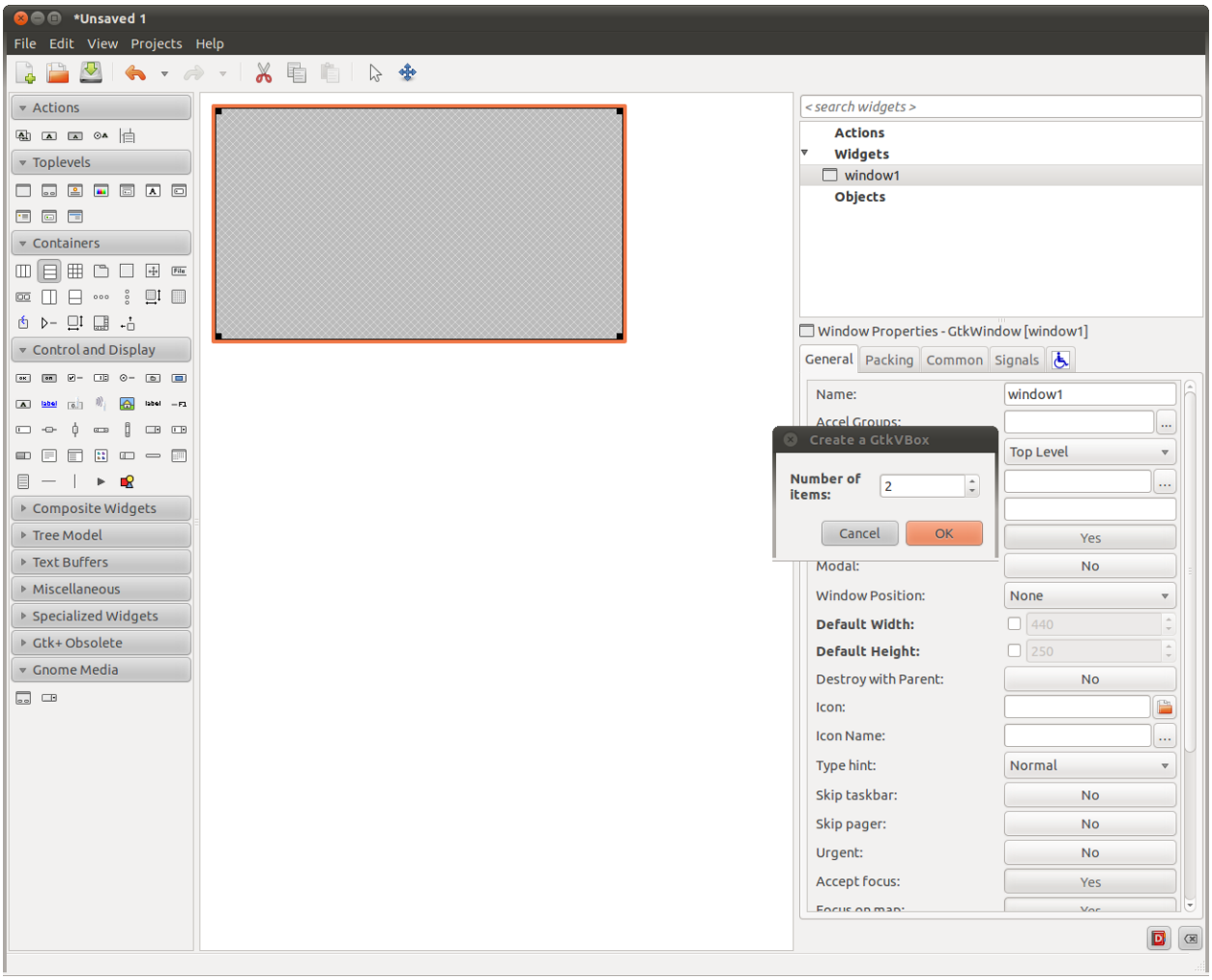

Figure 5.4: Creating Vertical Box.

5. A thin white line should now divide the main window area in half, creating two places to insert widgets. For this GUI, the top will be used for input, and the bottom for the "Submit" button.

   The input will require two widgets, an entry area and a label. Click on the Horizontal Box button, which is the first in the Containers list. Click in the area of the main window above the thin white line which now divides it. As before, a popup will appear. Set the number of items to 2 again and press "Ok." A new white line will divide the upper section of the main window.

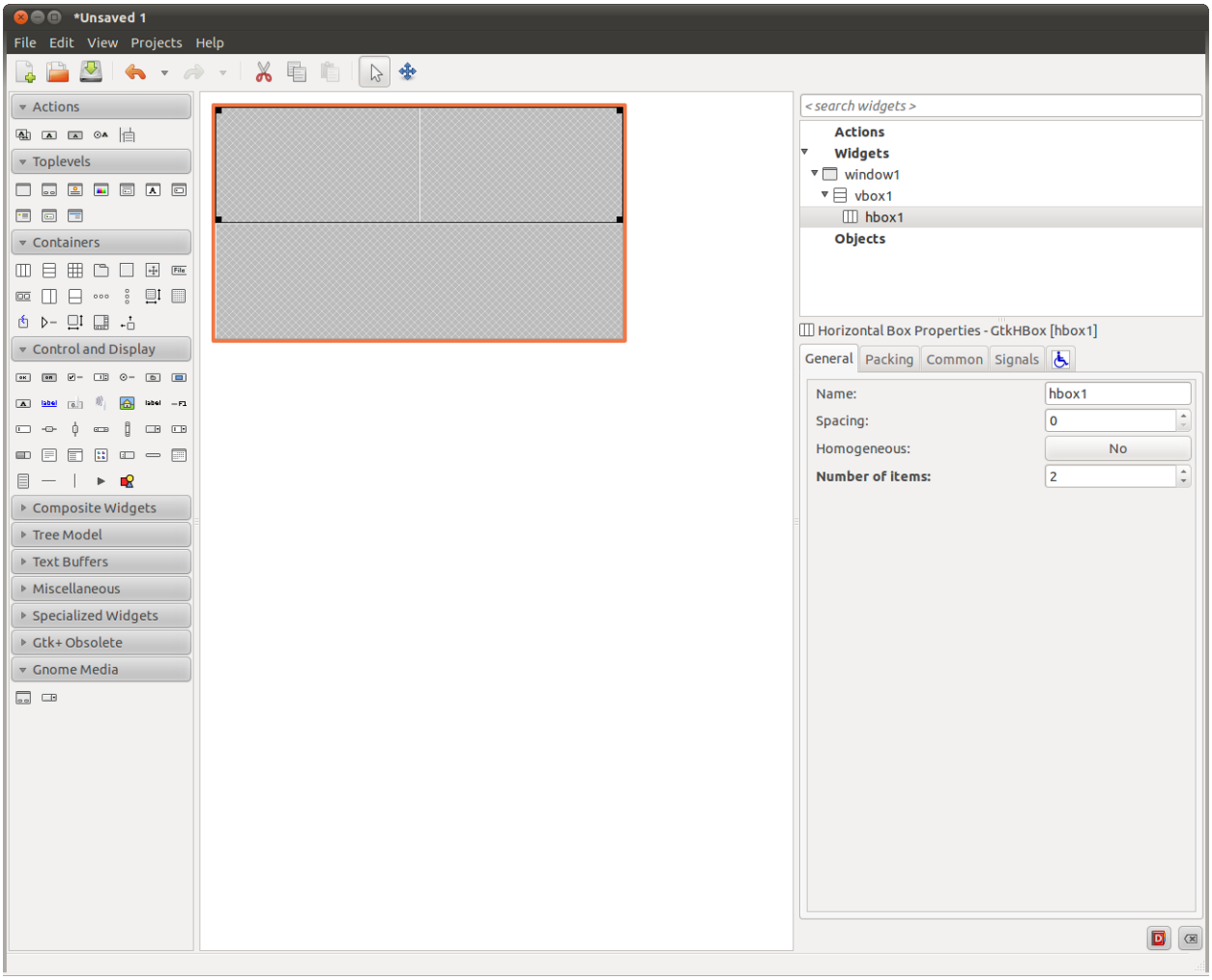

Figure 5.5: Creating Horizontal Box.

6. To create the label, find the “Label” button under “Control and Display.” It is the thirteenth widget, normally in the second row and the fifth column. Click on it and click in the upper-left section of the main window.

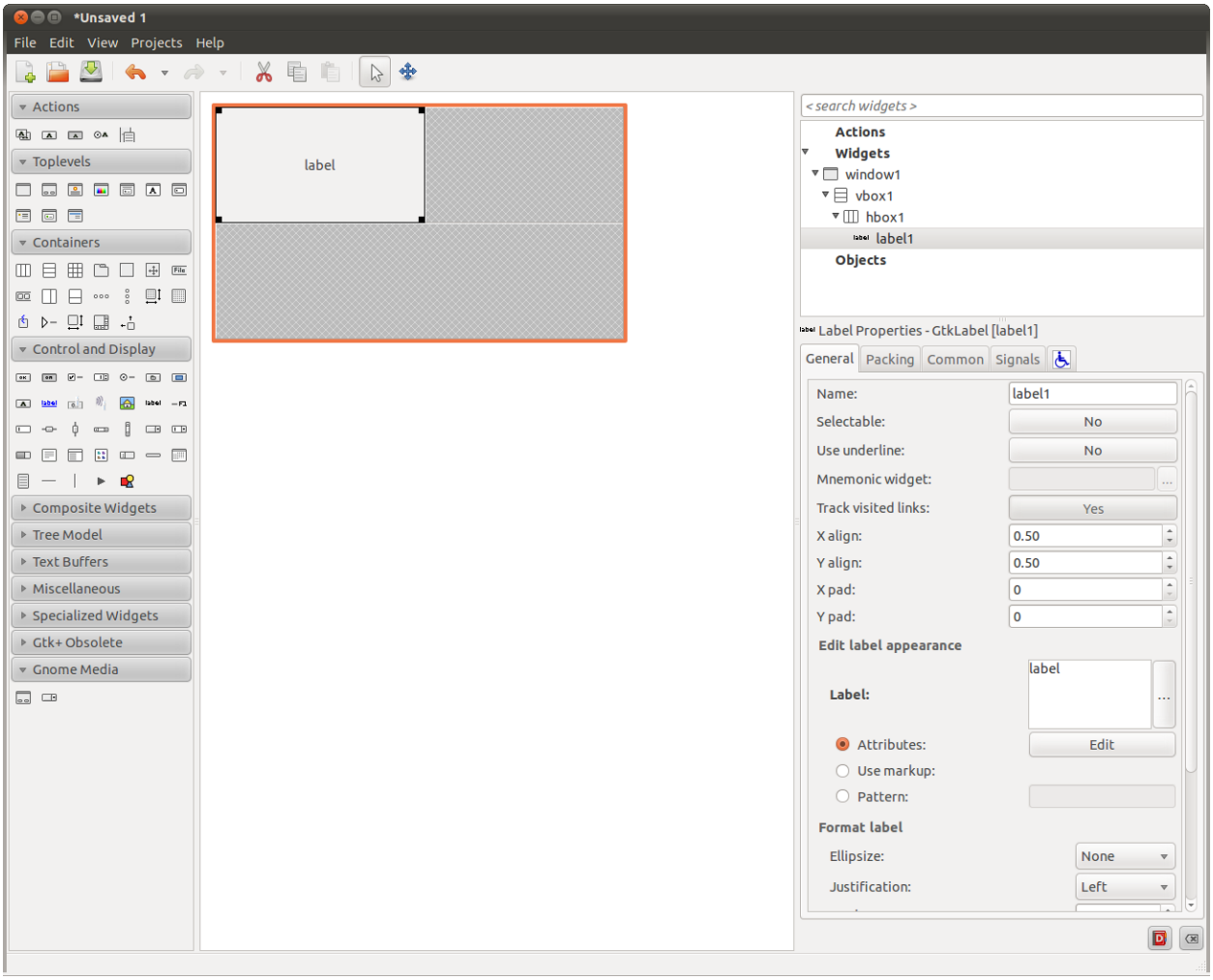

Figure 5.6: Adding Label.

7. In the “Label Properties” section on the right side of the Glade window, make sure the General tab is selected. Scroll down to “Edit Label appearance,” where the first section is “Label:” and change the text from “label” to “Name:” The label in the main window will change to reflect the new text.

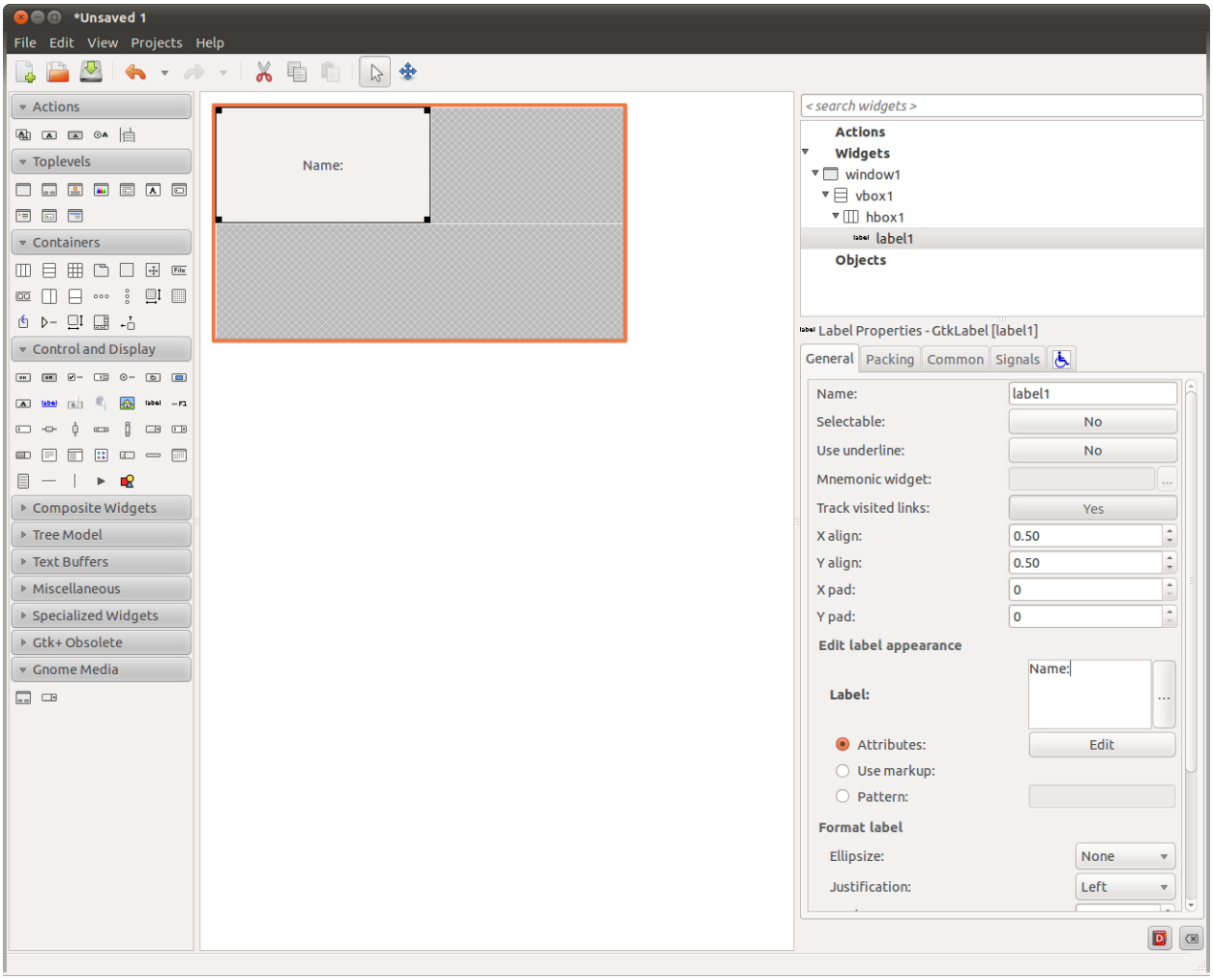

Figure 5.7: Setting Label text.

8. Next, add a “Text Entry” widget. It is in “Controls and Display,” usually the first element in the third row. Click to place it in the upper-right section of the main window.

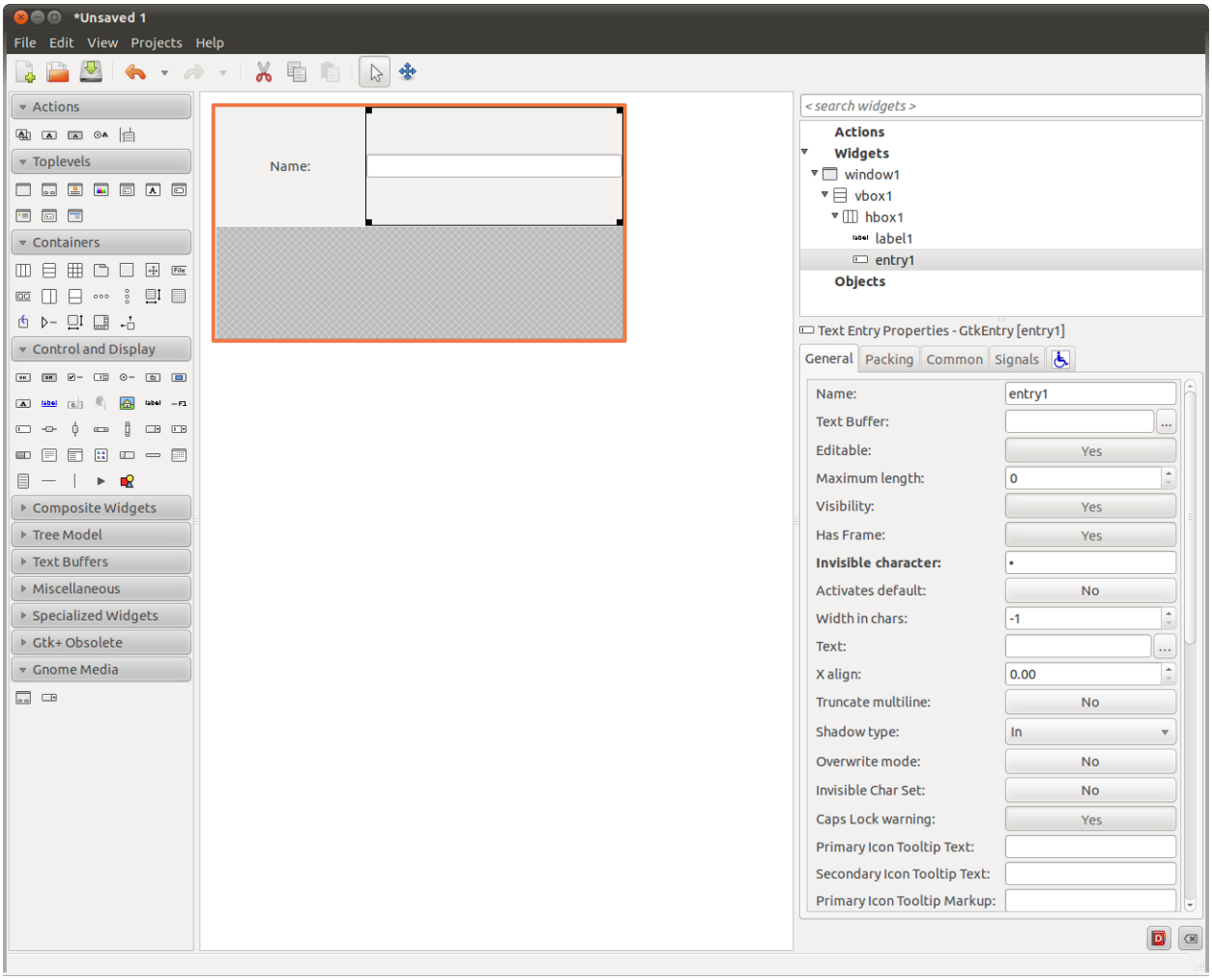

Figure 5.8: Adding Text Entry.

9. The Text Entry should still be selected. Click on it if it is not. Then, in the General tab of the Text Entry Properties area of the Glade window, find the “Name” label. Change this from “entry1” to “nameentered”. This will be used in the Web page later.

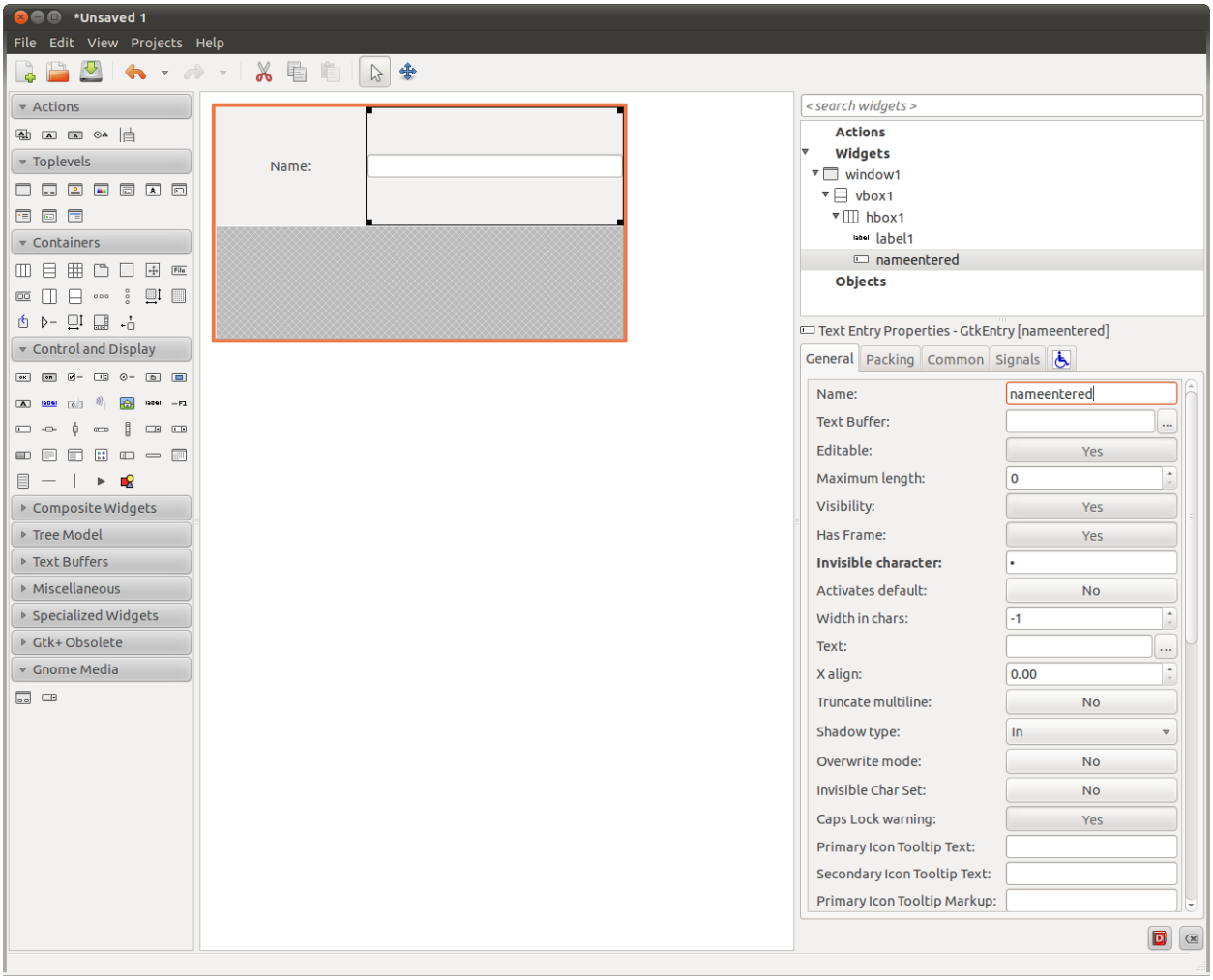

Figure 5.9: Setting Text Entry name.

10. For the final widget, choose a Button, the first widget in the Controls and Display list. Click in the bottom section of the main window to place it.

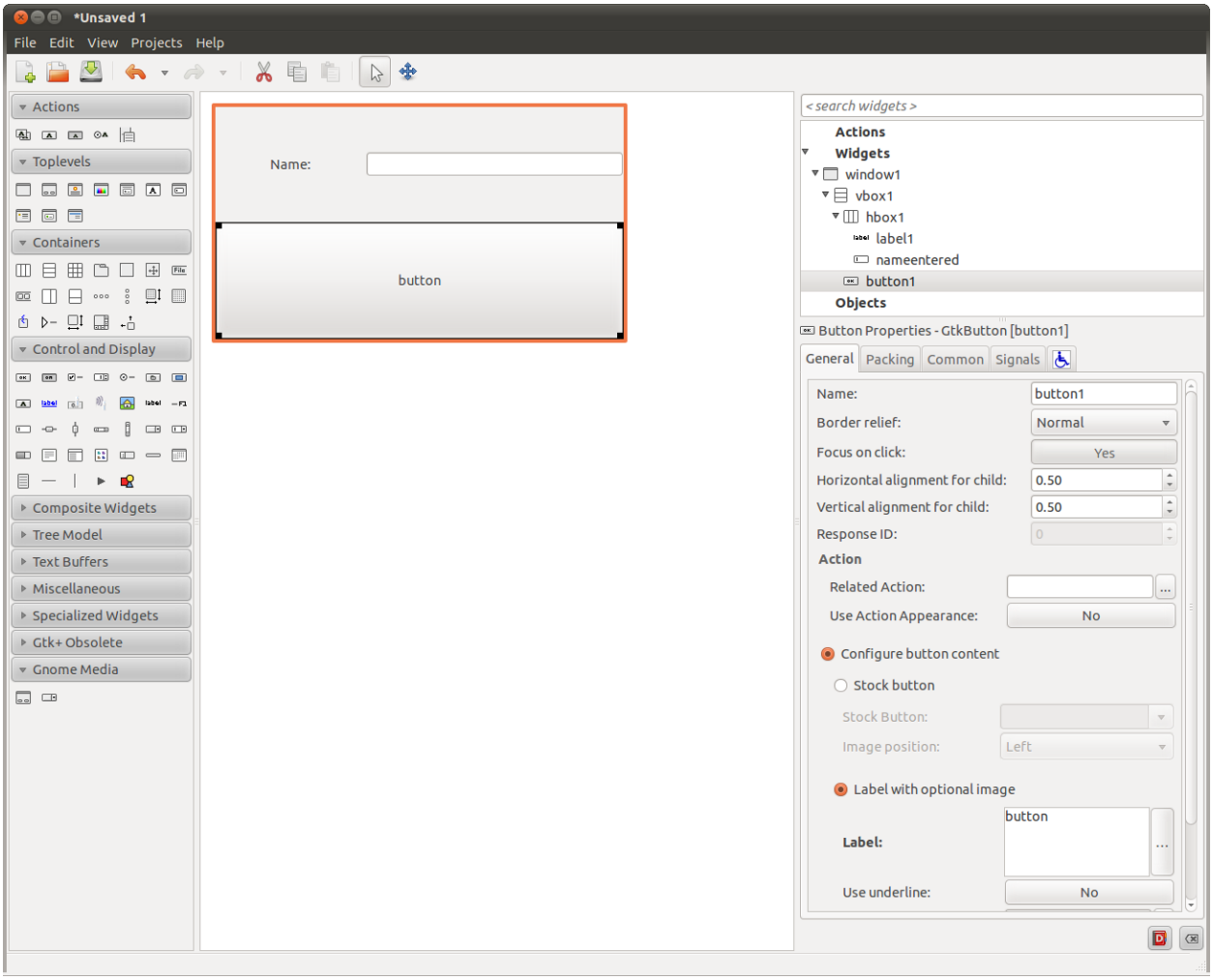

Figure 5.10: Adding Button.

11. In Button Properties, change the name to “submit” and the label to “Submit”, similar to how they were set for the Label and Text Entry.

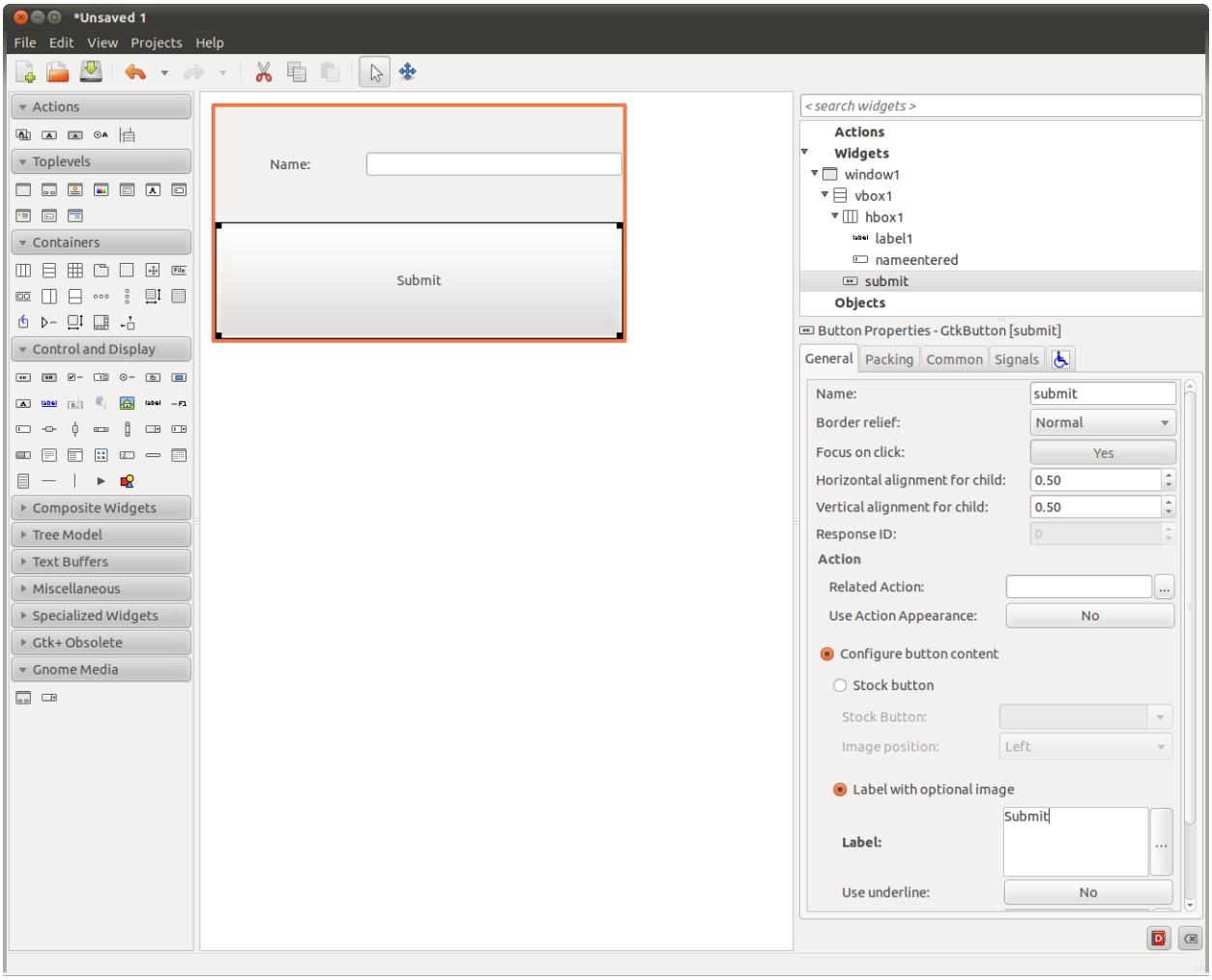

Figure 5.11: Setting Button properties.

12. Now, select the “Signals” tab under Button Properties. Find the “clicked” signal under “GtkButton”, and click where it says “<Type here>” under “Handler.” Type “submit_cb” and press Enter.

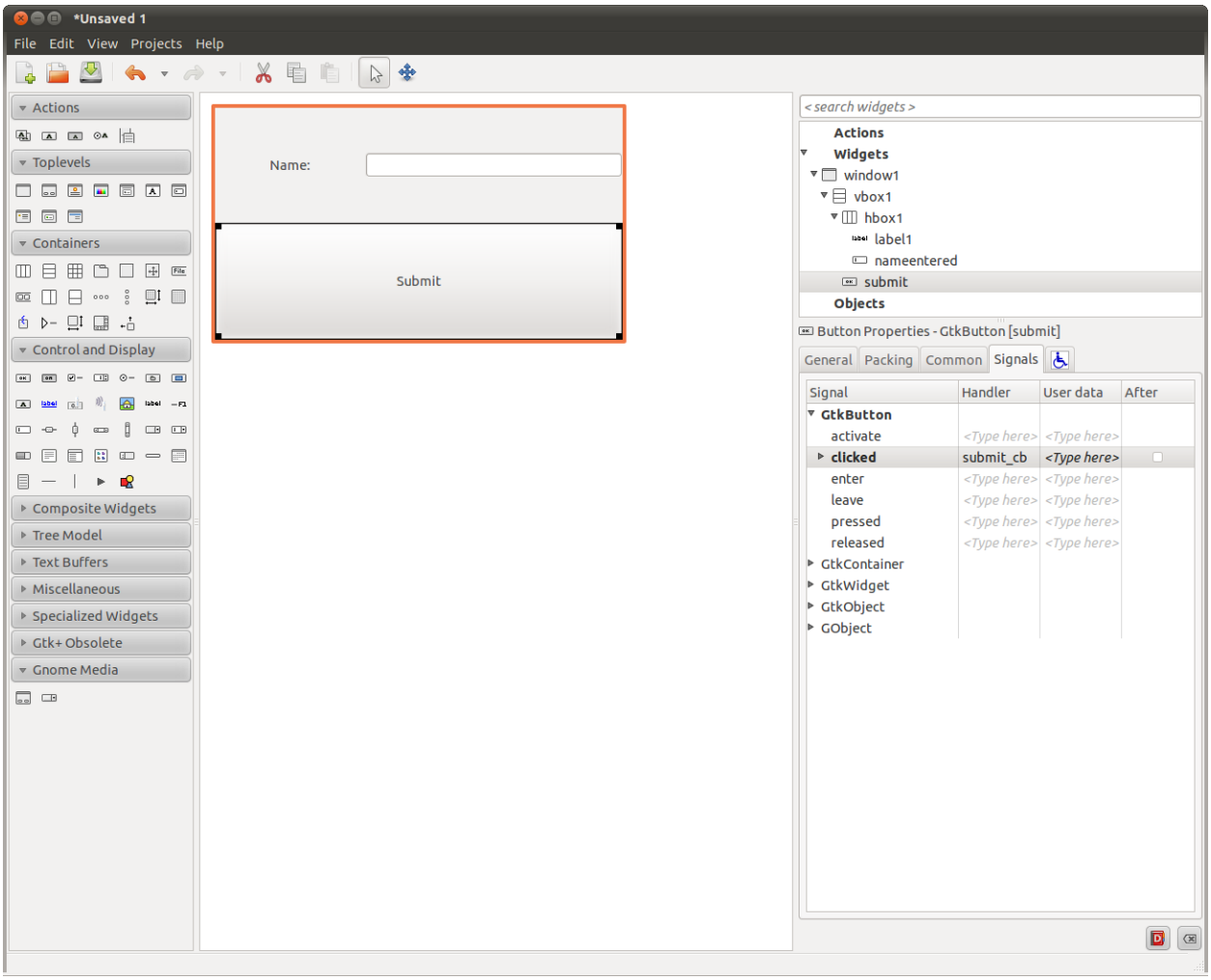

Figure 5.12: Setting “clicked” handler.

13. Save the GUI you have created by clicking the "Save" button or selecting the "Save" option under the "File" menu. Name it "example_gui.glade". You can now close Glade.

14. Now, on the command line, execute the program. If it is in the same directory as you are, the command is most likely:

```
$ gladetohtml example_gui.glade
```

    If the command is a success, the following table will be printed.

SIGNAL	OBJECT	ID	HANDLER
clicked	GtkButton	submit	submit_cb

    This indicates that one signal handler was defined, the "clicked" signal on the GtkButton object named "submit," and the function it will call is named "submit_cb."

15. Open the file "example_gui.js" in a text editor. The file should already exist, but be empty. Type this function:

```
function submit_cb () {
  alert (document.getElementById (``nameentered'').value);
}
```

16. Now, open a web browser and point it at the file in the directory you are currently in named "example_gui.html"

    You should see the GUI you designed, although it will look slightly different from how Glade displayed it. Glade handles certain aspects of widget dimensions differently from GTK+, and thus from the generated HTML.

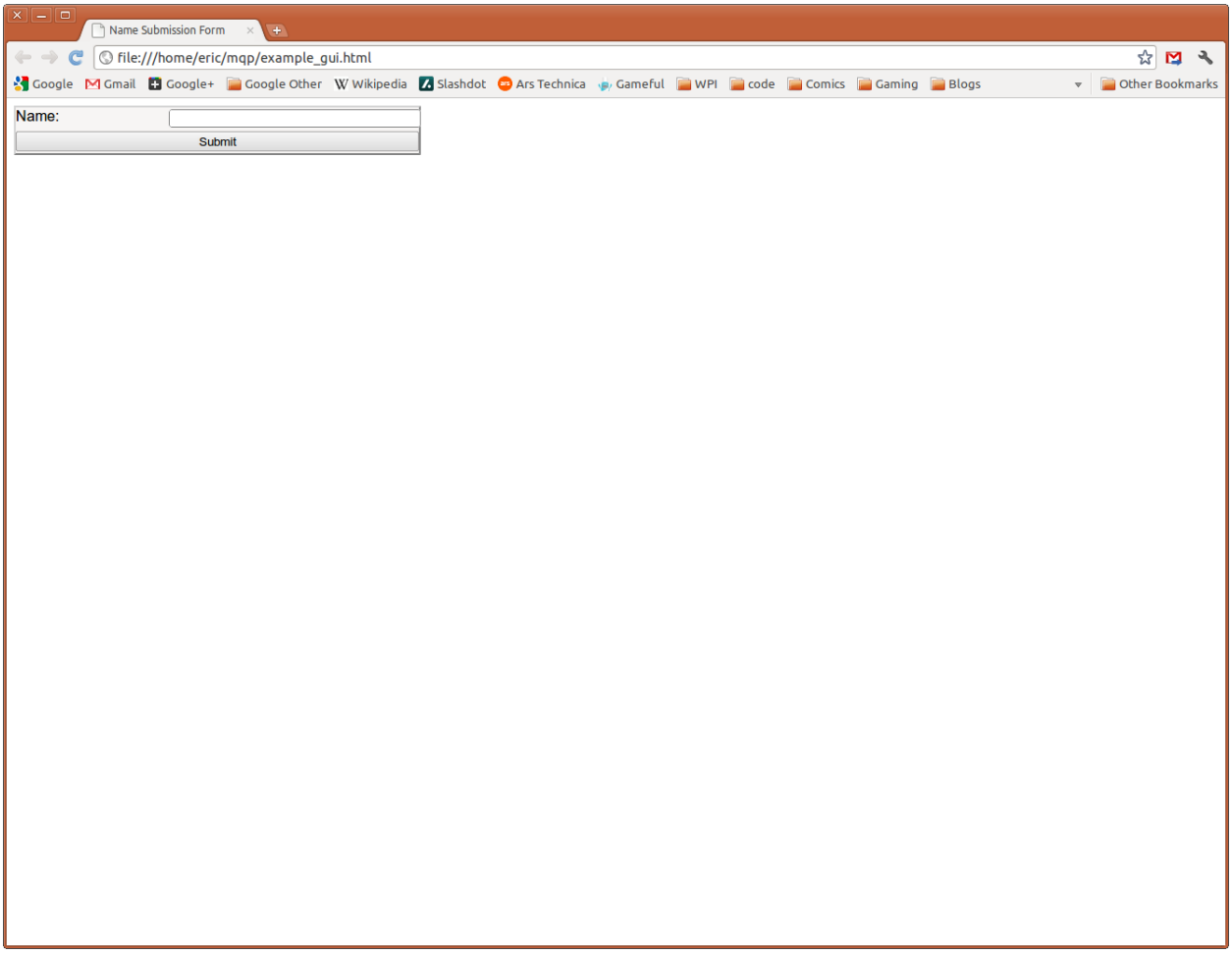

Figure 5.13: GUI viewed in Web browser.

17. The GUI is complete, and should be fully functional. If you type your name into the field and click "Submit", an alert box should pop up with the name you entered. You may now copy example_gui.html, example_gui.css, and example_gui.js to wherever you would like to make the GUI usable by others.

    The GUI produced when example_gui.glade is rendered in GTK+ is presented in figure 5.15 for comparison.

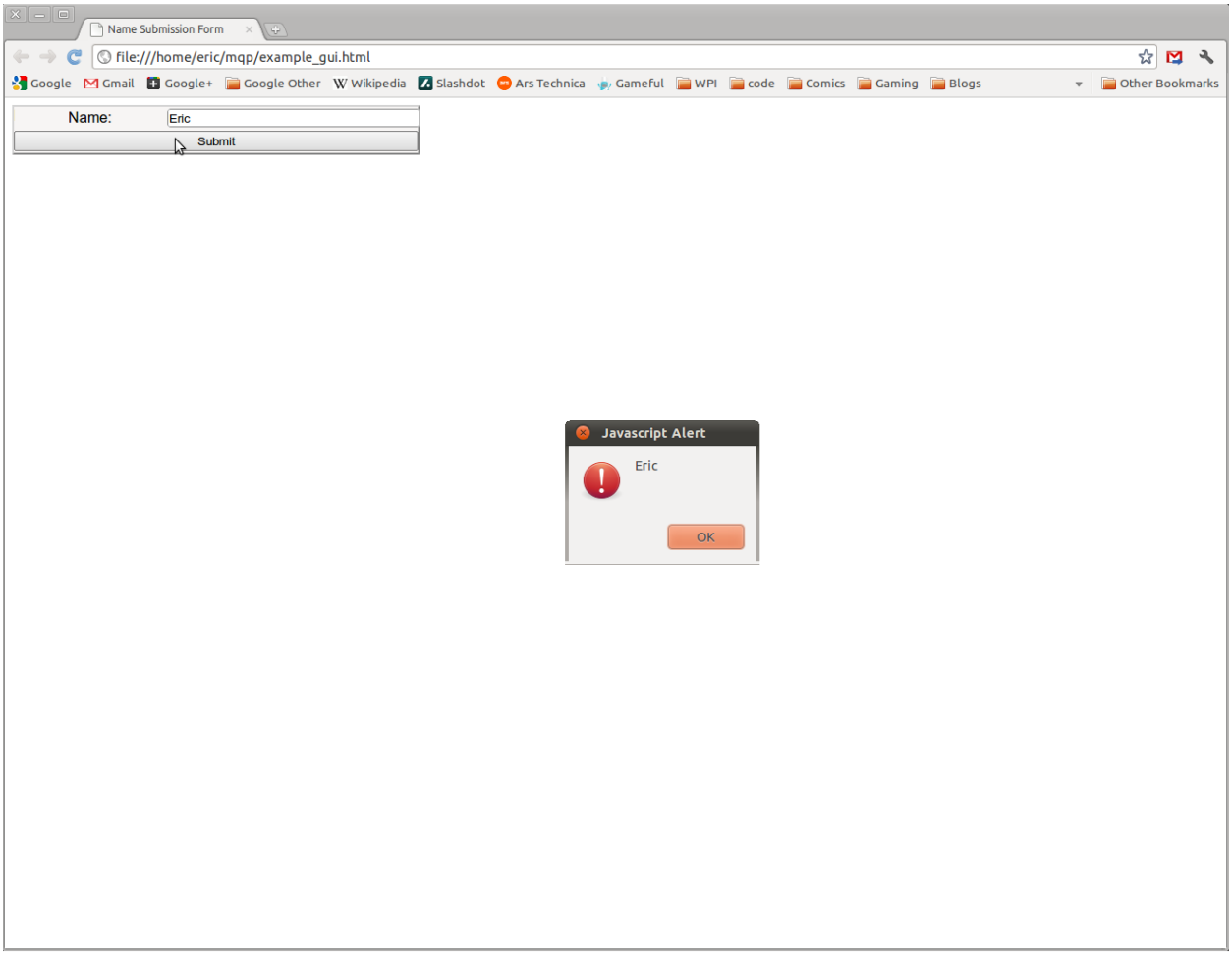

Figure 5.14: GUI in use.

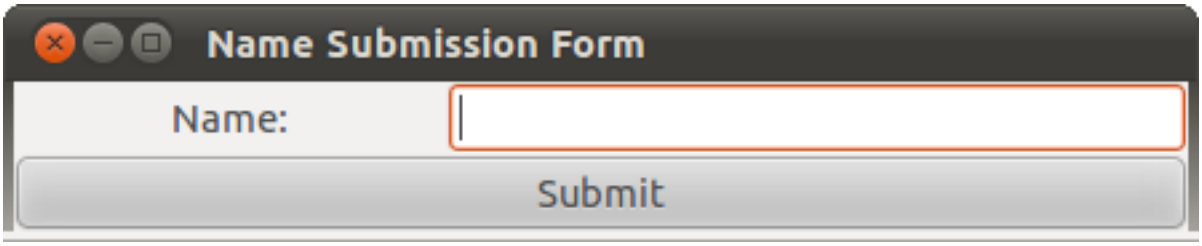

Figure 5.15: GUI rendered in GTK+.

Appendix B

Implemented Widgets, Properties, and Signals

The widgets currently supported by `gladetohtml` are:

- GtkWindow
- GtkVBox
- GtkHBox
- GtkVButtonBox
- GtkHButtonBox
- GtkLabel
- GtkButton
- GtkEntry
- GtkToggleButton
- GtkCheckButton
- GtkRadioButton
- GtkFileChooserButton
- GtkColorButton
- GtkLinkButton
- GtkImage
- GtkHScale
- GtkVScale

The properties currently supported by `gladetohtml` are:

- default_width
- width_request
- default_height
- height_request
- visible
- expand
- fill
- xpad
- ypad
- xalign (partial, is collapsed to left, center, or right)
- yalign (partial, is collapsed to top, middle, or bottom)
- single_line_mode
- has_frame

The signals currently supported by `gladetohtml` are:

- clicked
- enter
- leave
- pressed
- released
- button-press-event
- button-release-event

- focus
- focus-in-event
- focus-out-event
- key-press-event
- key-release-event
- insert-text
- delete-text
- changed

Appendix C

Program Files

This report should be distributed with copies of the three primary files necessary for the program to be used: `gladetohtml`, `gladetohtml.xsl`, and `gtk.css`. The most recent versions will be accessible from Github at the URL `https://github.com/epw/gladetohtml`. Instructions for how to download a copy of the files, and make changes, are on that page as well.

The source code is released under the GNU GPL v3.0

References

[1] The Glade Project, *Glade - A User Interface Designer*, http://glade.gnome.org, 2009.

[2] The GTK+ Team, *The GTK+ Project,* http://www.gtk.org, 2011.

[3] Aguilar, Rose, "Netscape Navigator 2.0 hits the streets," *CNET News*, 5 February 1996. Retrieved from http://news.cnet.com/Netscape-Navigator-2.0-hits-the-streets/2100-1023_3-203825.html.

[4] Cohn, Mike, *User stories applied: for agile software development,* Addison-Wesley Professional, 2004. Retrieved from http://books.google.com/books?id=DHZP_YL3FxYC.

[5] The Apache Software Foundation, *Welcome to the Apache Software Foundation!* http://www.apache.org/, 2011.

[6] W3Schools, "XSLT Browsers," *w3schools.com*, http://www.w3schools.com/xsl/xsl_browsers.asp, 2011.

[7] Android, "Issue 9312: Support XSLT in browser," *Android - An Open Handset Alliance Project.* Google, Inc. http://code.google.com/p/android/issues/detail?id=9312, 2011.

[8] The Apache Software Foundation, *The Apache Xalan Project*, http://xalan.apache.org, 2005

[9] Niccolai, James, "Ballmer Still Searching for an Answer to Google," *PCWorld Business Center*, 26 Sept 2008. Retrieved from http://www.pcworld.com/businesscenter/article/151568/ballmer_still_searching_for_an_answer_to_google.html.

[10] Torvalds, Linus B., "Message from discussion *What would you like to see most in minix?*" *comp.os.minix* 26 Aug 1991. Retrieved from http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b.

[11] Pilgrim, Mark. "No. 1. How Did We Get Here?" *Dive Into HTML5*, http://diveintohtml5.org/past.html, 2011.

Bibliography

Aguilar, Rose, "Netscape Navigator 2.0 hits the streets," *CNET News*, 5 February 1996. Retrieved from http://news.cnet.com/Netscape-Navigator-2.0-hits-the-streets/2100-1023_3-203825.html.

Android, "Issue 9312: Support XSLT in browser," *Android - An Open Handset Alliance Project.* Google, Inc. http://code.google.com/p/android/issues/detail?id=9312,2011.

The Apache Software Foundation, *The Apache Xalan Project*, http://xalan.apache.org, 2005

The Apache Software Foundation, *Welcome to the Apache Software Foundation!* http://www.apace.org/, 2011.

Cohn, Mike, *User stories applied: for agile software development*, Addison-Wesley Professional, 2004. Retrieved from http://books.google.com/books?id=DHZP_YL3FxYC.

GitHub, Inc., *github - Social Coding*, https://github.com, 2011.

The Glade Project, *Glade - A User Interface Designer*, http://glade.gnome.org, 2009.

Google, Inc., *Google Docs*. https://docs.google.com, 2011.

The GTK+ Team, *The GTK+ Project*, http://www.gtk.org, 2011.

Meyer, Eric, *More Eric Meyer on CSS*, New Riders Publishing, April 2004.

Niccolai, James, "Ballmer Still Searching for an Answer to Google," *PCWorld Business Center*, 26 Sept 2008. Retrieved from http://www.pcworld.com/businesscenter/article/151568/ballmer_still_searching_for_an_answer_to_google.html.

Pilgrim, Mark, *Dive into HTML5*, http://diveintohtml5.org, 2011.

Raymond, Eric S., *The Art of Unix Programming*, http://www.catb.org/~esr/writings/taoup/html/index.html, 2003.

Torvalds, Linus B., "Message from discussion *What would you like to see most in minix?*" *comp.os.minix* 26 Aug 1991. Retrieved from http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b.

W3Schools, "XSLT Tutorial," *w3schools.com*, `http://www.w3schools.com/xsl/default.asp`, 2011.

W3Schools, "XPath Tutorial," *w3schools.com*, `http://www.w3schools.com/xpath/default.asp`, 2011.