Web Application Internationalization and Localization in Action

Terence Parr
University of San Francisco
parrt@cs.usfca.edu

ABSTRACT

A template engine that strictly enforces model-view separation has been shown to be at least as expressive as a context free grammar allowing the engine to, for example, easily generate any file describable by an XML DTD [7]. When faced with supporting internationalized web applications, however, template engine designers have backed off from enforcing strict separation, allowing unrestricted embedded code segments because it was unclear how localization could otherwise occur. The consequence, unfortunately, is that each reference to a localized data value, such as a date or monetary value, replicates essentially the same snippet of code thousands of times across hundreds of templates for a large site. The potential for cut-and-paste induced bugs and the duplication of code proves a maintenance nightmare. Moreover, page designers are ill-equipped to deal with code fragments. But the difficult question remains: How can localization be done without allowing unrestricted embedded code segments that open the door to model-view entanglement? The answer is simply to automate the localization of data values, thus, avoiding code duplication, making it easier on the developer and designer, and reducing opportunities for the introduction of bugs–all-the-while maintaining the sanctity of strict model-view separation. This paper describes how the ST (*StringTemplate*) template engine strictly enforces model-view separation while handily supporting internationalized web application architectures. Demonstrations of page text localization, locale-specific site designs, and automatic data localization are provided.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures, Patterns, Languages*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

Keywords

Internationalization, Localization, Template engines, Web applications, Model-View-Controller

1. INTRODUCTION

One of the most challenging issues facing web application developers is preventing duplication of business logic (the *model*) and presentation elements (the *view*). Preserving the principle of *single-point-of-change* is an absolute necessity for maintaining large web applications. Yet pressure exists in normal development to cut-and-paste logic and HTML templates, rather than properly factoring them, to get a job done quickly before a deadline. Cut-and-paste is particularly deadly when using presentation layers like Java server pages (JSP) and Active Server Pages (ASP) or any other Turing complete template engine because such presentation layers encourage entanglement of model and view–embedded expressions are unrestricted code fragments. Any copying of a presentation element then necessarily duplicates any embedded logic as well.

Localization, adapting an application for a particular locale, further increases pressure to duplicate program elements because proper infrastructure (*internationalization*) has not been generally available to factor out the presentation text from the HTML layout tags in a template file. Often developers will duplicate all the HTML template files, with a complete set of files for each supported language. Changing the look of the site afterwards means changing $n * f$ files for n languages and f files. If language requirements actually force different designs for certain pages, the unchanged pages should still not be replicated just to yield the complete set of pages.

Localizing entities like dates and integers per locale further complicates the situation by requiring different formats for some of the dynamically generated data values even when the presentation text strings remain the same. "10/01/06" means "October 1, 2006" in the US but "January 10, 2006" in the UK even though the surrounding presentation language is the same. Rendering entities in different formats provides an excuse for developers to embed the same code repeatedly in their templates to invoke the appropriate formatter invocation code, again breaking the single-point-of-change principle. Instead, entities should be clearly identified as dates, integers, or monetary values and then automatically rendered according to locale.

One final problem related to localization involves the technical sophistication of the page designers and language translators. Even graphics designers find it difficult sometimes to deal with highly-factored or otherwise complicated templates. Translators typically have no technical background and wading through HTML looking for text to translate can lead to errors and, worse, inadvertent changes to the formatting or embedded expressions. Both designers and translators will have difficulty with anything beyond the simplest data reference expressions. This is another rea-

ICWE'06, July 11-14, 2006, Palo Alto, California, USA.
ACM 1-59593-352-2/06/0007.

son that the text presentation strings should be separated from the HTML formatting and that localized template data references should not be arbitrary code fragments.

Supporting localization for dynamic web pages means then that an internationalized architecture must:

1. treat text as a resource, separate from the HTML, that is easily accessible to a translator
2. allow groups of templates to be treated as a unit to support different page designs for different locales
3. automatically render entities, such as numeric and monetary values, according to locale

StringTemplate (or ST as it is abbreviated), the subject of this paper, is a Java-based template engine (with ports to C# and Python) that strongly encourages developers to avoid HTML and code duplication during localization by providing a simple, flexible, and powerful infrastructure satisfying these requirements. This engine also strictly enforces model-view separation [7], which requires that all logic and computations be done in the model rather than replicated around the templates in fragments. The techniques described here work well in practice for real sites. For example, `SchoolLoop.com` uses ST to flip between English and Spanish with a single click. Exactly one template exists per page and all strings are pulled from a database.

This paper illustrates simple mechanisms for separating language specific text strings from page designs (section 2), supporting locale-sensitive site designs through template groups and group inheritance (section 3), and for automatically rendering data values according to locale (section 4).

2. LOCALIZING PAGE TEXT

An internationalized web application must treat text and image names (for images with embedded text) as dynamically generated data just like it treats actual data elements like monetary values so that translators can work in isolation from the web application and HTML page templates. On this most developers agree. Google's "Language Console", that has been used to provide the requisite Klingon locale among others, is a prime example. Differences arise, however, in how templates refer to these strings.

For the most part, developer trade articles [8][5] recommend a disciplined approach to inserting localized strings such as the following JSP snippet

```
<title><%=session.getValue("title")%></title>
```

where embedded Java code (not listed) inserts strings into the session variables. Here is the equivalent JSTL (JSP Standard Tag Library) solution:

```
<title><fmt:message key="title"/></title>
```

The Velocity template engine [10] allows a tidier solution

```
<title>$strings.get("title")</title>
```

where `strings` is an object in the model implementing the `Map` interface. ST has a similar, though more direct syntax for accessing strings in a `Map`:

```
<title>$strings.title$</title>
```

All of these solutions can pull strings from property files, resource bundles, data bases, and so on. The simplest way to switch strings tables per locale is to encode the locale in a properties filename, *language*`.properties`, and then load the appropriate file according to `Locale`'s `getLanguage()` method. Property files are simple key-value pairs. For example, an English property file, `en.properties` would have

```
title=Welcome to my test page
```

whereas a `fr.properties` file would have

```
title=Bienvenue à ma page de test
```

The page name can be encoded in the property name or a new property file can be used for each page in order to handle strings for multiple pages.

To demonstrate how a site can use ST to separate and edit strings, consider `SchoolLoop.com`'s administration view of a teacher's course managment page shown in Figure 1. The "+" buttons indicate all the strings that the designer or translator can edit live on the site. Each supported language (currently English and Spanish) has a collection of strings stored in a database and cached in memory as an object implementing the `Map` interface. References within the templates look like the following.

```
$strings.teacher_office_todo_title$
```

where `strings` is the attribute name associated with the cache object.

Clicking on a "+" button brings up the string edit view as shown in Figure 2. The designer or translator may alter the text string and refer to attributes, defined by the page controller, such as `ins_name` and `first_name`.

On the surface, just about all template engines appear to provide a satisfactory solution, but in practice any implementation built with an unrestricted template engine quickly degrades into a fully entangled model and view. In other words, more and more code will creep into the template files causing code duplication and other maintenance problems. Because ST strictly enforces model-view separation, developers are forced to keep the two concerns separated. Unless restricted, a template can readily alter the model, thus, making the template part of the model. For example, the developer could inadvertently `remove` instead of `get` the title, which might arise from an overzealous string search and replace operation or even from a designer's lack of programming skill:

```
<title>$strings.remove("title")</title>
```

Any subsequent pages would generate null or blank titles. Designers do not understand program "state" nor code and expecting them to deal with unrestricted template expressions is unrealistic and makes an application vulnerable to the introduction of random bugs.

Another relevant weakness of other template engines is that the various translated strings are just that: strings not templates. ST evaluates all expressions lazily implying that expressions within templates are not evaluated until the entire page has been constructed. Other template engine would have to invoke themselves explicitly and recursively on the incoming strings to evaluate them as templates,

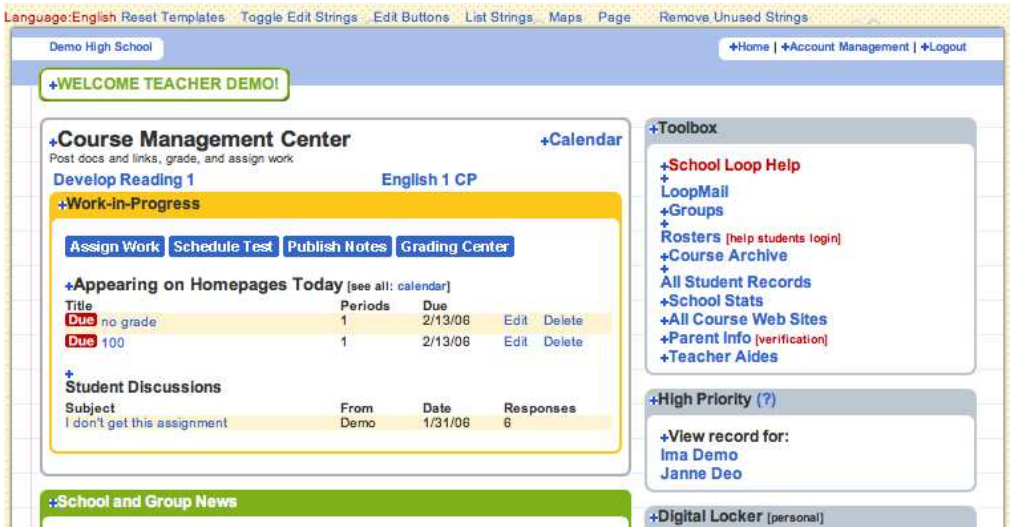

Figure 1: SchoolLoop.com's Admin View

Figure 2: SchoolLoop.com's Edit String View

increasing template complexity and again forcing code replication across all templates. Without lazy evaluation, it is unclear an engine could process expressions embedded in translated strings without computing the values too early. ST, in contrast, naturally deals with the following property strings:

```
title=Welcome to $username$'s test page
```

and

```
title=Bienvenue sur la page de test de $username$
```

References to `$strings.title$` would evaluate the embedded `$username$` expressions at the last possible moment, after the page controller had injected all data values (called *attributes*) for the surrounding template. Templates inherit all attribute values from their enclosing templates. After the `title` template is inserted, it has access to all page attributes. This mechanism assumes that the object implementing the `Map` interface called `strings` instantiates the `title` properties as ST objects not `String` objects.

By providing an internationalized architecture as described in this section, translators may create strings for various languages without forcing replication of code or HTML page templates. In some cases, however, the locale is so different that merely changing the text strings is insufficient–locales may require their own full or partial site designs.

3. SITE DESIGN PER LOCALE

Web applications easily deal with multiple site designs, albeit often by simply copying the directory full of template files and altering to suit. But what if the site design is almost identical and you just need to alter a few page design such as for a premium and non-premium version of the same site? Germane to this discussion, how can you provide multiple designs for a single page to deal with various locales without duplicating your entire site for each locale? For example, if the text direction changes to right-to-left from left-to-right, the page layout might have to change.

ST supports template group inheritance so that templates not found in one group may be inherited from a *supergroup*. Just as in object-oriented programming where new objects may be defined as they differ from existing objects, new template groups may be defined as they differ from existing template groups. In this way locales requiring special designs do not force the replication of unaltered templates. Only those pages that are different need be specified.

A simple example illustrates the mechanics. An object called `StringTemplateGroup` represents a group of templates stored in a particular directory. The following code fragment creates a group rooted at "`/var/data/templates/`*`language`*" where *`language`* would be `en`, `fr`, etc...

```
Locale locale = Locale.getDefault();
String language = locale.getLanguage();
String root = "/var/data/templates/";
StringTemplateGroup templates =
  new StringTemplateGroup(language, root+language);
```

The page controller responsible for generating a page ultimately invokes the `getInstanceOf()` method to get an instance of a template from a group:

```
StringTemplate testPageST =
  templates.getInstanceOf("test");
testPageST.setAttribute("username", ...);
...
```

and then invokes `testPageST.toString()` to render the template to plain text for transmission to the user's browser.

ST's group inheritance mechanism is dynamic in that new group hierarchies may be defined at run-time. Defining a template subgroup involves creating a new directory to hold subgroup templates, creating a new `StringTemplateGroup` instance attached to that directory, and finally setting the new group's supergroup to an existing group.

For example, if the `test` page design for the French locale requires a new design, but all the other page designs stay the same, the developer would create a new directory called "`fr`" and place the new page design template inside with the same template file name as before. The following code fragment creates a subgroup that inherits all templates from the English group except for the `test` page, which would be overridden in the French group. No template or code duplication would occur.

```
StringTemplateGroup english =
  new StringTemplateGroup("en", root+"en");
StringTemplateGroup french =
  new StringTemplateGroup("fr", root+"fr");
french.setSuperGroup(english);
...
// set according to locale; I explicitly set here
StringTemplate templates = french;
```

Later references to `templates.getInstanceOf("test")` load templates from the appropriate group depending on which group `templates` pointed to. Changing site design per locale then is simply a matter of flipping a pointer.

One final detail is worth mentioning, that of file character encoding. Presentation files will most likely need different character encodings than ASCII, such as ISO-8859-1 or UTF-8. Developers set the template character encoding for a `StringTemplateGroup` via a simple method before template instances are loaded from disk; e.g.,

```
french.setFileCharEncoding("ISO8859_1");
```

ST supports internationalization through simple mechanisms for separating text strings from HTML templates and by allowing new groups of templates to be defined as they differ from previous groups. Both of these mechanism prevent unnecessary template and code duplication, but do not solve the final and most challenging problem of locale-sensitive data formatting.

4. LOCALIZATION OF DATA VALUES

Many different data values in an application need to be formatted according to locale such as numbers, dates, monetary values, and percentages. Some applications use a brute force approach by directly invoking formatting objects that are sensitive to locale in expressions embedded within templates. Here is an example using a Velocity `DateTool` object stored in a variable called `date`:

```
Sold on $date.format('medium',$myDate).
```

This is rather verbose considering many date references will have the same format and, more importantly, this snippet will be difficult for designers to deal with. Even at the most basic level, an issue will be: When does the expression stop and HTML begin again? Using StringTemplate the designer would simply reference `$myDate$` where the "`$`" characters clearly bracket every expression and defers locale issues to ST.

The formatting of integer values makes this matter even more clear. A simple reference to a data value of type `Integer` such as `$n$` yields the string computed from invoking `Integer`'s `toString()` method rather than a localized string in most template engines. FreeMarker [3] does allow the developer to set a global number format, but beyond the common types there is no general mechanism for specifying formatting for arbitrary types.

ST introduces an extremely simple and general mechanism to format arbitrary data values, one that does not require designers to alter templates when internationalizing an application nor requires them to be programmers. The goal is to allow both single-language sites and internationalized sites to use `$n$` and `$myDate$` instead of code fragments.

This solution relies on ST's restricted interface between model and view. In a web application based upon ST, the page controller pulls data from the model and injects attributes into templates, which render objects to string via the `toString()` method. The only code execution initiated by a template is the implicit invocation of `toString()` methods during evaluation. For example, the following code fragment sets the `n` attribute for an instance of the `test` template.

```
StringTemplate testPageST =
  templates.getInstanceOf("test");
testPageST.setAttribute("n", new Integer(3));
```

References to `$n$` in a template are evaluated by calling `toString()` on the `Integer` object associated with `n` in the attribute table for that template.

The collection of all `toString()` methods can be viewed as the *renderer* component of a design pattern more complete than *MVC*: *MVCR* (*model-view-controller-renderer*) [7]. A template knows to format `n` as a locale-sensitive number if an attribute renderer for the `Integer` class type exists. The developer may register renderers per template group or per template instance. If a template does not have a renderer for a type, the associated group is consulted. If no renderer is registered for the group, its supergroup is consulted. A renderer is any object that satisfies the `AttributeRenderer` interface:

```
public interface AttributeRenderer {
    public String toString(Object o);
}
```

Here is a simple example of a renderer that formats integers according to locale.

```
public class IntegerRenderer
  implements AttributeRenderer
{
  public String toString(Object o) {
    Integer value = (Integer)o;
    NumberFormat nf =
      NumberFormat.getIntegerInstance();
    return nf.format(value.intValue());
  }
}
```

After the developer has registered the renderer with a group:

```
templates.registerRenderer(
  Integer.class,
  new IntegerRenderer());
```

any template instances created from that group will format `Integer` objects via `IntegerRenderer`'s `toString()` method.

What if, as a special case, a number must be formatted in binary ("`101`") rather than decimal ("`5`")? In this case, the easiest thing to do is to wrap the `Integer` attribute in an object whose `toString()` method does the appropriate conversion to binary digits:

```
Integer bits = 5;
StringTemplate st = templates.getInstanceOf("test");
// $mask$ will render in binary
st.setAttribute("mask", new BinaryWrapper(bits));
```

The controller must, in a sense, "paint" the object with a new coat of paint for this special case, but it is better than asking the nonprogrammer designer to write:

```
$numbers.binaryFormat($mask)
```

In the view of ST, the programmer provides the set all possible data values and the designer chooses from among them just like a library provides an API for programmers. This seemingly rigid strategy is necessary not only to enforce strict separation of model and view but also to deal with the realities of a designer's skill set.

What happens when the designer needs dates to be sometimes long and sometimes short format? This case is also easily handled without resorting to unrestricted code in the template and without burdening the programmer with the extra code needed to wrap all date attributes. The `Date` class can be subclassed to provide property methods such as `getMedium()` so that templates may reference `$myDate$` to get the default rendering and `$myDate.medium$` to get a medium format. For any given type, the developer may automatically provide augmented properties without having to manually create instances of a `Date` subclass. The `setAttribute()` method of the ST class can be overridden to trap and automatically wrap objects of a particular type to provide new properties:

```
class MyStringTemplate extends StringTemplate {
  public void setAttribute(String name, Object v) {
    if ( v instanceof Date ) {
      v = new DateWrapper(v); // adds medium, ...
    }
    super.setAttribute(name, v);
  }
}
```

To create `MyStringTemplate` objects rather than the default, the template group factory `createStringTemplate()` method is overridden. All attribute references to objects of type `Date` will support the added properties automatically without burdening the application programmer and without expecting the designer to be a programmer.

```
Date d = db.getUserLastLogin(id);
// d is automatically wrapped by setAttribute
// designer references $lastLogin$ or
// $lastLogin.medium$, ...
st.setAttribute("lastLogin", d);
```

This strategy also works nicely for `Integer`s so that `$n$`, `$n.binary$`, and `$n.currency$` are available to designers.

In summary, localizing data values for a dynamic web site should involve changes in the plumbing rather than additions of code in the template. Moreover, most of this plumbing can be automated to reduce the load on programmers. Attribute references should be simple like *name* or *name.property* because that is what designers understand and also because it prevents identical code snippets from being replicated thousands of times across hundreds of templates. The MVCR pattern, espoused by ST, is completely general and is easily understood by the average programmer.

5. RELATED WORK

A few engines are making progress towards automatically dealing with the localization of data values. FreeMarker [3] can automatically format numeric and time related values by locale. MonoRail [6] has "filters" that appear to automatically format dates. Both tools do not appear to allow general type-to-renderer mappings. Other tools require the programmer to write a code fragment for each localized data reference.

Most engines provide a mechanism to pull out text strings for translation, but do not provide formalisms such as template groups to deal with multiple site look-and-feels. The template groups of ST appears unique in their ability to naturally flip between template groups and to support partial designs that derive from other designs via group inheritance and template polymorphism.

After reviewing a draft of this paper, German developer Kay Roepke related his experiences building two internationalized web sites: `mobile.de` (an eBay company specializing in new and used vehicle sales) and `openBC.com` (an online professional networking management site for Europe and Asia). At `mobile.de`, the developers built a company-specific system very similar to ST that had very restricted template expressions. Ultimately, this system proved to be a performance problem mostly due to its Perl-based implementation. To avoid such performance problems in his next position at `openBC.com`, a simpler, but faster system was used. Unfortunately, the second system did not automatically handle localized data–data was formatted in the controller and pushed into the templates.

Roepke corroborates the suppositions and conclusions of this paper:

> I concur with your conclusion, that the fundamental problem is the support for expressions within templates. This quickly leads to two things:
>
> 1. unmaintainable applications due to lack of single-point-of-change in code
> 2. most syntactic [template expression] conventions either make the use of dedicated HTML editors impossible or are unintelligible to the average designer. I have seen horrible things happen to templates which have been in the hands of designers and/or translators.

A literature search reveals little activity from academics concerning internationalization and localization (particularly of web applications), probably because it is a question of engineering not pure research. The available papers tend to describe the general requirements and strategies for building multi-lingual applications [11] or the very need for internationalization [1]. Some focus primarily on the design methodology [9].

6. CONCLUSIONS

With few exceptions such as XMLC [2], engine designers have unrealistic expectations of programmer discipline. All provide Turing complete unrestricted embedded template languages thereby repeating the mistakes of JSP except with a new programming language to learn. New designs or partial designs often require replication of the embedded code fragments and, therefore, code changes require modifications to all versions of the same page.

Designers are not programmers and have trouble understanding how a table is generated with a for-loop and just generally have issues finding and properly altering HTML entangled with code. From my experience building sites such as `jGuru.com` and `antlr.org`, it is clear designers can deal with only the simplest template expressions. To collaborate and to work in parallel, designers must have templates with restricted expressions.

Internationalized architectures make matters worse. References to "`strings.get("title")`" used to separate text strings from HTML are not suitable for designers. Even the JSTL tags like

```
<fmt:message key="title"/>
```

are a problem because most designer cannot competently build HTML by hand, meaning these tags are definitely too much. One can question how engine designers expect the majority of page designers to work with their templates.

ST's distinguishes itself by strictly enforcing model-view separation via template expressions that are restricted syntactically and semantically. Other engines have rejected such draconian measures for fear of emasculating template power. The ability to specify locale-specific data value formats is one such area of concern. The primary contribution of this paper is to show that not only can a restricted template engine support internationalized architectures, but it can do so automatically thereby reducing the burden on developers and HTML designers alike.

Oddly enough, sticking to the principle of strict separation requires a solution to localizing data values that is simpler, more flexible, and more powerful than existing solutions. Other engines are Turing complete and can mimic ST's strategy, which I implore them to do, but they cannot overcome the fundamental weakness of not enforcing model-view separation. That weakness leads to code creeping into templates ultimately leading to code and HTML replication and also to HTML templates with which designers cannot work.

ST is available under the BSD license from `http://www.stringtemplate.org`.

7. ACKNOWLEDGMENTS

I would like to thank Anton Keks for suggesting the type to renderer object mapping and Thomas Aigner for pointing out the character encoding problem and workable solution.

8. REFERENCES

[1] *How We Made the Web Site International and Accessible: A Case Study*. Maria Gabriela Alvarez, Leonard R. Kasday, and Steven Todd 4th Conference on Human Factors & the Web. June 1998.

[2] Enhydra. XMLC. `http://xmlc.enhydra.org/project/aboutProject/index.html`

[3] FreeMarker. http://freemarker.sourceforge.net.

[4] JSP. http://java.sun.com/products/jsp.

[5] Sing Li. *Create internationalize JSP applications*. `http://www-128.ibm.com/developerworks/java/library/j-jspapp` March 2005.

[6] MonoRail `http://www.castleproject.org/index.php/MonoRail`

[7] Terence Parr. *Enforcing Strict Model-View Separation in Template Engines*. In WWW2004 Conference Proceedings p. 224, May 17-20 2004, New York City.

[8] Govind Seshadri. Internationalize JSP-based Websites. `http://www.javaworld.com/jw-03-2000/jw-03-ssj-jsp.html`, March 2000.

[9] De Troyer, O. and Casteleyn, S. (2004). *Designing localized web sites*. In 5th International Conference on Web Information Systems Engineering (WISE 2004), volume 3306, pages 547558. Springer.

[10] Velocity. `http://jakarta.apache.org/velocity/index.html`

[11] *A Framework for the Support of Multilingual Computing Environments*, Yip Chi Technical report TR-97-02 University of Hong Kong.