AN XML-BASED FORM CREATION SYSTEM

A graduate project submitted in partial fulfillment of the requirements
For the degree of Master of Science
in Computer Science

By

Justin E. Peckner

May 2013
The graduate project of Justin E. Peckner is approved:

_____________________________  _________________________
Richard Covington, Ph.D  Date

_____________________________  _________________________
John Noga, Ph.D  Date

_____________________________  _________________________
Jeff Wiegley, Ph.D, Chair  Date

California State University, Northridge
Dedication

To my family, for their unwavering love, support, and good humor.
# Table of Contents

Signature page ii

Dedication iii

List of Figures vi

Abstract viii

1 Introduction 1

2 Problem Description 3

3 Fundamental Technologies 5

4 Overview of Form Creation Process 9
   4.1 HtmlElementBase/CsunForm/FormElement . . . . . . . . . . . . . . . . . 10

5 XML to Java Compiler 13
   5.1 Lexical Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      5.1.1 Construct Tokens . . . . . . . . . . . . . . . . . . . . . . . . . 14
      5.1.2 Check Tokens . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
   5.2 Syntax Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      5.2.1 Check Root Tokens . . . . . . . . . . . . . . . . . . . . . . . . . 17
      5.2.2 Check Other Tokens . . . . . . . . . . . . . . . . . . . . . . . . 17
   5.3 Element Generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
      5.3.1 Attribute Verification . . . . . . . . . . . . . . . . . . . . . . . . 20
      5.3.2 Class XmlToJavaBase . . . . . . . . . . . . . . . . . . . . . . . 21
      5.3.3 Generation Steps . . . . . . . . . . . . . . . . . . . . . . . . . 22

6 Java to HTML Generator 26
   6.1 Data Finalizer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   6.2 HTML Generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
      6.2.1 CsunForm to HTML . . . . . . . . . . . . . . . . . . . . . . . . . 28
      6.2.2 FormElements to HTML . . . . . . . . . . . . . . . . . . . . . . 30

7 User Controls 32
   7.1 XML Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
   7.2 Webpage Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

8 Adding ElementTypes/Attributes 40
   8.1 HtmlElementType . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
   8.2 HtmlAttribute . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
   8.3 Generating Non-HTML Content . . . . . . . . . . . . . . . . . . . . . . 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>System Development Process</td>
<td>44</td>
</tr>
<tr>
<td>9.1</td>
<td>Programmatic Form Creation</td>
<td>44</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Defining the Fundamentals</td>
<td>45</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Combining FormElements into CsunForms</td>
<td>46</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Form Generation</td>
<td>47</td>
</tr>
<tr>
<td>9.2</td>
<td>Developing an XML Compiler</td>
<td>48</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Lexical and Syntax Analysis</td>
<td>48</td>
</tr>
<tr>
<td>9.2.2</td>
<td>XML Tokens to Java Objects</td>
<td>49</td>
</tr>
<tr>
<td>9.2.3</td>
<td>Separating Compiler from Forms</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>10.1</td>
<td>Future Expansion</td>
<td>51</td>
</tr>
<tr>
<td>10.2</td>
<td>Summary</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>54</td>
</tr>
<tr>
<td>A</td>
<td>Sample XML Markup and Resulting Forms</td>
<td>55</td>
</tr>
<tr>
<td>A.1</td>
<td>Form One: Bacon Ipsum</td>
<td>55</td>
</tr>
<tr>
<td>A.2</td>
<td>Form Two: Human Resources Sign-In form</td>
<td>58</td>
</tr>
<tr>
<td>B</td>
<td>XML Schema</td>
<td>65</td>
</tr>
<tr>
<td>B.1</td>
<td>Formal Schema</td>
<td>65</td>
</tr>
<tr>
<td>B.2</td>
<td>Informal XML Structure</td>
<td>73</td>
</tr>
<tr>
<td>C</td>
<td>Class UML Diagrams</td>
<td>77</td>
</tr>
<tr>
<td>C.1</td>
<td>Package edu.csun.forms_site.compiler</td>
<td>77</td>
</tr>
<tr>
<td>C.2</td>
<td>Package edu.csun.forms_site.generator</td>
<td>83</td>
</tr>
<tr>
<td>D</td>
<td>Source Code / Unit Tests / Javadoc / UML</td>
<td>88</td>
</tr>
</tbody>
</table>
## List of Figures

3.1 A sample XML file. ............................................. 5
3.2 A sample HTML file. ........................................... 6
3.3 Resulting page of the HTML markup in Figure 3.2. ............... 6

4.1 Overview of the XML to HTML process. ........................ 9
4.2 XML markup of a simple form. ................................. 10
4.3 Hierarchy of markup in Figure 4.2. ............................. 10
4.4 HtmlElementTypeFactory.HtmlElementType enum members. .... 11

5.1 Table of XML Predefined Entities. ............................... 13
5.2 Phases of lexical analysis. ...................................... 14
5.3 DFA used in lexical analysis. .................................... 15
5.4 Table of assumed TokenType. ................................... 15
5.5 Lexical errors are caught and thrown back to the user. ......... 16
5.6 Phases of syntax analysis. ....................................... 17
5.7 PDA used in syntax analysis. .................................... 18
5.8 Syntax errors are caught and thrown back to the user. ......... 20
5.9 UML representation of class XmlToJavaBase. .................... 21
5.10 Interface XmlToJavaInstantiator. ............................... 22
5.11 XmlToJavaInstantiator FORM_ELEMENT_INSTANTIATOR. ... 22
5.12 Phases of element generation. ................................... 22
5.13 FormElement generated from the button XML markup. ........ 23
5.14 FormElement generated from the label XML markup. .......... 23
5.15 Pseudocode for function tokensToElements(). ................ 25

6.1 Interface ValueGenerator. ....................................... 26
6.2 Interface WidthCalculator. ..................................... 27
6.3 Pseudocode for estimating text height. ........................ 28
6.4 Resulting page of an 8.5 x 11” cForm. ......................... 29
6.5 HTML div structure for a page in generated by the system. ... 29
6.6 Elements are laid-out left to right; new row is started when previous is full. 30
6.7 Space is automatically inserted if the user does not specify it. 31

7.1 Form compilation page on load. ................................ 38
7.2 Compilation errors are detailed for user review. ................ 39

8.1 Interface ElementType. .......................................... 42

9.1 XML markup for a combo box. ................................... 46
9.2 UML representation of class AttributeContent. ................. 46

A.1 XML markup for Bacon Ipsum text form (part 1/2). ............ 55
A.2 XML markup for Bacon Ipsum text form (part 2/2). ............ 56
A.3 Resulting Bacon Ipsum form (same as Figure 6.4). ............ 57
ABSTRACT

AN XML-BASED FORM CREATION SYSTEM

By

Justin E. Peckner

Master of Science in Computer Science

While web-based forms have become essential for any organization collecting and processing large amounts of data, CSUN currently has no central electronic form management system. This project is intended as the foundation of a future comprehensive electronic form system for CSUN. By inputting a block of XML markup consisting of easy-to-learn elements and attributes, users can instantly generate a usable HTML form containing labels, pictures, text boxes, buttons, and more. If the XML is malformed in one or more places, the system lists a specific location and message for each error.

To parse the XML, a customized XML compiler was created. The compiler first uses a deterministic finite automaton (DFA) to perform lexical analysis, character-by-character. This process breaks the original markup into separate tokens such as <label>, Hello World!, and </label>. Next, a pushdown automaton (PDA) performs syntax analysis to determine if the sequence of tokens entered is valid. If so, the sequence is converted to Java object instances, and any attribute values not entered by the user are intelligently guessed by the system. Finally, the instances are converted to HTML, which is displayed in the user’s browser.

Given that this is the first of several components comprising the future form management system, care has been taken to make this project’s code as open-ended and straightforward as possible. Additionally, while the XML compiler is used in this project to generate forms, it is completely independent of the form creation code. As such, programmers could easily interface with it for other XML-based purposes in the future.
Chapter 1
Introduction

With the advent of computers, databases, and the ability to link them together via networking, handling paperwork electronically has become standard practice for many organizations. Managing forms electronically saves in time and cost not only at the time the form is submitted, but later when the form’s data must be referenced as well. In medicine, where electronic records have recently come to the forefront, they were even found to improve care and significantly reduce accidental deaths [1].

Unfortunately, many departments at CSUN are still relying on paper as the medium for most of their forms. This is understandable, given that they do not have an easy-to-use system for creating and managing electronic forms. Such a system, at minimum, should include the following features:

- The ability to easily generate customized forms;
- The ability to have appropriate parties sign forms electronically, securely, and, if multiple signatures are needed, a system for relaying the form between the parties in the correct order;
- The ability to track the status of a form in real time;
- The ability to easily view and download a form’s data.

Software for creating and managing online forms has existed for many years. Adobe’s ubiquitous Acrobat software allows for the creation of professional-looking PDF forms, with input fields such as text, checkboxes, and others. Once a user has submitted a form, its data can be populated into a spreadsheet or database [2]. Other solutions, such as wufoo.com and perfectforms.com, offer the ability to create webpage-based forms (as opposed to PDFs), at various levels of pricing. For a truly customized solution tailored to fit the exact needs of a given business, large companies can turn to comprehensive (and correspondingly costly) solutions from companies such as IBM and SAP [3, 4].

However, given that commercial solutions can be quite expensive, free solutions also merit a look. A Google search for “create online forms” reveals a plethora of options. At a basic level, AppNitro Software’s phpform.org is a free service where users can select one of two dozen looks-and-feels; define the various input field types (text input, radio button, checkbox, etc.); and download the HTML and auxiliary files for their newly defined form [5]. Such a form, however, is useless without a back-end database and server to drive its usage.

Google’s own service, Google Drive, offers a “Create Form” capability. Users can define their own forms through a guided process. The interface is not quite as intuitive
as phpform.org’s, and does not offer as many input options (such as file upload or a date picker). However, the service does provide a means of data collection. Whenever a person inputs data to a form created by the service, the data is placed in a spreadsheet on the form owner’s Google Drive [6].

With so many options already in place for form creation, would it be possible to adapt one or some of them to CSUN’s needs? Unfortunately, the answer is no. Purchasing a customized solution from a commercial vendor is not realistic, given the California State University system’s budgetary situation. (A paid solution is also not necessarily a good one.) As for freely available tools, such as AppNitro’s and Google’s, these only provide front-end use features, without any kind of back-end data processing system. It would also not make sense to rely on a tool which could disappear in the future, should its parent company choose to no longer provide it.

It soon becomes apparent that a customized solution, built on top of free, open-source software platforms, is the best option for building a comprehensive form system for CSUN. In addition to costing virtually nothing for software tools, and a minimal amount for on-going maintenance, such a solution offers maximum customization and flexibility for any changes needed in the future. This project accomplishes the first of several steps required to make such a system a reality.
Chapter 2

Problem Description

In this section, the exact problem and solution will be explained in greater detail. As stated above, the general goal of the system is to create a tool whereby a CSUN staff member with average office computer experience, and little or no programming experience, could create a digital form for people to access via the Internet. The entire process of form creation and usage will be as follows:

• A CSUN department determines that it has a need for a particular form to be hosted online. This could be an existing paper form that needs to be ported to digital format; an existing digital form in need of a better management system; or a form that does not yet exist in any medium. In any of these cases, the department will determine the exact data that needs to be collected by the form, and appoints a liaison(s) to be in charge of the form creation process.

• Liaisons will register their intent to create a form with the Pioneering Technology Group (PTG), who will verify that the liaison is authorized to represent their department. Additionally, liaisons who are new to the system will receive training via a PTG webpage. Here they will learn about the XML language that the system uses to create forms, primarily through working XML examples, and pictures of the subsequent form outputs. These examples will range from simple to complex, and cover every type of input element. The training page will also include a list of frequently asked questions (FAQs), and contact information in case the user has questions that are not addressed.

• Once a user is comfortable with the XML language of the system, he will input the XML for their new form into the Form Creation page included in this project. Using the preview feature of the page, the user can view and print the form at any point during the XML input process. This page could be publicly available for anyone to create a form, even if he does not have system credentials.

• The system will automatically assume that each webpage element containing user input is to be stored as a unique field in the database. These element types include, but are not limited to, text boxes, date inputs, checkboxes, select boxes, and radio button groups.

• The liaison will then formally submit the form to the PTG for hosting. Upon doing so, the system will create a new table in the database exclusively for the form, with each field discovered in the above step being given its own column in the table. Depending on the level of customization requested by the liaison, a PTG programmer may also need to adjust the page’s HTML and/or JavaScript. The finalized webpage will be served from the PTG’s web server, unless the PTG and/or liaison determine that it will be served from elsewhere.
• Once the form is served to the campus at-large, its intended users will fill out and submit the form online instead of on paper. If needed, users will be able to save a draft of an in-progress form in the system. Once the form is submitted, the user can afterwards view the form’s live status: approved, rejected (and for what reasons), or still being processed. Both retrieving draft forms and viewing status will be done through a familiar central portal, preferably the myNorthridge Portal Login already hosted at www.csun.edu.

• Meanwhile, for forms requiring the approval of a person outside the department that created the form (for example, a professor who needs to sign off on a form that originated in Undergraduate Degree Services), an email will automatically be sent to that person. He will then digitally sign the form via a secure web portal. Once this is done, if other outside signatures are required, this step will be repeated until all have been obtained.

• Once all external approvals are received for a given form, the department which created the form will receive a notification that the form is ready for processing. Form data could be obtained by the department through a variety of means:
  ◦ A spreadsheet/CSV file
  ◦ Email
  ◦ A webpage table
  ◦ A direct link to the central form database (this should only be done if the system is very carefully arranged to prevent security breaches)

• The department then uses the data as it sees fit.

This submitted thesis project does not accomplish all of these steps, nor was it intended to. Rather, it implements the third bullet point, a web page that allows liaisons to generate HTML forms from an inputted block of XML markup. This satisfies the goals laid out and approved in the project proposal.
Chapter 3
Fundamental Technologies

This section details many of the technologies and standards used in this project.

- **XML**: Extensible Markup Language (XML) is one of the most widely used markup languages today. A markup language’s purpose is to describe things [7]. This differs from programming languages, whose job it is to instruct a computer what actions to perform, and in what sequence.

  A sample XML markup is as follows. Notice that the markup’s content is easily readable by both humans and computers:

  ```xml
  <movie>
  <title>Amadeus</title>
  <director>Milos Forman</director>
  <yearReleased>1984</yearReleased>
  </movie>
  
  Figure 3.1: A sample XML file.
  
  Certain parts of the markup, such as the word *Amadeus*, constitute “data”; that is, the actual information that the markup is intended to convey. Other items, such as `<title>` and `</title>`, are not part of that information themselves, but rather are metadata that tell us more about the information being conveyed. These are known as XML tags, or more often simply referred to as tags [8]. Tags are further discussed in Section 5.1.

  In XML, tags can also contain attributes, which provide additional data about the object being described [8]. For example, `<movie region="US">` contains an attribute listing the region in which the movie is being released. In this instance, an XML parser could require that movies released in the U.S. have an additional `<mpaaRating>` tag as a child of the `<movie>` tag, whereas a movie with `region="China"` might not be subject to that requirement.

  An XML document must have a single root element; that is, a pair of starting and ending tags which surround the rest of the document, and are not themselves surrounded by any other tags [9]. In Figure 3.1, `movie` is the root of the document. This ensures that the overall element being described by the XML is clearly distinguished.

- **HTML**: HyperText Markup Language (HTML) is the standard language used to create web pages. Like XML, HTML is a markup language [10]. Creating a simple text-only web page with HTML requires relatively little effort, while creating more complex pages, including forms for submitting data, can take significant amounts of
time and effort (a problem that this system was created to address). A sample HTML file is as follows:

```
<html>
<head>
  <title>My Web Page</title>
</head>
<body>
  <h1>This is a large header.</h1>
  <h5>This is a small header.</h5>
  <p>This is a paragraph of data.</p>
</body>
</html>
```

Figure 3.2: A sample HTML file.

When saved in a file and displayed in a web browser, the above HTML markup results in the following web page:

![Figure 3.3: Resulting page of the HTML markup in Figure 3.2.](image)

- **Java**: Java is an object-oriented programming language, first released in 1996 by Sun Microsystems (now a part of Oracle Corporation). A distinguishing characteristic of Java is its “write once, run anywhere” ability; once code for a Java program has been successfully written and compiled, it can be run on any platform that supports Java [11]. This list includes Windows, Mac OS X, Linux, Solaris, and others, together encompassing virtually every PC in use today.
• **Client**: a person/web browser requesting a web page.

• **Server**: a computer/computer application that returns a web page upon request by a client.

• **Dynamic Web Page**: HTML, on its own, only allows for static web pages. That is, a person writes the markup for an HTML page at a given point in time, and all future retrievals of the page return exactly the same content. However, people using today's Internet expect web pages to be highly customized to their own needs—everything from viewing grades, to trip itineraries, to income tax refund status. None of this would be possible if servers could only return pages taken from the same “frozen” set.

Instead, standards and programming languages have evolved which allow for servers to generate HTML pages on-the-fly, customized to the page requester’s exact needs. For example, upon receiving a request to view a particular student’s Spring 2013 grades, the server would look up the student’s grades in a database; write these grades into an HTML web page; and return the page to the user’s web browser. The first such standard to gain widespread use was the Common Gateway Interface (CGI) [12]. While it is still used in some quarters, CGI has been mostly replaced by standards such as PHP, JSP, .NET, Ruby, and others. This project uses JSP, as detailed below.

• **CSS**: Cascading Style Sheets (CSS) is a language for describing the on-screen look of markup language elements, especially those found in HTML web pages [13]. Developed by the World Wide Web Consortium (W3C), CSS is supported by most browsers. Using CSS, page designers can specify the width, height, color, positioning, and many other element layout factors.

• **JavaScript**: JavaScript is a scripting language used in web browsers [14]. Using JavaScript, page designers can write code in C/Java-like syntax to control virtually any behavior in a web page, from changing text to modifying CSS to communicating with the server, and more. Note that despite the name, JavaScript is not directly related to the Java language.

• **AJAX**: Asynchronous JavaScript and XML (AJAX) is a means through which a webpage can retrieve new data from the web server, without reloading the page [15]. There are countless uses for AJAX, including this project, as it allows for users of this system to view form previews without the hassle of page reloads.

• **jQuery**: jQuery is an open-source library written entirely in JavaScript [16]. Its motto is “Write less, do more.” By including the jQuery library with their applications, programmers have access to a wide variety of powerful functions, including AJAX, layout features such as date pickers, dynamic page manipulation, and more. Because jQuery works out and tests the details of these features in advance, programmers can typically invoke them with just one - two lines of JavaScript code.
• **JSP:** JavaServer Pages (JSP), released by Sun in 1999, is an extension to the core Java language. JSP expands Java into a language capable of generating dynamic web pages [17]. This allows for the full range of Java features and third-party extensions to be put to work in creating HTML tailored to individual client needs. A core feature of JSP is servlets, described below.

• **Servlet:** a servlet is a component of a JSP server responsible for receiving web requests, and issuing appropriate web responses [18]. These responses can consist of full-blown web pages, or smaller returns, such as AJAX data. Servlets provide a consistent interface for programmers to a wide variety of dynamic content from a single server instance.

• **Tomcat:** Tomcat is an open-source implementation of the JSP specification. It is developed and freely distributed by the Apache Software Foundation [19]. Tomcat allows administrators to configure the web server through either a graphic user interface (GUI), or XML configuration files. It is a popular choice for serving JSP sites, as it is lightweight and relatively easy to use.
Chapter 4
Overview of Form Creation Process

The heart of this project is its ability to convert a block of simple XML markup into full-throated HTML. This XML is simple enough that a person with no previous programming experience could quickly learn how to use it to create feature-rich and well laid-out HTML forms.

XML to HTML conversion is performed by Java code running on the server. The process takes place in the following stages. Edges labeled “C” indicate that the corresponding action is performed by the XML to Java compiler described in Chapter 5, and those marked with “G” are performed by the Java to HTML process described in Chapter 6.

- **User’s XML Markup**: the exact XML string entered by the user. The webpage that he uses to do this is described in Chapter 7.
- **XML Tokens**: through the lexical analysis process, the XML is parsed, and broken into Token objects. See details in Section 5.1.
- **Linked XML Tokens**: the Token objects are parsed via syntax analysis to ensure that their ordering and contents are valid. See Section 5.2.
- **HtmlElementBase Objects**: the Token objects are consolidated into HtmlElementBase objects. See Section 5.3.
- **Finalized HtmlElementBase Objects**: the HtmlElementBase objects are “finalized”; that is, attributes which are missing from the user’s input are supplied by the
system. (Note that this is not possible in cases where the user is required to specify the attribute value, such as the source location of a picture.) See Section 6.1.

- **HTML**: the HtmlElementBase objects are individually converted to HTML, and concatenated together to produce a final HTML output. See Section 6.2.

### 4.1 HtmlElementBase/CsunForm/FormElement

Consider the following XML markup, which is valid description of a form under this system’s specifications:

```xml
<csunForm>
  <button />
  <label>I am a label!</label>
  <textBox>You can type text right here...</textBox>
</csunForm>
```

Figure 4.2: XML markup of a simple form.

The system will view the XML just as a person typing onto a piece of paper would. The `cForm` can be seen as the piece of paper itself, with the `button`, `label`, and `textBox` elements placed inside it from left to right, beginning at the upper-left corner. Inside the Java code, the resulting hierarchy is a form object containing a list of element objects:

```
cForm
  button
  label
  textBox
```

Figure 4.3: Hierarchy of markup in Figure 4.2.
This code abstraction is obtained through three Java classes, one abstract, and two non-abstract:

- **HtmlElementBase (abstract)**: the parent class of all elements laid out in the form, as well as the form itself. HtmlElementBase is a child of abstract class XmlToJavaBase, which is explained in further detail in Section 5.3.2.

  A key field that HtmlElementBase inherits from XmlToJavaBase is its ElementType. ElementType is an interface for defining the type of element that an XmlToJavaBase can represent, such as a button type, text box type, checkbox type, and many others. In fact, ElementType can be implemented to represent any type, not just HTML types, as will be seen in Section 8.3.

  To satisfy their ElementType fields, HtmlElementBase objects use enum HtmlElementType, which implements ElementType. The enum members of HtmlElementType are listed below (with supporting code omitted):

  ```java
  protected static enum HtmlElementType implements ElementType {
      Button,
      CheckBox,
      ComboBox,
      ComboOption,
      CsunForm,
      Date,
      HorizontalLine,
      Label,
      Linebreak,
      Pagebreak,
      Picture,
      RadioButton,
      Space,
      TextBox;
      //Enum constructor and other functions omitted
  }
  
  Figure 4.4: HtmlElementTypeFactory.HtmlElementType enum members.
  ```

- **CsunForm extends HtmlElementBase (non-abstract)**: the class representing the form’s attributes (such as width, height, and margins), and its contained elements. The ElementType of a CsunForm is always HtmlElementType.CsunForm. A CsunForm contains zero to many FormElement objects, such as the example in Figure 4.3, but cannot contain another CsunForm.

- **FormElement extends HtmlElementBase (non-abstract)**: the class representing each element placed in the CsunForm, such as a button, text box, checkbox, and other
elements presentable in a web browser. A FormElement may have its ElementType equal to any HtmlElementType member, except for HtmlElementType.CsunForm. For a complete listing of how a FormElement is expressed in XML, and which FormElement types can be nested within one other, see Appendix B.
Chapter 5
XML to Java Compiler

The purpose of this phase is to take the user’s raw XML string, verify that it complies with XML structure that the system expects, and convert it to an equivalent sequence of XmlToJavaBase objects. As outlined in Chapter 4, the three steps of this are lexical analysis, syntax analysis, and element generation.

5.1 Lexical Analysis

Lexical analysis is performed by a new instance of the LexicalAnalyzer class. If the user’s XML markup is well-formed, the lexical analyzer breaks it into CompilerElements. Token class objects; otherwise, it notes the place in which the markup is malformed, and reports these to the user. There are seven possible forms of a Token object, each of which is represented by enum CompilerElements.TokenType:

- **XmlStartingTag**: a starting XML tag such as <from>, <email>, etc. The tag must begin and end with < and > characters, respectively. An XmlStartingTag may also contain attributes, as discussed in Chapter 3: XML.

- **XmlEndingTag**: an ending XML tag such as </from>, </email>, etc. The tag must begin and end with </ and > characters, respectively. An XmlEndingTag cannot contain attributes.

- **TextValue**: the text value placed between a starting and an ending tag. For example, in the markup <to>World</to>, the string World is the value token.

No delimiter tags are necessary for a TextValue token type, as it is identified as such due to being located between a starting and ending tag pair. Per W3C standards, a TextValue token cannot contain the characters listed in the following table outside of specially-escaped sequences. These are often referred to as “XML predefined entities” [20]:

<table>
<thead>
<tr>
<th>Predefined Entity</th>
<th>Must Replace With</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

Figure 5.1: Table of XML Predefined Entities.

- **XmlEmptyTag**: an XML tag such as <pic />, <input />, etc., in which an entire element is expressed through one tag. The tag must begin and end with <
and /> characters, respectively. Like an XmlStartingTag, an XmlEmptyTag may also contain attributes. While an XmlEmptyTag has the advantage of being more succinct, the disadvantage is that it cannot have child elements.

- **Comment**: a tag of the form <!-- Comment goes here -->; specifically, one beginning with the sequence <!--, and ending with the sequence -->. After being detected during lexical analysis, Comment tokens are not added to the master list of tokens found, and as such are subsequently ignored.

- **Whitespace**: a token consisting of the whitespace found between other tokens. The compiler allows the programmer to specify boolean preserveWhitespace, which, if true, concatenates Whitespace tokens onto their neighboring TextValue tokens. Otherwise, Whitespace tokens are discarded by the lexical analyzer [21].

- **Ambiguous**: a token not conforming to any of the above forms, and therefore one that is malformed. Lexical analysis will continue to completion if any Ambiguous tokens are found, but all subsequent compilation will not occur. Instead, all errors found during lexical analysis will be returned to the user.

Lexical analysis, while appearing to be an atomic operation from the view of outside calling classes, actually consists of two stages:

![Diagram](image)

Figure 5.2: Phases of lexical analysis.

### 5.1.1 Construct Tokens

The purpose of this phase is to construct an ordered list of all tokens indicated by the user’s inputted XML. This is accomplished through the deterministic finite automaton (DFA) shown below, which parses the XML character-by-character. State 1 is the DFA’s initial state. At all times, the lexical analyzer maintains a string of characters referred to here as Current. Current is used as follows:

- Whenever the DFA travels over one of the thick black edges shown in the DFA graph, the string that Current points to is added to the master token list, and Current is set to point to a new string whose contents are the character causing the edge traversal.

- Whenever the DFA shifts into State 1 from a different state, the string that Current points to is added to the master token list, and Current is set to point to a new empty string.

- For all other edges traversed, the character causing the traversal is concatenated to the end of Current.
The DFA also determines the TokenType of each token, for later use in both lexical and syntax analysis. Each time the DFA shifts state, the assumed TokenType of the current token is changed according to the following table.

<table>
<thead>
<tr>
<th>Edge Traversed</th>
<th>TokenType</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1)</td>
<td>Whitespace</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>Ambiguous</td>
</tr>
<tr>
<td>(1, 7)</td>
<td>TextValue</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>Ambiguous</td>
</tr>
<tr>
<td>(2, 6)</td>
<td>XmlEndingTag</td>
</tr>
<tr>
<td>(2, 8)</td>
<td>Comment</td>
</tr>
<tr>
<td>(3, 1)</td>
<td>XmlStartingTag</td>
</tr>
<tr>
<td>(3, 3)</td>
<td>Ambiguous</td>
</tr>
<tr>
<td>(3, 4)</td>
<td>Ambiguous</td>
</tr>
<tr>
<td>(3, 5)</td>
<td>XmlEmptyTag</td>
</tr>
<tr>
<td>(4, 3)</td>
<td>Ambiguous</td>
</tr>
<tr>
<td>(4, 4)</td>
<td>Ambiguous</td>
</tr>
<tr>
<td>(5, 1)</td>
<td>XmlEmptyTag</td>
</tr>
<tr>
<td>(6, 1)</td>
<td>XmlEndingTag</td>
</tr>
<tr>
<td>(6, 6)</td>
<td>XmlEndingTag</td>
</tr>
<tr>
<td>(7, 2)</td>
<td>TextValue</td>
</tr>
<tr>
<td>(7, 7)</td>
<td>TextValue</td>
</tr>
<tr>
<td>All edges originating at states 8, 9, 10, 11, and 12</td>
<td>Comment</td>
</tr>
</tbody>
</table>

Figure 5.4: Table of assumed TokenTypes.
Of course, users will not always enter well-formed XML markup. If the DFA, while in state \( S \), parses character \( c \) for which \( S \) has no matching outgoing edge, then \( c \) is concatenated to Current. Current is moved to the end of the master token list, and the DFA shifts to State 1. The malformed token will be detected in the Check Tokens phase.

5.1.2 Check Tokens

In this phase, the token objects collected in the master list during the Construct Tokens phase undergo the following two checks. These are performed on each token individually; the relationships between the tokens are not considered at this point.

- **Token syntax (for XmlStartingTag, XmlEndingTag, XmlEmptyTag, Comment, and Ambiguous tokens):** the token’s syntax is validated against its TokenType, which, as discussed above, is inferred from the final edge it traverses through the lexical DFA. For example, a token whose string is </label id="labeOne"> and inferred type is XmlEndingTag would fail this check, as no content except for the element name is allowed in an XmlEndingTag.

  Ambiguous tokens always fail this check, ensuring that the error will be reported back to the user.

- **Element type (for XmlStartingTag, XmlEndingTag, and XmlEmptyTag tokens):** the token’s declared ElementType is checked for validity against the list of valid ElementType members passed by the programmer as an argument to function CompilerElements.doLexAndSyntaxAnalysis(). For example, while <chocolate /> is a well-formed XmlEmptyTag, the type chocolate is not (currently!) supported by the system.

  If the token’s ElementType is valid, and its TokenType is XmlStartingTag or XmlEmptyTag token, the token is also checked for conformity to the TokenType expected for the ElementType. For example, because the system requires that button be inputted as an XmlEmptyTag, the user-inputted token <button> will fail this check (as it is wrongly structured as an XmlStartingTag). Similarly, the non-empty type textBox will reject the token <textBox />.

Figure 5.5: Lexical errors are caught and thrown back to the user.
All errors detected in the above two checks are added to a list of errors. If the compiler sees that the error list is not empty, it will abort the entire compilation process, and return the list of errors to the user. Each error message includes the line/character number at which malformed token starts; the line/character number at which it ends; and an explanation of the problem, so the user can make appropriate corrections.

Otherwise, if no errors are detected, any tokens found which are NOT XmlStartingTag, TextValue, XmlEndingTag, or XmlEmptyTag types are removed from the master list. The list is then passed to the syntax analyzer.

5.2 Syntax Analysis

Syntax analysis is performed by a new instance of the SyntaxAnalyzer class. The purpose of syntax analysis is ensure that the list of tokens returned from lexical analysis form a valid XML document. It also links each XmlStartingTag token with its corresponding XmlEndingTag token. Like lexical analysis, syntax analysis appears to be an atomic operation when called from outside, but in reality consists of the following phases:

5.2.1 Check Root Tokens

As discussed in Chapter 3: XML, all XML documents must have a root element. Therefore, it is required that the starting tag of this root element be the first non-comment XML tag in the markup, and its ending tag be the last non-comment tag. These two requirements are now verified against the list of tokens returned from lexical analysis. If so, the two tokens are removed, and the starting token is saved for future reference (described in below in Section 5.3.1). Otherwise, an error message is logged. Syntax analysis will continue to completion if either token is missing, but subsequent compilation will terminate after syntax analysis, and the list of all errors found will be returned to the user.

For the purposes of form generation, <cForm> and </cForm> are the starting and ending root tokens, respectively. However, this does not always have to be the case, as the compiler is in fact ignorant of the form generation system. This feature is further explored in Section 5.3.2.

5.2.2 Check Other Tokens

The list of tokens, which at this point does not have root element tokens at its beginning or end, is then parsed from beginning to end by the pushdown automaton (PDA) diagrammed below. The PDA begins at State 1, and, with each successive token, attempts to find an edge traveling out from the current state matching the token’s TokenType. If the
PDA finds a match, it shifts to the state indicated by the edge. If no match is found, an appropriate error is logged, and, like the lexical analyzer, the PDA shifts to State 1.

As can be seen in the PDA graph, several edges also trigger a stack push or pop:

- For edges specifying a push, a reference to the token which caused the edge to be traversed will be pushed to the stack. The type of this token will always be `XmlStartingTag`.

- Correspondingly, edges specifying a pop are always traversed by an `XmlEndingTag`. The pop will be performed without logging an error only if an `XmlStartingTag`, with the same `ElementType` as the `XmlEndingTag`, is present on top of the stack. Upon popping the matching `XmlStartingTag`, the PDA will set a reference to the ending tag in the starting tag (for reasons examined in the next section).

  Note that if the user’s XML is well-formed, the matching `XmlStartingTag` will always be on top of stack, but in case it is not, the code will actually descend through the stack until a match is found, in order to generate a more accurate error message.

Once all tokens in the master list have been parsed by the PDA, the syntax analyzer checks if the stack is empty. If not, an error message is generated for each `XmlStartingTag` token in the stack, informing the user that the token has no matching `XmlEndingTag` token. These errors, along with any others found at any point during syntax analysis, are then immediately thrown back to the user.

The PDA is formally defined as follows:

- \( Q = \{ S_1, S_2 \} \)

- \( \Sigma \) = the finite set of all `XmlStartingTag` tokens (without any attributes) defined by the programmer, not including the root element token. Each token is treated as an atomic unit. For HTML form generation, \( \Sigma = \{ <\text{label}>, <\text{link}>, <\text{textBox}>, <\text{comboBox}>, <\text{comboOption}>, <\text{date}> \} \).

- \( \Gamma = \Sigma \cup \{ z \} \)
- $q_0 = S_1$
- $F = \{ S_1 \}$

- **Transition function:**
  - Define $c$ as the token currently on top of the stack (or $z$ if the stack is empty).
  - An `XmlStartingTag` can only be removed from the stack by an `XmlEndingTag` of the same `ElementType`. For example, `</label>` will remove `<label>` from the stack, while `</textBox>` will not.
  - $\delta(S_1, <XmlStartingTag>, c) = \{(S_2, <XmlStartingTag>c)\}$
  - $\delta(S_1, </XmlEndingTag>, c) = \{(S_1, \lambda)\}$
  - $\delta(S_1, <XmlEmptyTag/>, c) = \{(S_1, c)\}$
  - $\delta(S_2, <XmlStartingTag>, c) = \{(S_2, <XmlStartingTag>c)\}$
  - $\delta(S_2, TextValue, c) = \{(S_2, c)\}$
  - $\delta(S_2, </XmlEndingTag>, c) = \{(S_1, \lambda)\}$

- **PDA Grammar:** the PDA is based on the following context-free grammar:
  - $S \rightarrow CS | <XmlEmptyTag/>S | \lambda$
  - $C \rightarrow <XmlStartingTag>C</XmlEndingTag>C | <XmlStartingTag>string literal</XmlEndingTag>C | \lambda$
  - The `<` and `>` arrows are not grammar meta-characters; they are the arrows that a user would include in his/her XML markup.
  - The `string literal` in rule $C$ may contain any characters and/or whitespaces, except for the predefined entities listed in Figure 5.1.
  - This grammar is based on the W3C XML Standard [22].

### 5.3 Element Generation

Reaching this point without any errors found means that the following are true:

- Each token inputted by the user is well-formed.
- Each token contains a valid element name.
- The sequence formed by the tokens is valid.

The system now converts the list of tokens into equivalent Java objects. First, the details of attribute verification and class `XmlToJavaBase` are explained, followed by an overview of the Java object generation process.
5.3.1 Attribute Verification

As discussed in Chapter 3, XML and HTML tags often include one or more attributes in their tags, which provide more information or context about the element being described. In the tag `<movie region="US">`, the string `region="US"` is an attribute, `region` is the attribute name, and `US` is the attribute value. The system expects attributes to always be expressed in this syntax, with the value being surrounded by double-quotes.

In order to keep the compilation process independent of the form generation process, the compiler is not programmed with a list of specific attributes it is capable of accepting. Instead, the compiler requires the programmer to supply an array of allowed Attribute members to function `CompilerElements.doLexAndSyntaxAnalysis()`, just as this function also requires an array of `ElementType`. Each member of the attribute array must implement the `AttributeFunctions.Attribute` interface.

Before this point, the system has not examined the attributes of any tokens. In a process referred to as attribute verification, performed by function `XmlToJavaConverter.addAttributesToElement()`, the attributes of each token are now checked and set as follows:

- A new map `foundAtts`, which maps an Attribute to a string value, is instantiated.
- The string consisting of all the token’s attributes is broken into a list of single attributes. For example, `width="11.0" height="8.5" marginTop=".5"` is broken into list `[width="11.0", height="8.5", marginTop=".5"]`.
- Each string in the list is then checked for conformity to the `name="value"` format. If so, the following is then verified:
  - `name` is an Attribute found in the Attribute list designated by the programmer.
  - An Attribute of type `name` is permitted as an Attribute for the `ElementType` of this token. The XML `<pic name="pic1" />` would cause an error to be...
thrown, as element `pic` does not allow the `name` attribute.

- The attribute value is valid for an `Attribute` of type `name` when used within this `ElementType`. For example, `width=".5hello"` will throw an error, as “.5hello” is not a valid floating-point number. This check for correctness is done via interface `DataValidator`, a concrete instance of which is set by the programmer in each `Attribute`.

- This `Attribute` currently has no mapping in `foundAtts` (i.e., no duplicate attributes). Accordingly, `<radio id="myRadio" id="yourRadio" />` would be rejected.

- If the above four checks are passed, a mapping of this `Attribute` to its discovered value is added to `foundAtts`.

  - Once all attribute names and values have been added to `foundAtts`, the system verifies that all required attributes for the token’s particular `ElementType` have been inputted. For example, a `pic` element requires that the user specify attribute `source`, so that the picture can be displayed on the webpage.

  - If any of the above steps fail for any reason, the system logs the error to a list which will later be thrown back to the user, and continues on to the next token.

### 5.3.2 Class `XmlToJavaBase`

While the system’s intended usage is the generation of HTML forms, care was taken to make the XML compiler be agnostic of this fact. As a result, no references to `HtmlElementBase`, `CsunForm`, or `FormElement` are present in package `edu.csun.forms_site.compiler`. Instead, the XML compiler directly handles only one class, `XmlToJavaBase`.

`XmlToJavaBase` contains fields which mirror the structure of an XML tag: a single instance of `ElementType`, and a map of `Attribute` objects to their values. Note that class `AttributeContent` is a container capable of holding either a string, or an `ArrayList<XmlToJavaBase>`, but not both. `AttributeContent` is explained in Section 9.1.1.

```java
Table 5.9: UML representation of class `XmlToJavaBase`.

<table>
<thead>
<tr>
<th><code>XmlToJavaBase</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>elementType : ElementType</code></td>
</tr>
<tr>
<td><code>attributes : HashMap&lt;Attribute,AttributeContent&gt;</code></td>
</tr>
<tr>
<td><code>XmlToJavaBase(ElementType)</code></td>
</tr>
<tr>
<td>//Functions for getting/setting values in the HashMap</td>
</tr>
</tbody>
</table>
```

`XmlToJavaBase` is the immediate parent of `HtmlElementBase`, and as such is the grandparent of `CsunForm` and `FormElement`. To generate concrete instances of these two classes, interface `XmlToJavaInstantiator` is used:
public static interface XmlToJavaInstantiator {
    public XmlToJavaBase instantiateXmlToJavaBase(ElementType elementType);
}

Figure 5.10: Interface XmlToJavaInstantiator.

Each instance of ElementType carries its own instance of XmlToJavaInstantiator. This allows the compiler to generate XmlToJavaBase descendants of the class needed for the ElementType, while simultaneously keeping the compiler free of references to classes outside its package. For example, to generate FormElement objects, the following anonymous implementation of XmlToJavaInstantiator is used:

private static final XmlToJavaInstantiator FORM_ELEMENT_INSTANTIATOR =
    new XmlToJavaInstantiator() {
        @Override
        public XmlToJavaBase instantiateXmlToJavaBase(ElementType elementType) {
            return new FormElement(elementType);
        }
    };

Figure 5.11: XmlToJavaInstantiator FORM_ELEMENT_INSTANTIATOR.

By using XmlToJavaInstantiator, the compiler package remains blissfully unaware of the exact class of each non-abstract object it generates. This allows for the possibility of using the compiler for non-HTML purposes in the future.

5.3.3 Generation Steps

Element generation is accomplished via the following steps, each of which is described in further detail below:

Generate Root Object

Generate Child Objects

Nest Children in Root

Figure 5.12: Phases of element generation.

5.3.3.1 Generate Root Object

As noted in Section 5.2.1, the XmlStartingTag root token found during syntax analysis is set aside during the Check Root Tokens phase. If syntax analysis is successful, a new root object is instantiated, and the attribute values listed in the token are loaded into the root via function XmlToJavaConverter.addAttributesToElement(). For the purposes of the form system, this root is a CsunForm object.
5.3.3.2 Generate Child Objects

Next, child objects of the root element are generated by inputting the tokens list to function `XmlToJavaConverter.tokensToElements()`, expressed in pseudocode in Figure 5.15 below. This function is a recursive-descent parser which implements the context-free grammar listed in Section 5.2.2. The “big picture” of the function is to do the following:

- For each token in the tokens list:
  - If the token is an `XmlEmptyTag`, move its `ElementType` and `Attribute` values directly into a new descendant of `XmlToJavaBase`. The exact class of this descendant is determined by the `XmlToJavaInstantiator` of the `ElementType`. For example, the empty tag `<button width=".5" value="Click Me"/>` will transformed into the following `FormElement` object:
    
    ![Figure 5.13: FormElement generated from the button XML markup.](image)

  - Else if the token is an `XmlStartingTag`, move its `ElementType` and `Attribute` values, except for `value`, into the `XmlToJavaBase` generated by the `XmlToJavaInstantiator`. Afterwards, `value` is recursively set to the tokens sandwiched between this `XmlStartingTag` and its corresponding `XmlEndingTag`. According, the markup `<label height="1.2">Label Me</label>` is transformed into:
    
    ![Figure 5.14: FormElement generated from the label XML markup.](image)

- Place all these generated `XmlToJavaBase` objects into a single `AttributeContent` object, and return it.

5.3.3.3 Nest Children in Root

Finally, the child objects extracted in function `tokensToElements()` are nested in the root `XmlToJavaBase` object generated in Section 5.3.3.1. Constraints are written into the
code to ensure that the root object is capable of having nested child elements: i.e., the root object is not expressed as an XmlEmptyTag, and the root object has an Attribute pre-designated for containing the child elements. In the case of the form system, this attribute is HtmlAttribute.Value.
public AttributeContent tokensToElements(List<Token> childTokens) {
    if (childTokens has exactly one token, and it’s a TextValue type token )
        return the token’s string value wrapped in a new AttributeContent;
    else {
        Create new List<XmlToJavaBase> returnList;

        for( Token nextToken : childTokens ) {
            ElementType elementType = nextToken’s ElementType;
            XmlToJavaInstantiator instantiator = ElementType’s
                XmlToJavaInstantiator;

            if( nextToken is an XmlEmptyTag token ) {
                Use instantiator to create new XmlToJavaBase newElement,
                    whose ElementType is that of nextToken;
                Check attributes of nextToken are valid (per Section 5.3.1),
                    and load them into newElement;
                Add newElement to returnList;
            }
            else if( nextToken is an XmlStartingTag ) {
                Use instantiator to create new XmlToJavaBase newElement,
                    whose ElementType is that of nextToken;
                Check attributes of nextToken are valid (per Section 5.3.1),
                    and load them into newElement;
                AttributeContent childContent;
                if( nextToken has child tokens ) {
                    Create new List<Token> childTokens;
                    Add all child elements of nextToken to childTokens;
                    childContent = tokensToElements(childTokens);
                }
                else
                    childContent = new AttributeContent('""'); //empty string

                Add childContent to newElement;
                Add newElement to returnList;
                Skip ahead in the for loop, so the next iteration starts on the
                    token after nextToken’s matching XmlEndingTag token;
            }
            //else: XmlEndingTag and TextValue types are handled in ‘else if’
        }
        return returnList wrapped in a new AttributeContent;
    }
}
Chapter 6
Java to HTML Generator

Reaching this point means that the user has entered valid XML markup per the system specifications, and that the equivalent CsunForm/FormElement objects have been successfully instantiated. Our goal from this point forward is to convert these objects into HTML.

6.1 Data Finalizer

Before HTML generation, however, we first take advantage of one of the key features of the system: its ability to fill in certain attribute values for CsunForm and FormElement objects, so that the user does not have to.

When the programmer specifies concrete instances of interface ElementType— which, in the case of the form generation system, are the members of enum HtmlElementType— he must define several fields to satisfy ElementType’s method signatures. One of these is an array of Attribute instances, designating which Attribute instances are permitted for an HtmlElementBase of this ElementType. Before the HtmlElementBase is converted to HTML, all of these Attributes must have a corresponding value.

However, in all but a few cases, the value does not have to be user-specified. Instead, if the system sees that an Attribute not required to be set by the user has not yet had its value defined, function finalizeAttributes() will see to it that a value is provided. This can be either a default value, such as today’s date for a date field, or an intelligent guess, such as calculating .4 inches as the width of the string “music” when set in non-bold Times New Roman 12pt type.

To ensure that it is possible to generate values for any Attribute, constructing a concrete instance of Attribute requires that the programmer specify an instance of the ValueGenerator interface. ValueGenerator requires the implementation of a single function:

```java
1  public static interface ValueGenerator {
2    public AttributeContent generateValue(HtmlElementBase element,
3                                               CsunForm csunForm);
4  }
```

Figure 6.1: Interface ValueGenerator.

If the system determines that a given HtmlElementBase is missing a value for an Attribute, generateValue() for the attribute’s ValueGenerator is called. Because one of the parameters to generateValue() is the HtmlElementBase itself, the programmer can customize exactly how the value will be generated.

To see this in practice, consider the HtmlElementType.Radio element, whose XML tag is radio. This element allows for attributes id, value, width, height, checked,
and name. Of those, only name must be inputted by the user; attempting to create a radio without specifying a name results in an XML compiler error (without this requirement, the web browser would be unable to associate radio buttons into groups). If the user does not specify any of the other attributes, they will be automatically generated by the system as follows:

- **Id**: if the user does not specify an id for a radio or any other element, the system will generate an id unique to the currently-processed CsunForm.

- **Value**: when the programmer specifies a new HtmlElementType, he must designate a way to supply a default for HtmlAttribute.Value (see Section 8.1 for how this designation is done). Currently, all HtmlElementType members have an empty string set as the default value, except for date, which will have today’s date supplied; and button, which has “Button Value”.

- **WidthInches**: because radio represents a radio button, which can have an adjoining text label but no text itself, the system currently returns a constant value for the width of a radio button (approximately .16 inches). This is done by the WidthCalculator object stored in each instance of HtmlElementType. Like ValueGenerator, WidthCalculator is an interface, one that specifies how to estimate the width of an element when displayed in the user’s browser:

```java
public static interface WidthCalculator {
    public double calcWidthInches(HtmlElementBase element, CsunForm csunForm);
}
```

Figure 6.2: Interface WidthCalculator.

While no involved calculations are needed for radio elements given that a constant width is always returned, elements containing text, such as label, textBox, and button, each contain a WidthCalculator programmed to estimate the width of the text in inches. This is accomplished by using a java.awt.Graphics2D object, which contains function stringWidth(String toMeasure) for estimating toMeasure’s width in inches. The Graphics2D object takes into account if toMeasure is bold and/or italicized.

Note that, like any other attribute, WidthCalculator is only called if the user does not specify the width attribute his or herself. If he specifies a valid number for the width, the system will use that value.

- **HeightInches**: like the radio’s width, the height of a radio button is also estimated as a constant value in inches. And, like width, this estimate is provided by an instance of an interface, this time one called HeightCalculator. This interface is identical to WidthCalculator, except its single function is called calcHeightInches().
For elements such as label or button which contain text, the height of the text is typically estimated as follows:

```
1  Let totalWidth = width in inches of the text as printed on a single line
2       (either entered by the user, or estimated by the system);
3  Let elementWidth = width in inches of the label, button, or other object
4       that the text belongs to;
5  Let height = totalWidth / elementWidth;
6  return height * a constant greater than one (to avoid underestimates);
```

Figure 6.3: Pseudocode for estimating text height.

- **Checked**: the system will by default return an empty string for this attribute, indicating to the user’s browser not to check the radio button.

Similar means for generating values are found in the ValueGenerator instances present in all other Attributes.

### 6.2 HTML Generation

We have finally reached the point at which the HtmlElementBase objects can be converted to HTML. There are two main steps in this process:

- Create an HTML div matching the dimensions specified by the CsunForm object created during element generation. This div is essentially the “paper” on which the elements are displayed; the white background in Figure 6.4 is an example. Additional divs are created as needed to accommodate additional pages.

- Convert each FormElement element to HTML, and concatenate this HTML into a single string beginning with the CsunForm’s HTML div.

#### 6.2.1 CsunForm to HTML

As seen in Appendix B, attributes of a CsunForm (cForm in the XML markup) include the form’s width, height, and its four margins. The system retrieves these values for the CsunForm object created in the element generation phase, and creates what is essentially the HTML equivalent of a blank page. For example, if a user specified a page width of 8.5 inches, height equal to 11.0 inches, half-inch margins on the left and right, and .75 inch margins at the top and bottom, the resulting image in the user’s browser would be similar to Figure 6.4, seen below.
**Figure 6.4:** Resulting page of an 8.5 x 11” cForm.

The entire structure seen in Figure 6.4, including the black border, consists of three HTML divs structured as follows:

```html
1 <div> <!-- Div with the black border -->
2  
3 </div>
```

**Figure 6.5:** HTML div structure for a page in generated by the system.
The width of the innermost div is equal to the page width minus the left margin minus the right margin, and similarly, the height is equal to the page height minus the top margin minus the bottom margin. It is inside this inner div that the HTML of the FormElement objects is placed, as seen in the next section.

6.2.2 FormElements to HTML

We have finally reached the last step in the form creation process. Now that the basic outline of each page of the form’s HTML has been defined in the three divs of Figure 6.5, we fill the innermost div with the HTML of the FormElement objects. The toHtml() function of each FormElement supplies its HTML equivalent. This is done by taking the HtmlElementType’s equivalent HTML tag, and concatenating it to the HTML equivalents of the FormElement’s attributes. CSS attributes are first arranged into a single style="..." string. If the FormElement has child elements, their HTML is determined recursively, and sandwiched in-between this element’s starting and ending tags.

To layout the elements into properly sized rows and pages, the process works as follows. As shown in Figure 4.3, the system begins in the upper-left corner of the usable area of the page, and proceeds to layout elements from left to right. When the system sees that the next element to be placed is unable to fit in the current horizontal row of elements (managed by class FormRow), it begins a new one. To make this determination, the system adds the value of the element’s width attribute, which at this point has been set either by the user or the system, to the row’s total width. If this sum exceeds the innermost div’s width, the system starts a new row. Note that the user can also create a new row at any time via a \(<\text{linebreak}/>\) element.

![Figure 6.6: Elements are laid-out left to right; new row is started when previous is full.](image)
A similar process is used to determine when to create new pages, this time based on element height. The system considers the height of a FormRow to be the height of the tallest FormElement in the row. After the row is filled with FormElement objects, but before it is added to the current FormPage (a container for all FormRow objects of a given page), the system determines if adding the row to the page would cause the combined height of all rows on the page to exceed the height of the innermost div. If so, a new FormPage is instantiated. The user can also start a new page at any time using a `<pagebreak/>` element.

However, the above description omits a key detail. When the system lays out elements, it inserts more than just the FormElement objects generated from the user’s XML. Since a goal of the system is to generate nicely laid-out forms with a minimum of user effort, the system also inserts a small space between adjoining elements if the user does not do so. Currently, the width of this space is set to a default width 1/32 of an inch. If the user wishes to negate adding the space at a particular spot, he must place `<space width="0"/>` in his/her XML.

The system also places a space element between adjoining rows, whose width is equal to the innermost div, and height is equal to the default height of 1/32 of an inch. It is not possible to remove this row padding, but users can augment it by specifying `<space width="W" height="H"/>`, where W equals the desired width of the layout area, and H equals the desired additional height.

![FormElement button] S [FormElement label] S [FormElement textBox]

![FormElement radio] S [FormElement checkbox] S [FormElement comboBox]

Figure 6.7: Space is automatically inserted if the user does not specify it.

Once the HTML of all elements, including system-inserted space elements, has been concatenated into a single string, it is displayed in the user’s browser, as discussed in the following section.
Chapter 7
User Controls

This chapter details the XML elements and attributes that users would typically use in form creation, and how forms are actually created via the Form Generator page.

7.1 XML Usage

The following is a list of all XML elements supported by the system, and how they might be used by form designers. The exact details of which attributes are supported by each element are specified in Appendix C, but any attributes meriting special attention are also mentioned in this list.

- **cForm**: as seen in Section 5.2.1, all forms must be surrounded by `<cForm [attribute]>` tags in order to compile. Additionally, the form designer must specify the `width`, `height`, `marginTop`, `marginRight`, `marginBottom`, and `marginLeft` attributes. In prototypes of this project, these values were set to hard-coded defaults if not specified by the user. However, it was later realized that if these defaults ever changed, the layout for any existing forms would likely break. Thus, the safer approach of requiring designers to specify these attributes upfront was adopted.

  As an example, the form in Figure 6.4 has the following `cForm` tags:

  ```xml
  <cForm width="8.5" height="11" marginTop=".75" marginRight=".75"
  marginBottom=".75" marginLeft=".75">
  <!-- Label content goes here, omitted in this listing. -->
  </cForm>
  ```

- **label**: used to place static text in a form that provides information or instructions to users. A label often designates the intended purpose of other elements. For instance, the XML `<label>DATE OF EMPLOYMENT</label>` yields the following label for placement next to a `date` element:

  ```xml
  <DATE OF EMPLOYMENT
  ```

- **pic**: used to place static pictures in a form. Designers are required to provide the path to the picture via the `source` attribute, and an alternate label via the `alt` attribute should the picture fail to load. They must also provide `width` and `height` values, so that `CsunForm.toHtml()` can properly size the picture for layout on the page. For instance, assuming that the desired image is at path “images/CSUN_logo.png”, the markup `<pic source="images/CSUN_logo.png" alt="CSUN Logo" width="1.5" height=".5" />` yields the following picture:
• **textBox**: allows users to input text, in amounts from one word to several paragraphs. Examples include inputting first name, last name, address, or a list of composers that a music student will perform at his recital. Using the `height` attribute, form designers can allow for multiple lines of input, and with `maxChars` can limit the amount of input. For example:

```xml
  <textBox width="1.75" height=".5">
    Johann Sebastian Bach
    Wolfgang Amadeus Mozart
    Ludwig van Beethoven
  </textBox>
```

• **checkBox**: used to make users explicitly indicate approval of something. Examples include agreement with rules, terms, or approval to charge a fee to the user’s CSUN account. When used in conjunction with a `label`, the XML `<checkBox />
<label>I agree!</label>` appears as:

![I agree! checkbox](image)

• **comboBox/comboOption**: used to make people choose a single value from a list, typically a large one (note that the picture below shows a small list due to space considerations). Examples include choosing a home state or country. At least one child `comboOption` must be nested under the parent `comboBox`, as seen below:

```xml
  <comboBox width="2">
    <comboOption>(select one)</comboOption>
    <comboOption>Comprehensive Exam</comboOption>
    <comboOption>Thesis/Project/Artistic Abstract</comboOption>
  </comboBox>
```

• **radio**: like a `comboBox`, a group of `radio` elements is used to make people choose a single value from a pre-defined list. `radio` elements are a better choice for smaller lists, as the user is able to view all options at a glance. The designer must set the `name` attribute of each `radio`, so that the web browser knows how to associate
them into groups (of which only one radio can be selected at a time). Examples include choosing one’s gender, faculty status, or, as seen below, payroll type:

```xml
1  <radio name="payroll" checked="checked" value="univesityCorp"/>
2  <label>University Corp</label>
3  <radio name="payroll" value="assocStudents"/>
4  <label width="2.5">Assoc. Students</label>
5  <linebreak/>
6  <radio name="payroll" value="state"/>
7  <label width="1.2">State</label>
8  <radio name="payroll" value="USU"/>
9  <label width="2.5">USU</label>
```

- **date**: used to allow people to indicate a date, using a GUI date picker. Usage could include entering an intended date of admission, graduation, or final exam. If the designer leaves the TextValue between the starting and ending tags blank, the system automatically inserts today’s date. For example, on April 27, 2013, the tags `<date></date>` will result in:

![Date picker](image)

- **button**: when submission of forms to the server is implemented in future phases of this project (described in Chapter 2), users will do so by clicking on a button. The XML `<button value="Submit Form"/>` is all that is required to create the following button:

![Submit Form button](image)
• **link**: used to direct people to other webpages. This is useful when an existing webpage—such as a department homepage, list of requirements, or non-CSUN page—has information needed by the user before form submission. The markup `<link linkTo="http://www.mywebpage.com">Home Page</link>` results in:

```
Home Page
```

• **hLine**: creates a horizontal line used to delimit areas of the form from each other. The `bgColor` attribute is used to set the line’s color. If the designer does not specify `width`, the system sets the width to the available width of the page (which does not include margins). Accordingly, the markup `<hLine bgColor="red">` results in the following line drawn across the page:

![Horizontal Line](image)

• **space**: the `space` element creates an empty area of a given `width` and `height`. `space` is key for designers to obtain the exact look they desire for the form. To create horizontal space between elements in the same row, designers should use a `space` that has only a `width` attribute:

```
1  <button value="I’m a button!''/>
2  <space width="1"/>
3  <label>A label one inch from the button!</label>
```
To specify vertical padding between rows, designers should specify a `space` with `width` equal to the available page width, and `height` equal to the desired padding size:

1. `<label>First row</label>` `<button value="Button 1"/>`
2. `<space width="10" height="1.5"/>`
3. `<label>Second row</label>` `<button value="Button 2"/>`
4. `<space width="10" height=".5"/>`
5. `<label>Third row</label>` `<button value="Button 3"/>`

bottom edge
• **linebreak:** placing a linebreak in the XML causes the following element to start on a new row. Like space, linebreak is another important tool for the form designer. Had a linebreak been placed between the button and label in Figure 6.6’s markup, the result would have been:

![Example of linebreak](image.png)

• **pagebreak:** the final layout tool at the designer’s disposal is the pagebreak element, which, as its name suggests, causes the following element to be placed on a new page. Had a pagebreak been placed between the button and label in Figure 6.6’s markup, the result would have been:

![Example of pagebreak](image.png)
7.2 Webpage Interface

In order to facilitate form creation, a webpage was created for users to input XML to the system, view error messages if the XML is malformed, and otherwise view the form indicated by the XML. The page is designed with a minimum amount of controls to make for an easy user experience. It also has virtually no CSS specified, so as not to override the generated form with unintended styling.

Upon loading the page, the user sees the following simple interface:

![Form compilation page on load.](image)

The user then inputs his/her XML markup into the upper text box, and clicks “Compile and Preview” to generate the form. (See Appendix A for examples of XML markup and their corresponding generated forms.) This submits an AJAX request to the server, which responds with either the form’s generated HTML, or an error:

- If the XML is well-formed, the form appears above the XML input box. Several other controls will appear as well:
  - **Print**: displays the form by itself in a new page, and calls the JavaScript window.print() function.
  - **Show Form HTML**: clicking this link will reveal a read-only text box containing the HTML of the form generated. Clicking the link again will hide this text box.
  - **Show HTML Template**: clicking this link reveals the contents of an HTML file, into which the user can paste the HTML revealed in the above step. The
file also includes the necessary CSS and JavaScript links needed to provide full-functionality for the form (such as enabling jQuery date pickers).

- After the form has been loaded, the page will execute any JavaScript code needed to provide functionality for the form. Currently, this consists of searching for all elements of class “dateElements” (which the system automatically inserts into the HTML of all date elements), and activating them as jQuery date pickers.

- Otherwise, if the XML has any errors, a detailed message for each error will appear in the lower text-box. Each message will include the starting line/character, ending line/character, and explanation for the specific error.

<table>
<thead>
<tr>
<th>Compilation Errors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR: Line 38, char 12 to line 38, char 23: Value contains illegal &gt; character(s). Use &gt; instead</td>
</tr>
<tr>
<td>ERROR: Line 76, char 6 to line 76, char 12: banana is not a valid element name</td>
</tr>
<tr>
<td>ERROR: Line 108, char 50 to line 109, char 43: Ending tag must be &lt;/label&gt;</td>
</tr>
</tbody>
</table>

Figure 7.2: Compilation errors are detailed for user review.
Chapter 8

Adding ElementTypes/Attributes

Realizing that others will likely wish to add or modify allowable XML markup in the future, the code was setup to make this as easy as possible. In order to add a new form element to the system— for example, a tag `<video>` to match the HTML5 video element— a programmer simply needs to specify it in enum(HtmlElementType). Similarly, to add a new attribute, a new member must be added to enum(HtmlAttribute). The details of each are explained below.

8.1 HtmlElementType

(HtmlElementType) resides in class HtmlElementTypeFactory, and implements interface ElementType. The constructor for HtmlElementType requires the programmer to pass in instances of two wrapper objects: an ElementTypeFunctions.GeneralElementTypeFields object, and an ElementTypeFunctions.HtmlElementTypeFields object.

Because these wrapper objects cannot be constructed without providing all field values to their constructor, this eliminates the possibility that future programmers working on the system can forget to provide all information necessary for them to work properly. Specifically, the following fields must be defined for each wrapper class:

- **GeneralElementTypeFields**:
  - **String xmlTag**: XML tag name of the associated ElementType. For example, in the case of `<radio/>` elements, this would be “radio”.
  - **boolean isEmptyXmlElement**: true if the associated ElementType is treated as an empty XML element within this system; else false. For example, this Boolean is true for the `<pic/>` type, and false for the `<label/>` type.
  - **AttributeWrapper[] allowedAttributes**: array of AttributeWrapper objects designating which attributes are permitted and required for the associated ElementType. An AttributeWrapper is a class consisting of two required fields: an Attribute, and a Boolean. If this Boolean is true, the user must specify the associated Attribute in his/her markup.
  - **ElementType[] allowedChildTypes**: array of ElementType instances, used to indicate are permitted as nested children of this parent ElementType. For most HTML elements, this is empty, as they are either empty elements and cannot have children, or their “child” is a TextValue token. One exception is the `comboBox` element, for which allowedChildTypes = { HtmlElementType.ComboOption }.
  - **XmlToJavaInstantiator instantiator**: XmlToJavaInstantiator for creating non-abstract descendants of XmlToJavaBase for this ElementType. See Section 5.3.2.
• **HtmlElementTypeFields**

  o **String htmlTag**: equivalent HTML tag name of the associated ElementType. For example, the HTML equivalent of the system’s `pic` element is `img`.

  o **boolean isEmptyHtmlElement**: true if the associated ElementType is treated as an empty element in HTML; else false. Note that this does not need to be the same as the boolean `isEmptyXmlElement` value set above.

  o **DefaultGenerator defaultGenerator**: anonymous object which generates a AttributeContent instance for the HtmlAttribute.Value attribute, in case the user does not specify it in his/her markup. For example, the `date` element has a DefaultGenerator which returns today’s date wrapped in an AttributeContent.

  o **boolean hasMultipleTextLines**: true if this element potentially could contain multiple lines of text; else false. This needs to be known for the height of the element to be estimated properly.

  o **String extraCss**: extra CSS to be automatically added to the HTML rendering of objects of the associated ElementType. For example, several types have `extraCss = "float: left;"` to improve positioning.

  o **String extraHtml**: extra HTML to be automatically added to the HTML rendering of objects of the associated ElementType. For example, `date` elements have `extraHtml = "class=dateElements"`, so that these elements are set as jQuery date pickers on form load.

  o **WidthCalculator widthCalculator**: WidthCalculator for the associated ElementType. See Section 6.1 for details.

  o **HeightCalculator heightCalculator**: HeightCalculator for the associated ElementType. See Section 6.1 for details.

• **HtmlAttribute**

  Similarly, each member of enum HtmlAttribute requires that an instance of the AttributeFunctions.GeneralAttributeFields class, and an instance of the HtmlAttributeFactory.HtmlAttributeFields class, be passed to its constructor. Their fields are as follows:

  • **GeneralAttributeFields**:

    o **String xmlTagName**: XML tag name of the associated attribute. For example, in the case of `marginLeft="1.5"`, this would be “marginLeft”.

    o **DataValidator validator**: DataValidator for the associated attribute. See Section 5.3.1.

  • **HtmlAttributeFields**

    o **String htmlString**: Equivalent HTML tag name of the associated attribute. For example, the HTML equivalent of the system’s `source` attribute is `src`.

41
o **ValueGenerator generator**: ValueGenerator for the associated attribute. See Section 6.1.

o **CssFormatter cssFormatter**: CssFormatter for the associated attribute (or null if this is not a CSS attribute). CssFormatter is an interface for converting a value into its CSS equivalent, including the appropriate CSS unit at the end. Currently, a concrete instance of CssFormatter is used to convert width and height values in inches to their equivalents in pixels.

### 8.3 Generating Non-HTML Content

As mentioned in 5.3.2, the system’s XML compiler was designed for the possibility of expanding to other outputs than HTML in the future. Interfaces ElementType and Attribute were also designed with the same goal in mind. To facilitate this, ElementType is structured as follows:

```java
public static interface ElementType {
    public GeneralElementTypeFields getGeneralElementTypeFields();
    public SpecificElementTypeFields getSpecificElementTypeFields();
}
```

Figure 8.1: Interface ElementType.

Notice that ElementType does not actually reference class HtmlElementTypeFields. Instead, it references SpecificElementTypeFields, an empty, abstract class which is the immediate parent of HtmlElementTypeFields. A programmer could use this fact to create a new ElementType whose end-product is not HTML. To do so, in addition to an appropriate instance of GeneralElementTypeFields (which, being final, will always require its current set of fields to be satisfied), he would specify a non-abstract child of SpecificElementTypeFields, containing data pertinent to the new output type.

For example, if the programmer wished for XML tags `<jFrame>...</jFrame>` to convert to a corresponding javax.swing.JFrame object, the programmer would do something along the following lines:

- Create a non-abstract descendant of SpecificElementTypeFields. Let this class be called SwingElementTypeFields.

- In SwingElementTypeFields, declare all fields necessary to build a Swing object. For example, one such field could be a string of Java code used to instantiate the associated Swing element. Assume this is declared as String instantiateCode.

- Create a new ElementType called JFrame. After the programmer satisfies ElementType’s first method signature of getGeneralElementTypeFields(), a new instance of SwingElementTypeFields must be created to satisfy getSpecificElementTypeFields(). Here, define instantiateCode as "JFrame jFrame = new JFrame();".

42
• Continue defining each of the fields in JFrame’s instance of SwingElementTypeFields, so that JFrame is fully equipped to generate Swing code. Additional fields could include an anonymous object which generates Swing code for adding elements to the JFrame; another anonymous object which generates code setting the visibility of the JFrame; and others as needed.

Similarly, interface Attribute does not actually require an instance of HtmlAttributeFields, but rather SpecificAttributeFields, an empty, abstract class which is the immediate parent of HtmlAttributeFields. In the example of Swing code generation, the programmer would define a non-abstract child class of SpecificAttributeFields—say, one called SwingAttributeFields—in which all fields needed for attribute generation are declared.

It must be noted that the system does not inherently “know” how to generate Swing code, HTML markup, or any other type of content. Instead, it must be explicitly taught to do so, such as how FormElement.toHtml() and CsunForm.toHtml() dictate exactly how to create HTML from Java objects. While the XML compiler will faithfully parse valid XML markup into XmlToJavaBase objects, how these objects are handled afterwards is entirely the responsibility of the programmer.
This project began with a clear requirement: from a minimally-formatted block of XML, create a nicely-designed HTML form, in a process which takes care of as many details as possible for the user. This requirement, however, in no way specified the design necessary to achieve it. Thus, the system design process began with exploring the “big picture”, through questions such as:

- What exactly should the user need to input in order to output the desired form?
- How much should the system take care of on the user’s behalf (or even forbid the user from doing)?
- How will the XML be parsed for correctness?
- Once the XML’s correctness is verified, how will it be converted to anything useful?
- Once the system has verified which elements will be placed on-screen in the form, how will the elements be neatly laid-out in a page?

Drawing on past experiences in programming complex systems, a decision was made to “start with the end in mind.” That is, the first phase of the project would set aside the question of XML parsing, and instead focus on programmatically creating HTML forms from Java objects. Once this was successfully accomplished, the focus would then shift to generating these objects from XML. The following sections recount this overall process, including challenges faced, how they were solved, and the thought process behind these solutions.

9.1 Programmatic Form Creation

Once the decision was made to tackle Java to HTML, new questions arose about how to this specific process would work. These included:

- What types of HTML elements would be laid out? (This was partially set out in the project proposal.)
- What sorts of attributes would be allowed, and for which element types?
- How would attributes be automatically generated by the system so that user doesn’t have to? Especially important was how to estimate the width and height of elements when displayed on-screen.
- It was always intended that elements be laid out from left to right, starting at the top-left corner. However, should the goal be to fill up every possible square inch of
page space by stacking elements; or, to layout them out “single-file”, even if doing so created empty space above elements?

### 9.1.1 Defining the Fundamentals

Because a form’s usefulness is defined in part by its range of available elements, the question of which HTML elements to support was a logical place to start. The elements set forth in the project proposal—namely, `cForm`, `textBox`, `comboBox`, `comboOption`, `checkBox`, `radio`, `date`, `label`, and `pic`—became the first members of enum `HtmlElementType`. This enum was later expanded to include additional elements, as will be seen below.

From there, the question turned to what attributes those elements would need. A few came immediately to mind:

- Width (in inches, for ease of use)
- Height (also in inches)
- Source (for pictures)
- Id (for HTML data to the server)
- Value (discussed in more detail below)
- Font attributes, including name, point size, boldness, italicization, and underlining

Accordingly, these became the first members of enum `HtmlAttribute`, with more added later as needed.

The question now turned to how to synthesize `HtmlElementType` and `HtmlAttribute` together. Looking at the big picture, it became apparent that an “HTML element” is the combination of exactly one `HtmlElementType`—no more, no less—and a mapping of attributes to their respective values. Put simply, an element has a specific type which defines its look and feel—for example, a radio button looks and feels quite different from a text box—and, it has attributes which define more about what the element represents, what it does, and what it looks like. From this synthesis, class `FormElement`, defined in Section 4.1, was born.

Out of all these attributes, the `HtmlAttribute.Value` attribute required the most care and consideration. The value of an XML or HTML element is typically defined as a string of text. For example, in the “First name” text box seen in the Human Resources form in Appendix B, the value of the text box is the string “Henry” which a user typed into it. For static content, such as labels, the system considers the value to be the string it displays to the user, such as “First name.”
However, it did not take much further thought to realize that a definition of value which only includes strings is incomplete. Consider the following XML markup:

```
<comboBox id="highestDegreeEarned">
  <comboOption>Bachelors</comboOption>
  <comboOption>Masters</comboOption>
  <comboOption>Doctorate</comboOption>
</comboBox>
```

Figure 9.1: XML markup for a combo box.

In this instance, the Value of the comboBox is clearly not a single string. Instead, it is a list containing three comboOption elements. Given this need to treat certain attribute contents differently, a wrapper class called AttributeContent was created:

```
AttributeContent
contentString : String
containedElements : ArrayList<XmlToJavaBase>

AttributeContent(String)
AttributeContent(ArrayList<XmlToJavaBase>)
getContainedString() : String
getContainedElements() : ArrayList<XmlToJavaBase>
```

Figure 9.2: UML representation of class AttributeContent.

Because of the structure of the two constructors, it is possible to imbue an instance of AttributeContent with a value that is a string, or with a value that is a list of elements, but not both. For consistency, every Attribute maps its value to an AttributeContent object, not just HtmlAttribute.Value.

9.1.2 Combining FormElements into CsunForms

Now that the fundamental building blocks of ElementType, Attribute, and AttributeContent had been defined, as well as their synthesis into the FormElement class, the next question was how to combine FormElement instances into a single form. From this, class CsunForm was created. Mirroring the structure of a paper form, CsunForm originally was simply a container for a list of FormElement objects, as well as fields for width, height, and margins.

However, it was soon realized that CsunForm had far more in common with FormElement than it had differences. A CsunForm, like a FormElement whose type is comboBox, is also an element whose value is a list of child FormElement objects. In this case, the list is comprised of all FormElement instances laid out in the form (except for those who are themselves children of other elements). Both classes also need to implement function fi-
nalizeAttributes(), which generates AttributeContent objects for attributes the user has not defined.

Drawing on good programming practices, the fields and functions common to both classes were moved to a newly created abstract class, HtmlElementBase. This class declares functions finalizeAttributes() and toHtml() to be abstract, forcing the programmer to tailor them to the implementing class’ needs.

9.1.3 Form Generation

Once CsunForm and FormElement were defined in theory, then wrought in code, it was time to use them to generate actual forms. In the early stages, these often consisted of just five to eight FormElement instances. Each of these was hand-constructed in Java code, with each attribute being explicitly specified by the programmer (automatic attribute generation had not yet been implemented). An issue that had not been previously considered quickly presented itself—there was no spacing between elements! Thus, the space element was born, which allows the user to specify that an area of defined width and height be left empty. Better yet, if a user did not specify a space element between two elements, CsunForm.toHtml() was modified to automatically place a small space in between them, as well as between adjacent rows.

After forms with small collections of elements were successfully converted from Java to HTML, it was time to see if the system could create a more complete form which would be useful to a CSUN department. To be as authentic as possible, an actual form hosted by CSUN Human Resources at www-admn.csun.edu/ohrs/compemp/forms/newempl.htm was replicated. The XML markup and picture of the resulting form, as well as a picture of the original form, can be seen in Appendix A. Creating this form led to the addition of the following elements and attributes which had previously not been considered:

- **Elements**
  - Spacing, expressed as `<space/>`.
  - Linebreaks, expressed as `<linebreak/>`.
  - Pagebreaks, expressed as `<pagebreak/>`. These are not used in the human resources form, but came as a natural extension to `<linebreak/>`.
  - Stylized horizontal lines: expressed as `<hLine/>` in the system’s XML, and resulting in `<hr/>` in HTML.

- **Attributes**
  - Form margins, expressed separately as `marginTop`, `marginRight`, `marginBottom`, and `marginLeft`, all measured in inches.
  - Text alignment options (left, right, center, or justify).
  - Element color.
  - Element background color.
An alternative label for pictures which fail to load (known both in the system and in HTML as alt).

While the Human Resources form does not contain every ElementType supported by the system—types including comboBox, comboOption, checkBox, and radio are absent—between this form and other informal prototypes, every element type had been successfully converted from Java to HTML. It was now time to implement the other piece of the puzzle—XML to Java.

9.2 Developing an XML Compiler

Like the HTML generation process, the XML compilation process presented unique challenges. The requirement of “XML goes in, Java objects come out” was quite clear, but how this would happen was entirely open-ended. One potential option entailed avoiding a customized XML compiler altogether, and instead attempting to use an existing open-source solution such as XStream. XStream is library capable of transferring the contents of valid XML markup into a pre-defined Java class, and vice-versa [23].

After consideration, however, it was decided to avoid XStream or other external libraries, given that shoe-horning one into the idiosyncrasies of this project would have resulted in far more headaches, and much less flexibility, than creating a customized solution. This turned out to be the correct assessment. For example, whereas a great deal of hackneyed transition code would have been required to correctly form AttributeContent objects using an outside library, the system’s custom XML compiler seamlessly creates these upon attribute data validation.

Additionally, reference materials explicitly detailing how to build an XML compiler were not sought out. Using such materials would have negated the learning opportunity inherent in developing a customized solution. Instead, general computational theory and compiler design concepts were employed.

9.2.1 Lexical and Syntax Analysis

As learned in Comp 333: Concepts of Programming Languages, the parsing of most languages begins with breaking the input into tokens [24]. An advantageous quality of XML is that it is relatively easy to distinguish tokens from one another. This is due to the relatively small set of user-defined token types—XmlStartingTag, XmlEndingTag, XmlEmptyTag, TextValue, and Comment—and the unique structure of each.

As is typical for parsing tokens from input, a deterministic finite automaton (DFA) was programmed, and is detailed in Section 5.1.1. Defining the DFA’s states and transitions actually proved to be simpler than expected. Using pencil and paper, the programmer starts with an empty string of characters at State 1, and considers the implications of all potential first characters to be parsed. A < character indicates that a new starting, ending, or empty token has been instantiated; a whitespace character indicates that a new Whitespace token has been instantiated; and any other character indicates the beginning of a new TextValue token. Similarly, each other state has its own set of characters which, upon being seen in
that state, signifies a specific token structure and state transition.

The first major challenge with the XML compiler arose with this question: how should the compiler react to a character that does not fit any of the edges traveling out of the current state? Of course, an error message should be logged and the compilation process should be halted before trying to generate a form, but what should happen to the lexical analysis currently in progress? Should it immediately abort, or should it try to continue despite the fact that the XML is fatally flawed in some fashion?

Eventually, a philosophy of “simple is best” was adopted. When an edge to match the current character cannot be found, the error is logged (guaranteeing that no HTML form will be generated), and the state machine returns to State 1. This presents a “blank slate” for future characters parsed by this DFA instance, which could very well form valid XML tokens.

The same basic approach was also taken with designing the syntax analyzer. Because XML syntax constitutes a grammar that is context-free, but not regular (see Section 5.2.2 for details), a pushdown automaton (PDA) was required. Like the lexical DFA, the PDA can be designed by beginning with an empty stack at State 1, and considering the implications of the four token types parsed in syntax analysis: XmlStartingTag, TextValue, XmlEndingTag, and XmlEmptyTag. (Per Section 5.1.2, tokens with any other TokenType are filtered out before syntax analysis.) From there, the programmer constructs the rest of the PDA based on the simple rules of XML syntax. For the same reasons as the lexical DFA, it was decided that the PDA would also return to State 1 upon seeing any unexpected tokens, logging an error along the way.

9.2.2 XML Tokens to Java Objects

After the lexical and syntax analysis parsers were implemented, the next question was clear: how would the XML tokens be transformed into the CsunForm/FormElement classes which had been defined at the start of the project? This transformation would entail looking at each distinct XML element— whether an empty element such as `<button />`, or a non-empty element such as `<label>CSUN Computer Science</label>`— and transfer both the ElementType and map of attributes into the HtmlElementBase. (At this time, XmlToJavaBase had not yet been created; instead, the compiler was programmed to generate CsunForm and FormElement objects directly).

As before, “simple is best” proved to be the correct philosophy. A simple recursive-descent function, seen in Figure 5.15 as tokensToElements(), was implemented. When an XmlEmptyTag token is found, its contents are moved directly into a new FormElement object. When an XmlStartingTag is found, all tokens “sandwiched” between it and its matching XmlEndingTag are recursively transferred into an AttributeContent object, which is then inserted into the parent XmlStartingTag.

A problem presented by this, however, is that the corresponding XmlEndingTag cannot be definitively identified with a simple left-to-right scan. However, the solution soon became clear— the corresponding ending tag is clearly identified during syntax analysis,
by virtue of it being the ending tag causing the starting tag to be popped from the stack. Accordingly, the syntax analysis process was modified, so that a reference to the XmlEndingTag is stored in the XmlStartingTag just after it is removed from the stack. This solved the issue in a clean and intuitive manner.

9.2.3 Separating Compiler from Forms

As mentioned above, the compiler was initially programmed to directly generate CsunForm and FormElement objects. However, based on feedback from committee members, the compiler was modified so as to be agnostic of these specific objects. Under this new scheme, the compiler extracts ElementType and Attribute data common to any well-formed XML markup—not just form-system XML—and place this data into generic XmlToJavaBase objects. These objects are then be handed off to another package with a specific purpose, such as the edu.csun.forms_site_generator package, which performs the actual HTML generation.

A key question this raised, however, was how to also treat these generated XmlToJavaBase objects as the non-abstract CsunForm and FormElement classes which had already been defined, so as to retain these classes’ functionality. The solution soon became clear with the realization that both CsunForm and FormElement are themselves descendants of XmlToJavaBase, since they both have a single element type, and a mapping of attributes to values. By making HtmlElementBase, the parent class of CsunForm and FormElement, itself a child class of XmlToJavaBase, the compiler was now able to indirectly generate CsunForm and FormElement objects through the use of interface XmlToJavaInstantiator.
Chapter 10

Conclusion

10.1 Future Expansion

In addition to implementing the full system envisioned in Chapter 2, the following are ideas for expanding the system in the future:

- **Form HTML Layout:**
  - Automatically add “springs” to GUI elements in generated HTML markup, so that adjacent elements push each other apart to fill up empty space.

- **Compiler Feedback:**
  - After a block of XML successfully compiles, color-code the characters in the user’s XML markup based on whether they are part of an XML tag, attribute data, or part of a TextValue token.
  - Similarly, if a block of XML fails to compile, the offending locations could be underlined in red.

- **User Controls:**
  - Allow for users to upload pictures before generating a preview form, so they can see a form closer to the end product.
  - Allow users to set the width, height, and other attributes of the cForm via GUI controls on the webpage. This could replace the `<cForm>` and `</cForm>` tokens in the user’s markup, or, these controls could manipulate the `</cForm>` token’s markup via JavaScript.

- **Other:**
  - Function `WebFunctions.logThrowable()` should be changed to output to a `java.util.logging.Logger` instance, instead of a plaintext file.

10.2 Summary

This project is the first of a multi-stage effort to bring a comprehensive electronic form system to CSUN. It accomplishes the goals formulated and approved in the project proposal: an XML schema for form creation, and a system for generating HTML forms from markup conforming to this schema.

While developers working on the system in the future will no doubt find room for improving the code submitted here, overall they should find it to be a very solid foundation.
Thanks to the orthogonality provided by clearly and narrowly defined interfaces, such as ElementType and Attribute, programmers can expand the system’s functionality without fear of breaking existing code. It is also hoped that the XML to Java compiler itself will prove useful beyond HTML form creation. The compiler’s ability to validate and parse XML into XmlToJavaBase objects, while remaining agnostic towards the descendant class of those objects, provides a powerful tool for XML usage.
Bibliography


Appendix A

Sample XML Markup and Resulting Forms

A.1 Form One: Bacon Ipsum

```
<cfForm width="8.5" height="11" marginTop=".75" marginRight=".75"
marginBottom=".75" marginLeft=".75">
  <label height="1.4">
    Bacon ipsum dolor sit amet ribeye salami pork belly, jowl cow
    jerky capicola pork loin tenderloin pig. Jerky spare ribs meatball
    meatloaf biltong, hamburger brisket ground round jowl andouille
    cow shankle ball tip beef ribs. Ball tip flank pork belly spare ribs
    t–bone brisket shank, chicken chuck strip steak. Tongue filet mignon
    turkey boudin swine. Turducken rump pig biltong fatback.
    Prosciutto tail pork chop, tenderloin corned beef t–bone rump chuck
    shankle hamburger pork loin short ribs kielbasa sausage.
  </label>

<cfForm width="8.5" height="11" marginTop=".75" marginRight=".75"
marginBottom=".75" marginLeft=".75">
  <label height=".8">
    Swine pork chop meatball, tri–tip ham hock pork belly short loin
    t–bone boudin andouille flank leberkas venison shankle. Turkey rump
    doner pork beef ribs pork loin. Ribeye chicken chuck ground round
    beef ribs. Boudin tenderloin prosciutto, sirloin sausage turkey short ribs.
  </label>

<cfForm width="8.5" height="11" marginTop=".75" marginRight=".75"
marginBottom=".75" marginLeft=".75">
  <label height="1.1">
    Ham beef pastrami, boudin doner strip steak comred beef ground round
    leberkas meatball shank. Ball tip turkey meatball, spare ribs drumstick
    flank sirloin capicola meatloaf ham pastrami boudin sausage doner
    pancetta. Kielbasa bacon meatloaf, ribeye sausage jerky jowl sirloin tail
    cow pork pork belly hamburger. Spare ribs pork loin tail salami, short
    ribs filet mignon bresaola pork belly meatball doner tongue.
  </label>

<cfForm width="8.5" height="11" marginTop=".75" marginRight=".75"
marginBottom=".75" marginLeft=".75">
  <label height="1.1">
    Biltong kielbasa pig ground round brisket jowl pancetta drumstick
    pork doner pork chop. Pork chop tri–tip jowl, tail short loin doner
    andouille sirloin rump flank filet mignon. Bacon pork chop sausage
    pork belly, fatback beef ribs sirloin ribeye tri–tip drumstick ham
    turkey. Ground round capicola leberkas, strip steak brisket turducken
    meatloaf frankfurter shoulder t–bone tri–tip.
  </label>
```

Figure A.1: XML markup for Bacon Ipsum text form (part 1/2).
Figure A.2: XML markup for Bacon Ipsum text form (part 2/2).

Figure A.3: Resulting Bacon Ipsum form (same as Figure 6.4).
A.2  Form Two: Human Resources Sign-In form

The following markup, broken up across five code listings, should be treated as a continuous block of XML. This form is a partial reconstruction of the form at http://www-admn.csun.edu/ohrs/compemp/forms/newempl.htm.

```
1  <cForm width='11.0' height='8.5' marginTop='.5'
2     marginRight='.5' marginBottom='.5' marginLeft='.5'
3     fontBold='bold' fontItalic='normal' fontUL='none'
4     fontPoint='9' fontName='Arial'>
5     <pic source='CSUN_logo.png' alt='CSUN Logo' width='2.25'
6        height='.75'/>
7     <space width='1.0' />
8     <label width='4.0' textAlign='center' fontBold='normal'
9        fontItalic='normal' fontUL='none' fontPoint='14'>
10        Office of Human Resources
11        &lt;b&gt;New Employee Sign−In Form&lt;/b&gt;
12        UN 165 Mon to Fri − 8:30 am to 4:30pm
13     </label>
14     <!−− This is a comment! −−>
15     <space width='1.0' />
16     <label width='1.7' height='.5' fontPoint='8'>
17        OFFICE OF HUMAN RESOURCES
18        TEL NO: (818) 677−2101
19        FAX NO: (818) 677−7863
20        MAIL DROP: 8229
21     </label>
22     <space width='0' height='0' />
23     <linebreak/>
24     <space width='10.0' height='.05' />
25     <hLine/>
26     <linebreak/>
27     <label width='10.0' textAlign='center' fontBold='bold'
28        fontPoint='11'>
29        Please complete and print this form, and send with
30        the new employee to the
31     </label>
32     <label width='10.0' textAlign='center' fontBold='bold'
33        fontPoint='11'>
34        Office of Human Resources, University Hall 165.
35     </label>
36     <linebreak/>
```

Figure A.4: XML markup for Human Resources Sign-In form (part 1/5).
Employees will not receive a Pay Warrant until the SIGN−IN PROCESS has been completed.

Figure A.5: XML markup for Human Resources Sign-In form (part 2/5).
Figure A.6: XML markup for Human Resources Sign-In form (part 3/5).
<radio name="payrollType" value="USU"/>
<space width=".05"/>
<label width="2.5" fontBold="normal">USU</label>
<label width="2.5" fontBold="normal">USU</label>
<linebreak/>
<radio name="appointmentType" value="partTimeFaculty"/>
<space width=".05"/>
<label width="1.5" fontBold="normal">Part-Time Faculty</label>
<radio name="appointmentType" value="studentAssistant"/>
<space width=".05"/>
<label width="3.35" fontBold="normal">Student Assistant</label>
<linebreak/>
<radio name="appointmentType" value="teachingAssociate"/>
<space width=".05"/>
<label width="1.5" fontBold="normal">Teaching Associate</label>
<radio name="appointmentType" value="instructionalStudentAsstnt"/>
<space width=".05"/>
<label width="3.35" fontBold="normal">Instructional Student Assistant</label>
<linebreak/>
<hLine/>
<space width="10" height="0"/>
<label width="10" fontPoint="14" fontBold="bold" textAlign="center">*
Students/Employees of THE UNIVERSITY CORPORATION
must have the following section completed and signed
before appointment begins:
</label>
<hLine/>
<space width="10" height="0"/>
<label width="10" fontPoint="14" fontBold="bold" textAlign="center">ACCOUNT
FUND
DEPT ID
PROGRAM
</label>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
<space width=".75"/>
Figure A.8: XML markup for Human Resources Sign-In form (part 5/5).
Figure A.9: Resulting Human Resources Sign-In form.

Please complete and print this form, and send with the new employee to the Office of Human Resources, University Hall 165. Employees will not receive a Pay Warrant until the SIGN-IN PROCESS has been completed.

<table>
<thead>
<tr>
<th>CSUN ID (State Only)</th>
<th>LAST NAME:</th>
<th>First Name:</th>
<th>RIJ:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATE OF EMPLOYMENT</th>
<th>SUPERVISOR NAME:</th>
<th>EMAIL:</th>
<th>EXTENSION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>04/26/2013</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT ID (State Only):</th>
<th>DEPARTMENT:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appointment Type: (Check One)</th>
<th>Payroll Type: (Check One)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Staff</td>
<td>University Corp *</td>
</tr>
<tr>
<td>Full-Time Faculty</td>
<td>Assoc. Students</td>
</tr>
<tr>
<td>Part-Time Faculty</td>
<td>State **</td>
</tr>
<tr>
<td>Teaching Associate</td>
<td>USU</td>
</tr>
<tr>
<td>Graduate Assistant</td>
<td></td>
</tr>
<tr>
<td>Work-Study Student</td>
<td></td>
</tr>
<tr>
<td>Student Assistant</td>
<td></td>
</tr>
<tr>
<td>Instructional Student Assistant</td>
<td></td>
</tr>
</tbody>
</table>

* Students/Employees of THE UNIVERSITY CORPORATION must have the following section completed and signed before appointment begins:

<table>
<thead>
<tr>
<th>ACCOUNT</th>
<th>FUND</th>
<th>DEPT ID</th>
<th>PROGRAM</th>
<th>CLASS</th>
<th>PROJECT</th>
<th>PAY RATE $</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TUC Sponsored Program Approval ONLY

Signature: _______________________________ Date: _______________
Figure A.10: Original Sign-In form, as found on CSUN website.

California State University Northridge
Office of Human Resources
New Employee Sign-In Form
UN 165 Mon to Fri - 8:30 am to 4:30 pm

Please complete and print this form, and send with the new employee to the Office of Human Resources, University Hall 165.
Employees will not receive a Pay Warrant until the SIGN-IN PROCESS has been completed.

<table>
<thead>
<tr>
<th>CSUN ID: (State Only)</th>
<th>LAST NAME:</th>
<th>FIRST NAME:</th>
<th>M.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATE OF EMPLOYMENT:</th>
<th>SUPERVISOR NAME:</th>
<th>EMAIL:</th>
<th>EXTENSION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm/dd/yy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT ID: (State Only)</th>
<th>DEPARTMENT:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Appointment Type: (Check One)**
- [ ] Staff
- [ ] Full-Time Faculty
- [ ] Part-Time Faculty
- [ ] Teaching Associate
- [ ] Graduate Assistant
- [ ] Work-Study Student
- [ ] Student Assistant
- [ ] Instructional Student Assistant
- [ ] University Corp
- [ ] Assoc. Students
- [ ] State
- [ ] USU

**Payroll Type: (Check One)**
- [ ] University Corp
- [ ] Assoc. Students
- [ ] State
- [ ] USU

† Students/Employees of THE UNIVERSITY CORPORATION must have the following section completed and signed before appointment begins:

<table>
<thead>
<tr>
<th>ACCOUNT</th>
<th>FUND</th>
<th>DEPT ID</th>
<th>PROGRAM</th>
<th>CLASS</th>
<th>PROJECT</th>
<th>PAY RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TUC Sponsored Program Approval ONLY**

Signature: ___________________________ Date: ________________
Appendix B

XML Schema

B.1 Formal Schema

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- Attributes -->

<xs:simpleType name="alt">
    <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="bgColor">
    <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="checked">
    <xs:restriction base="xs:string">
        <xs:pattern value="|checked"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="color">
    <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="fontBold">
    <xs:restriction base="xs:string">
        <xs:pattern value="normal|bold|bolder|
                  lighter|inherit"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="fontItalic">
    <xs:restriction base="xs:string">
        <xs:pattern value="normal|italic|
                  oblique|inherit"/>
    </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Figure B.1: Formal XML schema (part 1/8).
Figure B.2: Formal XML schema (part 2/8).
Figure B.3: Formal XML schema (part 3/8).
<xs:complexType name="checkBox">
  <xs:attribute name="id" type="id" />
  <xs:attribute name="name" type="name" />
  <xs:attribute name="value" type="value" />
  <xs:attribute name="width" type="width" />
  <xs:attribute name="height" type="height" />
  <xs:attribute name="checked" type="checked" />
</xs:complexType>

<xs:complexType name="comboOption">
  <xs:simpleContent>
    <xs:extension base="nonWhitespaceString">
      <xs:attribute name="id" type="id" />
      <xs:attribute name="fontName" type="fontName" />
      <xs:attribute name="fontPoint" type="fontPoint" />
      <xs:attribute name="fontBold" type="fontBold" />
      <xs:attribute name="fontItalic" type="fontItalic" />
      <xs:attribute name="fontUL" type="fontUL" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="comboBox">
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:element name="comboOption" type="comboOption" minOccurs="1" />
  </xs:choice>
  <xs:attribute name="id" type="id" />
  <xs:attribute name="name" type="name" />
  <xs:attribute name="width" type="width" />
  <xs:attribute name="height" type="height" />
</xs:complexType>

Figure B.4: Formal XML schema (part 4/8).
<xs:complexType name="date">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="id" type="id" />
      <xs:attribute name="name" type="name" />
      <xs:attribute name="width" type="width" />
      <xs:attribute name="height" type="height" />
      <xs:attribute name="fontName" type="fontName" />
      <xs:attribute name="fontPt" type="fontPoint" />
      <xs:attribute name="fontBold" type="fontBold" />
      <xs:attribute name="fontIl" type="fontItalic" />
      <xs:attribute name="fontUL" type="fontUL" />
      <xs:attribute name="maxChars" type="maxChars" />
      <xs:attribute name="textAlign" type="textAlign" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="hLine">
  <xs:attribute name="id" type="id" />
  <xs:attribute name="width" type="width" />
  <xs:attribute name="height" type="height" />
  <xs:attribute name="color" type="color" />
  <xs:attribute name="bgColor" type="bgColor" />
</xs:complexType>

<xs:complexType name="label">
  <xs:simpleContent>
    <xs:extension base="nonWhitespaceString">
      <xs:attribute name="id" type="id" />
      <xs:attribute name="width" type="width" />
      <xs:attribute name="height" type="height" />
      <xs:attribute name="fontName" type="fontName" />
      <xs:attribute name="fontPt" type="fontPoint" />
      <xs:attribute name="fontBold" type="fontBold" />
      <xs:attribute name="fontIl" type="fontItalic" />
      <xs:attribute name="fontUL" type="fontUL" />
      <xs:attribute name="color" type="color" />
      <xs:attribute name="bgColor" type="bgColor" />
      <xs:attribute name="textAlign" type="textAlign" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

Figure B.5: Formal XML schema (part 5/8).
<xs:complexType name="linebreak">
  <xs:attribute name="id" type="id" />
</xs:complexType>

<xs:complexType name="link">
  <xs:simpleContent>
    <xs:extension base="nonWhitespaceString">
      <xs:attribute name="id" type="id" />
      <xs:attribute name="linkTo" type="linkTo" use="required" />
      <xs:attribute name="width" type="width" />
      <xs:attribute name="height" type="height" />
      <xs:attribute name="fontName" type="fontName" />
      <xs:attribute name="fontPoint" type="fontPoint" />
      <xs:attribute name="fontBold" type="fontBold" />
      <xs:attribute name="fontItalic" type="fontItalic" />
      <xs:attribute name="fontUL" type="fontUL" />
      <xs:attribute name="color" type="color" />
      <xs:attribute name="bgColor" type="bgColor" />
      <xs:attribute name="textAlign" type="textAlign" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="pagebreak">
</xs:complexType>

<xs:complexType name="pic">
  <xs:attribute name="id" type="id" />
  <xs:attribute name="source" type="source" use="required" />
  <xs:attribute name="alt" type="alt" use="required" />
  <xs:attribute name="width" type="width" use="required" />
  <xs:attribute name="height" type="height" use="required" />
</xs:complexType>

Figure B.6: Formal XML schema (part 6/8).
<xs:complexType name="radio">
  <xs:attribute name="id" type="id" />
  <xs:attribute name="name" type="name"
    use="required" />
  <xs:attribute name="value" type="value" />
  <xs:attribute name="width" type="width" />
  <xs:attribute name="height" type="height" />
  <xs:attribute name="checked" type="checked" />
</xs:complexType>

<xs:complexType name="space">
  <xs:attribute name="id" type="id" />
  <xs:attribute name="width" type="width" />
  <xs:attribute name="height" type="height" />
</xs:complexType>

<xs:complexType name="textBox">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="id" type="id" />
      <xs:attribute name="name" type="name" />
      <xs:attribute name="width" type="width" />
      <xs:attribute name="height" type="height" />
      <xs:attribute name="fontName" type="fontName" />
      <xs:attribute name="fontPoint" type="fontPoint" />
      <xs:attribute name="fontBold" type="fontBold" />
      <xs:attribute name="fontItalic" type="fontItalic" />
      <xs:attribute name="fontUL" type="fontUL" />
      <xs:attribute name="maxChars" type="maxChars" />
      <xs:attribute name="textAlign" type="textAlign" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="cForm">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="button" type="button"
      minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="checkBox" type="checkBox"
      minOccurs="0" maxOccurs="unbounded" />
  </xs:choice>
</xs:complexType>

Figure B.7: Formal XML schema (part 7/8).
<xs:element name="comboBox" type="comboBox"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="date" type="date"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="hLine" type="hLine"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="label" type="label"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="linebreak" type="linebreak"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="link" type="link"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="pagebreak" type="pagebreak"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="pic" type="pic"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="radio" type="radio"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="space" type="space"
    minOccurs="0" maxOccurs="unbounded" />
<xs:element name="textBox" type="textBox"
    minOccurs="0" maxOccurs="unbounded" />
</xs:choice>
<xs:attribute name="id" type="id" />
<xs:attribute name="width" type="width" />
<xs:attribute name="height" type="height" />
<xs:attribute name="marginTop" type="marginTop" />
<xs:attribute name="marginRight" type="marginRight" />
<xs:attribute name="marginBtm" type="marginBottom" />
<xs:attribute name="marginLeft" type="marginLeft" />
<xs:attribute name="fontName" type="fontName" />
<xs:attribute name="fontPoint" type="fontPoint" />
<xs:attribute name="fontBold" type="fontBold" />
<xs:attribute name="fontItalic" type="fontItalic" />
<xs:attribute name="fontUL" type="fontUL" />
<xs:attribute name="color" type="color" />
</xs:complexType>
</xs:schema>

Figure B.8: Formal XML schema (part 8/8).
B.2 Informal XML Structure

The listings below are an “informal schema” that would be more accessible for typical form designers. The tags `<cForm [attributes]>` and `</cForm>` MUST be the outermost tags. All elements, except for `<cForm>` and `<comboOption>`, must be nested children of the `<cForm>`. Element `<comboOption>` must nested under `<comboBox>`. Attributes which users must always input for an element are designated above each element.

```xml
<!-- Required: none -->
<cForm id="string" width="decimal" height="decimal"
    marginTop="decimal" marginRight="decimal"
    marginBottom="decimal" marginLeft="decimal"
    fontName="string" fontPoint="integer"
    fontBold="string" fontItalic="string"
    fontUL="string" color="string">

<!-- Required: none -->
<button id="string" name="string" value="string"
    width="decimal" height="decimal"
    fontName="string" fontPoint="integer"
    fontBold="string" fontItalic="string"
    fontUL="string" />

<!-- Required: none -->
<checkBox id="string" name="string" value="string"
    width="decimal" height="decimal"
    checked="string" />

<!-- Required: child comboOption(s) -->
<comboBox id="string" width="decimal"
    height="decimal" name="string">

<!-- Required: text value -->
<comboOption id="string" fontName="string"
    fontPoint="integer"
    fontBold="string"
    fontItalic="string"
    fontUL="string">
    Non-empty text must go here
</comboOption>
</comboBox>

<!-- Required: none -->
<date id="string" width="decimal" name="string"
    height="decimal" fontName="string"
    fontPoint="integer" fontBold="string"
```

Figure B.9: Informal XML structure (part 1/3).
Figure B.10: Informal XML structure (part 2/3).
<!-- Required: none -->
<pagebreak />
<!-- Required: source, alt, width, height -->
<pic id="string" source="string" alt="string"
    width="decimal" height="decimal" />
<!-- Required: name -->
<radio id="string" name="string" value="string"
    width="decimal" height="decimal"
    checked="string" />
<!-- Required: none -->
<space id="string" width="decimal"
      height="decimal" />
<!-- Required: none -->
<textBox id="string" name="string" width="decimal"
        height="decimal" fontName="string"
        fontPoint="integer" fontBold="string"
        fontItalic="string" fontUL="string"
        maxChars="integer" textAlign="string">
  Starting text goes here, user can then modify
</textBox>
</cForm>

Figure B.11: Informal XML structure (part 3/3).
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Allowed Values</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt</td>
<td>Any string of characters</td>
<td>Alternative label if picture does not load</td>
</tr>
<tr>
<td>bgColor</td>
<td>Any string of characters (but should be an HTML-recognized color identifier, such as “grey” or “#E7E7E7”)</td>
<td>Background color of element</td>
</tr>
<tr>
<td>checked</td>
<td>Either an empty string, or string “checked”</td>
<td>Whether to check the given checkbox or radio button</td>
</tr>
<tr>
<td>color</td>
<td>Any string of characters (but should be an HTML-recognized color identifier, such as “grey” or “#E7E7E7”)</td>
<td>Element color</td>
</tr>
<tr>
<td>fontBold</td>
<td>normal, bold, bolder, lighter, inherit</td>
<td>Font boldness</td>
</tr>
<tr>
<td>fontItalic</td>
<td>normal, italic, oblique, inherit</td>
<td>Font styling</td>
</tr>
<tr>
<td>fontName</td>
<td>Arial, Comic Sans MS, Courier New, Georgia, Helvetica, Impact, Lucida Console, Lucida Sans Unicode, Palatino Linotype, Tahoma, Times New Roman, Trebuchet MS, Verdana</td>
<td>Font name</td>
</tr>
<tr>
<td>fontPoint</td>
<td>Any floating point base-ten number</td>
<td>Font point size</td>
</tr>
<tr>
<td>fontUL</td>
<td>none, underline, overline, inherit, line-through</td>
<td>Font underlining</td>
</tr>
<tr>
<td>height</td>
<td>Any floating point base-ten number</td>
<td>Element height in inches</td>
</tr>
<tr>
<td>id</td>
<td>Any string of characters</td>
<td>DOM Id of element</td>
</tr>
<tr>
<td>linkTo</td>
<td>Any string of characters</td>
<td>URL of page to link to</td>
</tr>
<tr>
<td>marginTop</td>
<td>Any floating point base-ten number</td>
<td>Top margin in inches of the element</td>
</tr>
<tr>
<td>marginRight</td>
<td>Any floating point base-ten number</td>
<td>Right margin in inches of the element</td>
</tr>
<tr>
<td>marginBottom</td>
<td>Any floating point base-ten number</td>
<td>Bottom margin in inches of the element</td>
</tr>
<tr>
<td>marginLeft</td>
<td>Any floating point base-ten number</td>
<td>Left margin in inches of the element</td>
</tr>
<tr>
<td>maxChars</td>
<td>Any positive integer base-ten number</td>
<td>Maximum characters that can be inputted to element</td>
</tr>
<tr>
<td>name</td>
<td>Any string of characters</td>
<td>Name of element in DOM</td>
</tr>
<tr>
<td>source</td>
<td>Any string of characters</td>
<td>Path to picture source</td>
</tr>
<tr>
<td>textAlign</td>
<td>left, right, center, justify, inherit</td>
<td>Text alignment</td>
</tr>
<tr>
<td>value</td>
<td>Any string of characters</td>
<td>Value of the element</td>
</tr>
<tr>
<td>width</td>
<td>Any floating point base-ten number</td>
<td>Element width in inches</td>
</tr>
</tbody>
</table>
Appendix C
Class UML Diagrams

C.1 Package edu.csun.forms.site.compiler

AttributeFunctions.java

<<Java Class>>
AttributeFunctions
edu.csun.forms.site.compiler

AttributeFunctions()
getAllowedAttributes(ElementType) : ArrayList<Attribute>
isValidAttributeFor(ElementType, Attribute) : boolean
isAttributeRequired(ElementType, Attribute) : boolean
getGeneralAttributes(ElementType, Attribute)
getSpecificAttributes(ArrayList<String>, Attribute)

<<Java Class>>
AttributeContent
edu.csun.forms.site.compiler

content(String)
getAttributeValue(java.util.List)</java.util.List, java.lang.String>
getAttributeValue(String)
getAttributeValue(ArrayList<XmloJo, JavaBase>)

<<Java Class>>
SpecificAttributeFields
edu.csun.forms.site.compiler

getAttributeValue(String)
getAttributeValue(ArrayList<XmloJo, JavaBase>)

<<Java Class>>
AttributeWrapper
edu.csun.forms.site.compiler

isMustSpecify(): boolean
getAttributeValue(Boolean)

<<Java Class>>
GeneralAttributeFields
edu.csun.forms.site.compiler

getAttributeValue(String)
getAttributeValue(DataValidator)
getAttributeValue(ArrayList<XmloJo, JavaBase>)

<<Java Interface>>
Attribute
edu.csun.forms.site.compiler

getAttributeValue(ArrayList<XmloJo, JavaBase>)
getAttributeValue(DataValidator)
getAttributeValue(ArrayList<XmloJo, JavaBase>)
getAttributeValue(ArrayList<XmloJo, JavaBase>)

ElementTypeFunctions.java
SyntaxAnalyzer.java
C.2 Package edu.csun.forms_site.generator

CsunForm.java
FormElement.java

```
<<Java Class>>

FormElement
du.oeun.forms_xmls_generator

- numLines: int
- FormElement(ElementType)
- getNumLines(): int
- setNumLines(int): void
- finalizeAttributes(Constraint): void
- toString(int, boolean): String
- getIndent(int): String
```

FormGenerator.java

```
<<Java Class>>

FormGenerator
du.oeun.forms_pitas_generator

- ERROR_LOG_PATH: String
- HTML_TEMPLATE_PATH: String
- loadSuccessful: boolean
- POSTIVE_INT_VALIDATOR: DataValidator
- DOUBLE_VALIDATOR: DataValidator
- VARCHAR_VALIDATOR: DataValidator

- FormGenerator()
- main(String[]): void
- getXmlFromFile(String): String
- writeHtmlFile(String, String): void
- xmlToHtml(String, boolean): String
- initializeApplication(): void
- wasLoadSuccessful(): boolean
- arrayContains(E[]): boolean
```
Appendix D

Source Code / Unit Tests / Javadoc / UML

These will be included with the attachment submitted with the project.

- **Source Code/Unit Tests:**
  - It is recommended that the source code in folder `/forms_site_project` be opened in an IDE such as Eclipse or Netbeans.
  - Source code is in packages `edu.csun.forms_site.compiler`, `edu.csun.forms_site.generator`, and `edu.csun.forms_site.web`.
  - Unit tests are in `edu.csun.forms_site.unit_tests`. All currently pass when executed.

- **Javadoc:** in folder `/doc`, open `index.html` in a web browser.

- **UML:** picture files for the UML diagrams seen above are in folder `/uml`. 