

WEBMATE

CAS 703 – Software Design Project

Prof. Richard Paige

Alap Dhruva

Deesha Patel

Azam (Mina) Mahdipour

Contents

1. Introduction	3
2. ECore Meta-Model	4
2.1 Assumption	4
2.2 Alternative Design	5
3. Concrete Syntax and Editor	6
4. Validation	8
5. Model to Text Transformation	10
5.1 Examples	10
5.2 Challenges	11

1. Introduction

WebMate is a web-development DSL¹ for automating the conversion of shorthand HTML abbreviations to structured and meaningful HTML code. With WebMate, a developer can type an expression in the form of an abbreviation and our DSL will convert that expression into a code fragment in HTML. Our editor makes it easy for a developer to quickly create and maintain complex web pages.

Using symbols shown in table 1-1, we can form an abbreviation string which can be converted into meaningful HTML code chunk.

Table 1-1: Symbols in WebMate

Symbol	Meaning	Example
+	Add two parallel HTML tags	div + h1
*	Duplicate the same tag multiple times	Input * 2
>	Add inner HTML tag	ul > li
{ }	Input string between start and end tag	h1 {exampleText}
[]	Attribute name and value of a tag	a [href=src]
#	Specify the id of the tag	class#container
.	Specify the class name	class.city
()	Group the tags	(div.city#cityNames)

The full code for the WebMate can be accessed through [github](#).

¹ Domain Specific Language

2. Ecore Meta-Model

As described in the introduction section, WebMate accepts the abbreviation strings from the user and generates the expansion format of the HTML code according to the meta-model we defined in the project. We went through many changes along the way to define our final version of the meta-model.

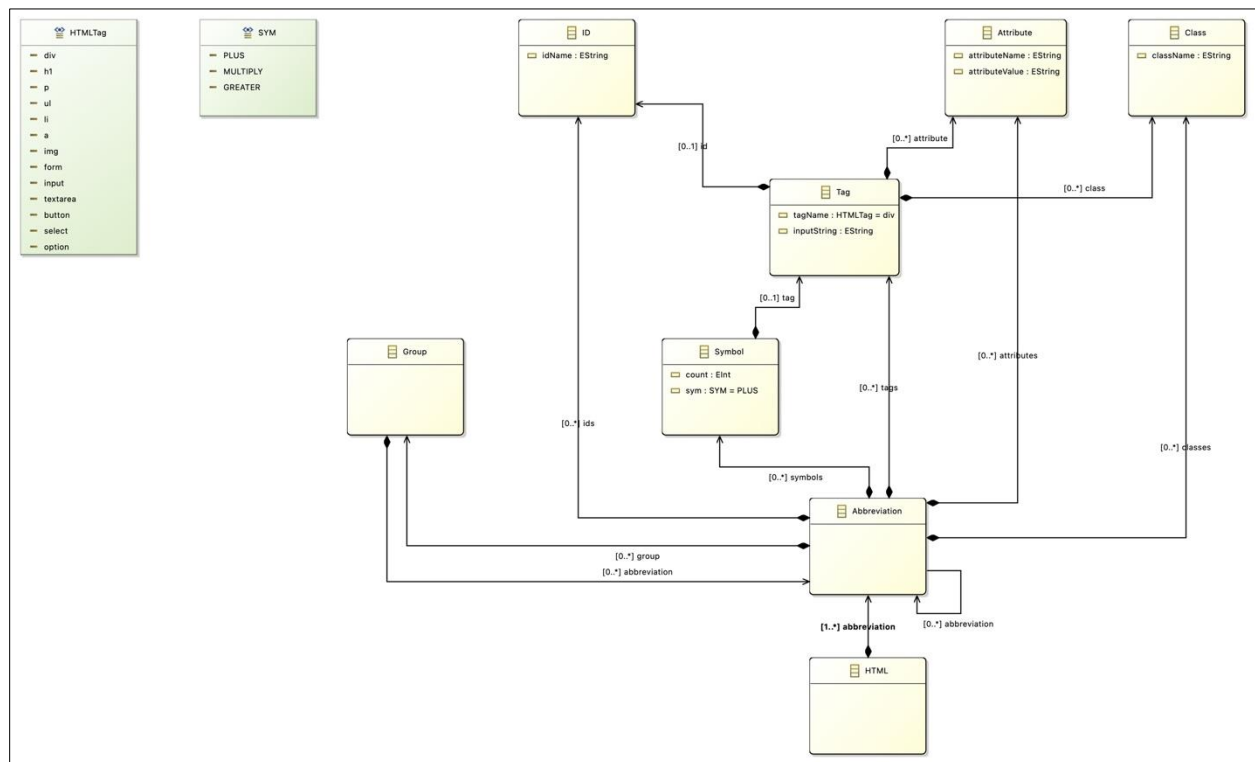


Figure 2-1: Meta-Model (Final)

Figure 2-1 shows our final version of the meta-model. The main object of our meta-model is the HTML class. It contains the list of abbreviation. Abbreviation has references to different classes including Tag, Symbol, Class, ID, and Attribute. To ensure we can have nested groups of tags, we also include composite relationship of tags with class, attribute, id and symbol. Tag can have many classes, attributes, and only one id. We are also making sure of this relationship with the validation as well. Tag has a property called tagName and inputString which is the type of the enum of HTMLTag defined in the meta model and EString respectively. This enum has a list of html tags from which user can make its abbreviation string.

2.1 Assumption

1. An abbreviation does not have multiple nested tags. For example, `{ul > li > h1}` is not acceptable as an abbreviation string.
2. An abbreviation string cannot start with Group class.
3. For now, our meta-model is only supported given HTML tags.

2.2 Alternative Design

We went through multiple implementations of meta-model for the improvement. As shown in the figure 2.2, StringInput was the main object to get started to use the application. It consisted of InputWord and inputSymbol. InputWord is supposed to be any element or attribute name and InputSymbol can be any symbol which we use to form the abbreviation like (., #,>, <,...etc). It also had composite relationship with the class InputMathOperator which has an InputMathDigit.

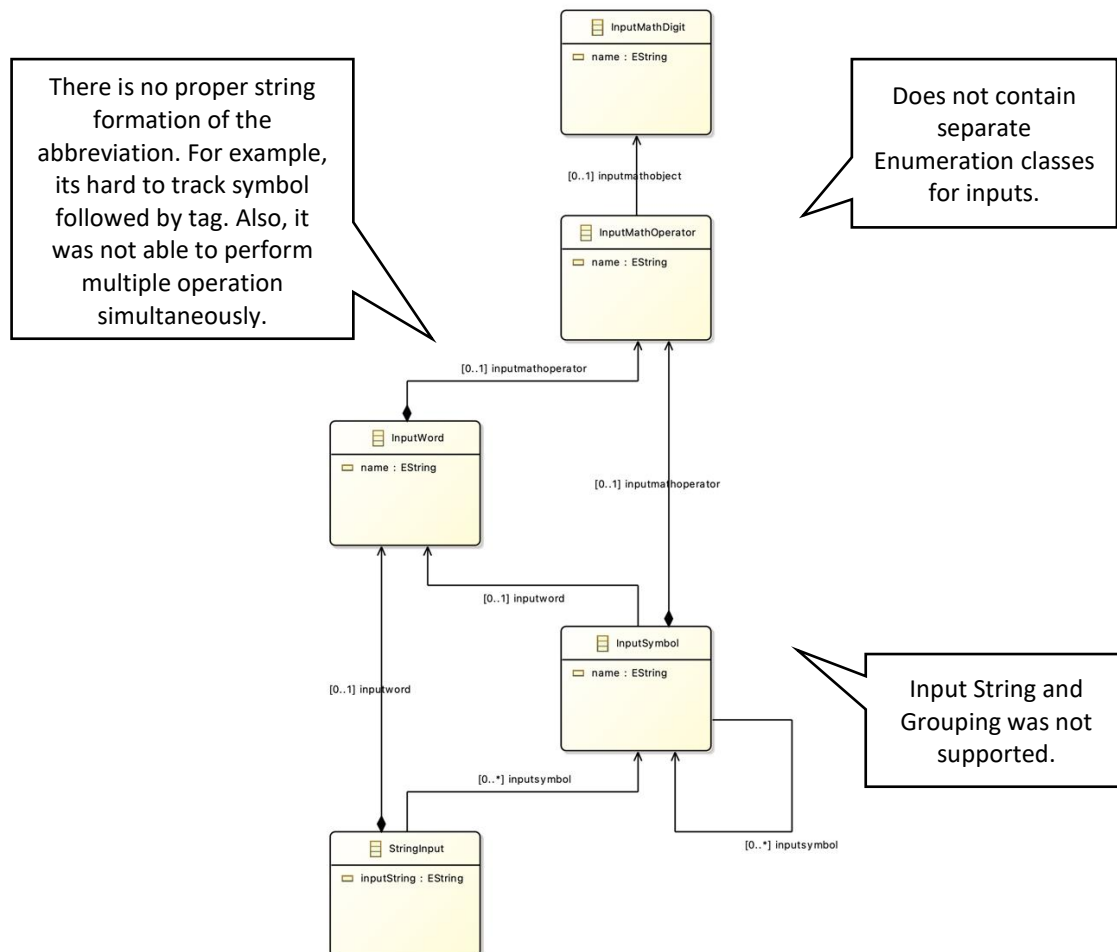


Figure 2-2: Meta-Model (old)

Final ecore file for can be found at WebMate.ecore.

3. Concrete Syntax and Editor

To define the concrete syntax for WebMate, we first started using GMF and then Eugenia. After working some days with these graphical editors (which was hard to use them) and talking with Professor Paige, we decided to use Xtext, because we need primarily a text-based editor. Although Xtext does not give us a graphical editor, it provides a text-based editor, and the final output of our program will be also in text. In addition, our program concepts are like regular expressions, therefore, Xtext is a suitable choice to define the grammars governing our concepts.

On the other hand, we should consider needs and preferences of the users. Stakeholders of this program are mainly HTML developers, and most of them prefer to work with textual editors because such tools allow them to write HTML code quickly and efficiently from shorthand notation.

When we run the Xtext, the editor is shown below (figure 3-1), will be launched. The editor suggests the user options according to defined syntax and user can write the expressions or select from options. Also, we set the extension for our model as **“.webmate”**.

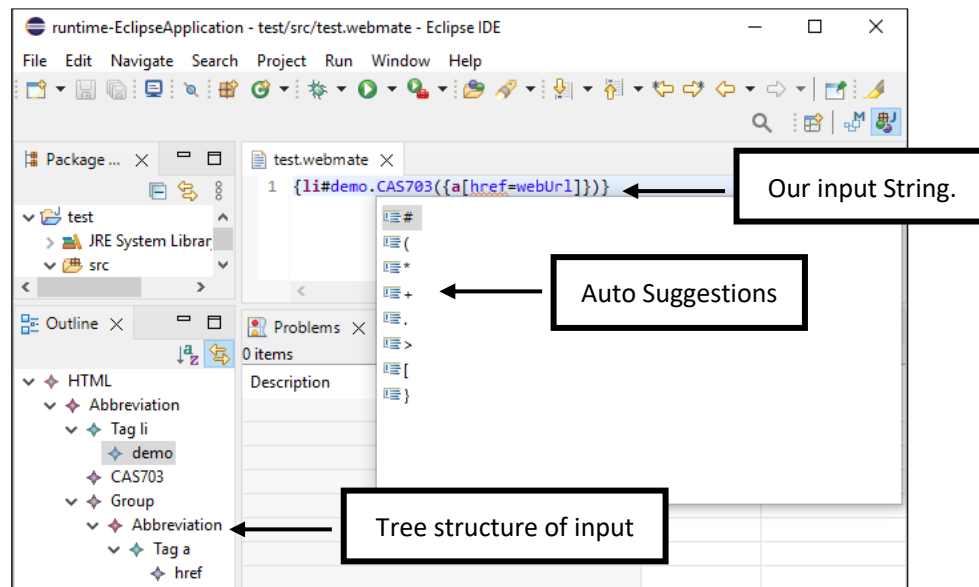


Figure 3-1: A Picture of Textual Editor for WebMate

Using Xtext, we specify our concrete syntax by defining different grammars for our language. We made a lot of changes to achieve final grammar from the first version of it. One of the main changes is the definition of ID, Symbol, Tag, Class, and Attribute which are depicted in figure 3-2 and figure 3-3. In addition to being **more readable and understandable**, the final version is much **more appropriate**, shows the structure of our language in comparison to first version. For instance, in the final version, we affirm that before **“id”**, the **“#” symbol** is needed and for Class and Attribute as well. In Symbol, user should use one of the defined symbols of language in

addition to a tag or a number and this **definition of Tag** can parse nested HTML tags. One of the weaknesses of the new grammar is that it has multiple alternatives for defining the tags, attribute, class, and id.

```
ID0 returns ID:
{ID}
'ID'
'{'
  ('idName' idName=EString)?
'}';

Symbol returns Symbol:
{Symbol}
'Symbol'
'{'
  ('count' count=EInt)?
  ('sym' sym=SYM)?
  ('tag' tag=Tag)?
'}';

Tag returns Tag:
{Tag}
'Tag'
'{'
  ('tagName' tagName=HTMLTag)?
  ('inputString' inputString=EString)?
  ('attribute' '{' attribute+=Attribute ( "," attribute+=Attribute)* '}' )?
  ('id' id=ID0)?
  ('class' '{' class+=Class ( "," class+=Class)* '}' )?
'}';

Class returns Class:
{Class}
'Class'
'{'
  ('className' className=EString)?
'}';

Attribute returns Attribute:
{Attribute}
'Attribute'
'{'
  ('attributeName' attributeName=EString)?
  ('attributeValue' attributeValue=EString)?
'}';
```

Figure 3-2: A picture of First Version of Grammar

```
ID0 returns ID:
{ID}
('#' idName=EString) ;

Symbol returns Symbol:
{Symbol}
(sym=SYM count=EInt) | (sym=SYM tag=Tag);

Tag returns Tag:
{Tag}
tagName=HTMLTag (class+=Class)* id=ID0? attribute+=Attribute* ('{' inputString=EString '}')?;

Class returns Class:
{Class}
'.' className=EString;

Attribute returns Attribute:
{Attribute}
 '[' attributeName=EString ( '=' attributeValue=EString )? ']' ;
```

Figure 3-3: A Picture of Final Grammar

The Xtext file for the grammar can be found at WebMate.xtext.

4. Validation

To check the validation of models conforming to WebMate meta-model, we found that we can use Xtext tool for validation as well, Instead of EVL². One of the most important benefits of using Xtext is that, it provides different options for adding Maven or Gradle internally, but if we want to integrate EVL with our editor, we need extra dependency to these tools. Therefore, we use Xtext to define constraints that we could not check in our meta-model.

There are a set of constraints for different tags and different attributes that our language covers, such as corresponding attributes for each tag, the lengths of some attributes and so on. For each part the appropriate message is generated for user.

The code in figure 4-1-part1 checks the input string to ensure that there is at least one element to start parsing.

```
@Check
public void checkAbbreviationIsPresent(HTML abb) {
    if (abb.getAbbreviation().size() == 0) {
        error(REQUIRED_ABBREVIATION, WebmatePackage.Literals.HTML__ABBREVIATION);
    }
    if (abb.getAbbreviation().get(0).getTags().size() == 0) {
        error("You should have atleast one tag", WebmatePackage.Literals.HTML__ABBREVIATION);
    }
}
```

Figure 4-1: Validity Check Using Xtext-Part1

If user wants to add “a” tag, “href” attribute must be added too, and for “input” tag, the “name” attribute must be identified.

```
public void checkTag(Tag tags) {
    if(tags.getTagName() == HTMLTag.A) {
        EList<Attribute> attributeList = tags.getAttribute();
        if(attributeList.size()==0)
        {
            error(A_TAG_HREF_ERROR, WebmatePackage.Literals.TAG__TAG_NAME);
        }
    }
    else {
        for(Attribute a: attributeList) {
            if(!a.getAttributeName().equals("href")) {
                error(A_TAG_HREF_ERROR, WebmatePackage.Literals.TAG__TAG_NAME);
            }
        }
    }
}

if(tags.getTagName() == HTMLTag.INPUT) {
    EList<Attribute> attributeList = tags.getAttribute();
    for(Attribute a: attributeList) {
        if(!a.getAttributeName().equals("name")) {
            error(INPUT_TAG_NAME_ERROR, WebmatePackage.Literals.TAG__TAG_NAME);
        }
    }
}
```

Figure 4-2: Validity Check Using Xtext-Part2

² Epsilon Validation Language

This piece of Java code investigates in attributes used for “img” tag, the attribute “src” must exist and the dimension of the image cannot be negative.

```
if(tags.getTagName() == HTMLTag.IMG) {
    EList<Attribute> attributeList = tags.getAttribute();
    for(Attribute a: attributeList) {
        if(!a.getAttributeName().equals("src")) {
            error(IMG_TAG_SRC_ERROR, WebmatePackage.Literals.TAG__TAG_NAME);
        }
        if(a.getAttributeName().equals("height")) {
            if(Integer.parseInt(a.getAttributeValue()) <= 0) {
                error(IMG_TAG_HEIGHT_ERROR, WebmatePackage.Literals.TAG__TAG_NAME);
            }
        }
        if(a.getAttributeName().equals("width")) {
            if(Integer.parseInt(a.getAttributeValue()) <= 0) {
                error(IMG_TAG_WIDTH_ERROR, WebmatePackage.Literals.TAG__TAG_NAME);
            }
        }
    }
}
```

Figure 4-3: Validity Check Using Xtext-Part3

If the tag user inputted is “class”, name attribute should prepared and it must not be null.

```
if(tags.getTagName() == HTMLTag.SELECT) {
    EList<Attribute> attributeList = tags.getAttribute();
    for(Attribute a: attributeList) {
        if(!a.getAttributeName().equals("name")) {
            error(SELECT_TAG_NAME_ERROR, WebmatePackage.Literals.TAG__TAG_NAME);
        }
    }
}
```

Figure 4-4: Validity Check Using Xtext-Part4

This function checks the condition for length of attribute name, we want the names with more than three characters.

```
@Check
public void checkAttribute(Attribute attribute) {
    if(attribute.getAttributeName().length() < 3) {
        error(ATTRIBUTE_ERROR, WebmatePackage.Literals.ATTRIBUTE__ATTRIBUTE_NAME);
    }
}
```

Figure 4-5: Validity Check Using Xtext-Part5

Please refer the real example for the [Code Validation](#).

The validation file with Xtext can be found at [WebMateValidator.java](#).

5. Model to Text Transformation

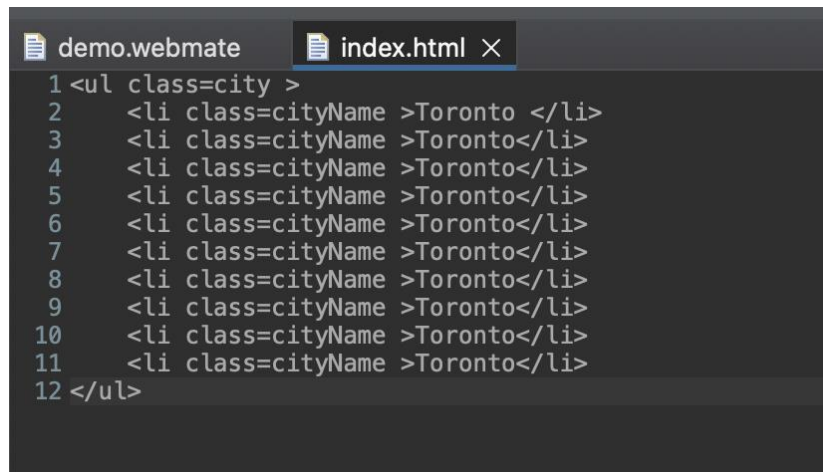
To obtain the text transformation from our meta-model, we wrote our main logic into the [WebMateGenerator.xtend](#) file, which generated when we ran artifacts from the [WebMate.xtext](#) file.

We declared the method called **toHTMLCode** which accepts the parameter of type Abbreviation. To use object of Abbreviation, we fetched the list of symbols and used as an input of our switch case, where we have a logic placed to generate the HTML code. Our codebase also has another method named **toHTMLTag**, which accepts the object of Abbreviation and Tag classes. It is basically responsible for attaching id, classes, attributes, and string to the HTML Tag. These methods are called by **overridden doGenerate** method, which is also responsible to create an html file named **"index.html"**. File gets generated right after saving the abbreviation string into the webmate.mydsl file.

5.1 Examples

1. **Input:** {ul.city > li.cityName{Toronto} * 10}

Output:

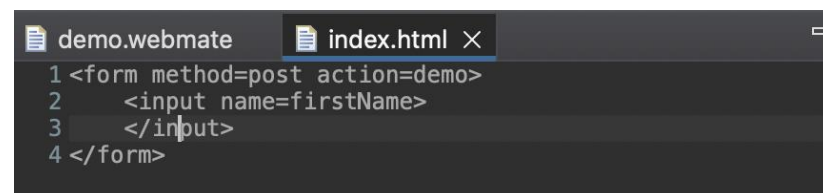


```
1 <ul class=city >
2   <li class=cityName >Toronto </li>
3   <li class=cityName >Toronto</li>
4   <li class=cityName >Toronto</li>
5   <li class=cityName >Toronto</li>
6   <li class=cityName >Toronto</li>
7   <li class=cityName >Toronto</li>
8   <li class=cityName >Toronto</li>
9   <li class=cityName >Toronto</li>
10  <li class=cityName >Toronto</li>
11  <li class=cityName >Toronto</li>
12 </ul>
```

Figure 5-1: Example 1

2. **Input:** {form[method=post][action=demo] > input[name=firstName]}

Output:

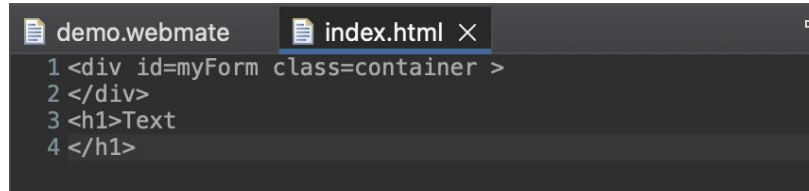


```
1 <form method=post action=demo>
2   <input name=firstName>
3   </input>
4 </form>
```

Figure 5-2: Example 2

3. Input: {div.container#myForm+h1{Text}}

Output:



```
1 <div id=myForm class=container >
2 </div>
3 <h1>Text
4 </h1>
```

Figure 5-3: Example 3

5.2 Challenges

At first, we faced problems regarding code generation for symbols in the abbreviation string, which led to change in the meta-model by adding a separate Enum called SYM to specify the symbols including plus, multiply, and greater. We also required to update our grammar to reflect our challenge faced during this step.

```
Symbol returns Symbol:
{Symbol}
(sym=SYM count=EInt) | (sym=SYM tag=Tag);
```

Hard to fetch these
Hardcoded symbols
during code generation.

Updated ecore and
grammar for parsing input
symbol.

```
Symbol returns Symbol:
{Symbol}
('*' count=EInt) | ('+' tag=Tag) | ('>' tag=Tag) | ('^' tag=Tag);
```

Another challenge that we faced was file generation. Due to some wrong setup of generation in “.mwe2” file, we were not able to generate the file called index.html. Finally, we solved it by looking and setting up property called generator property in GenerateWebMate.mwe2 file.

Please refer the real example for the [code generation](#).