Enforcing Strict Model-View Separation in Template Engines

Nominated for best paper

Terence Parr
University of San Francisco

ABSTRACT

The mantra of every experienced web application developer is the same: *thou shalt separate business logic from display*. Ironically, almost all template engines allow violation of this separation principle, which is the very impetus for HTML template engine development. This situation is due mostly to a lack of formal definition of separation and fear that enforcing separation emasculates a template's power. I show that not only is strict separation a worthy design principle, but that we can enforce separation while providing a potent template engine. I demonstrate my `StringTemplate` engine, used to build jGuru.com and other commercial sites, at work solving some nontrivial generational tasks

My goal is to formalize the study of template engines, thus, providing a common nomenclature, a means of classifying template generational power, and a way to leverage interesting results from formal language theory. I classify three types of restricted templates analogous to Chomsky's type 1..3 grammar classes and formally define separation including the rules that embody separation.

Because this paper provides a clear definition of model-view separation, template engine designers may no longer blindly claim enforcement of separation. Moreover, given theoretical arguments and empirical evidence, programmers no longer have an excuse to entangle model and view.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures, Patterns, Languages*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Languages

Keywords

Template engines, Web applications, Model-View-Controller

1. INTRODUCTION

The need for dynamically-generated web pages, such as the book description pages at Amazon.com, has led to the development of numerous template engines in an attempt to make web application development easier, improve flexibility, reduce maintenance costs, and allow parallel code and HTML development. These enticing benefits, which have driven the proliferation of template engines, derive entirely from a single principle: separating the specification of a page's business logic and data computations from the specification of how a page displays such information. With separate encapsulated specifications, template engines promote component reuse, pluggable site "looks", single-points-of-change for common components, and high overall system clarity.

I have discussed the principle of separation with many experienced programmers and have examined many commonly-available template engines used with a variety of languages including Java, C, and Perl. Without exception, programmers espouse separation of logic and display as an ideal principle. In practice, however, programmers and engine producers are loath to enforce separation, fearing loss of power resulting in a crucial page that they cannot generate while satisfying the principle. Instead, they *encourage* rather than *enforce* the principle, leaving themselves a gaping "backdoor" to avoid insufficient page generation power.

Unfortunately, under deadline pressure, programmers will use this backdoor routinely as an expedient if it is available to them, thus, entangling logic and display. One programmer, who is responsible for his company's server data model, told me that he had 3 more days until a hard deadline, but it would take 10 days to coerce programmers around the world to change the affected multilingual page displays. He had the choice of possibly getting fired now, but doing the right thing for future maintenance, or he could keep his job by pushing out the new HTML via his data model into the pages, leaving the entanglement mess to some vague future time or even to another programmer.

The opposite situation is more common where programmers embed business logic in their templates as an expedient to avoid having to update their data model. Given a Turing-complete template programming language, programmers are tempted to add logic directly where they need it in the template instead of having the data model do the logic and passing in the boolean result, thereby, decoupling the view from the model. For example, just about every template engine's documentation shows how to alter the display according to a user's privileges. Rather than asking simply if the user is "special", the template encodes logic to compute whether or not the user is special. If the definition of special changes, potentially every template in the system will have to be altered. Worse, programmers will forget a template, introducing a bug that will pop up randomly in the future. These expedients are common and quickly result in a fully entangled specification.

Is the ideal possible? That is, can one enforce separation without emasculating the power of the template engine? Does a strict engine result in some pages that are incomputable? As usual, theory and practice disagree. Theory dictates that a Turing-complete template engine can generate any page and is strictly stronger than a restricted template engine. In practice, my experience building web

WWW2004, May 17–20, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

servers, such as jGuru.com (110k lines), provides strong evidence that programmers can in fact get the best of both worlds: separation and sufficient power. For the past five years, I have essentially conducted an experiment in software design while building sites and tweaking my template engine called `StringTemplate`, rigidly avoiding logic and computation in my templates. Still, this anecdotal evidence does not prove my restricted engine is sufficient. On the other hand, while most template engines can approximate $sin(x)$ via a Taylor series, I'm unconvinced a designer will need the ability.

The trick is to provide sufficient power in a template engine without providing constructs that allow separation violations. After examining hundreds of template files used in my sites, I conclude that one only needs four template constructs: attribute reference, conditional template inclusion based upon presence/absence of an attribute, recursive template references, and most importantly, *template application* to a multi-valued attribute similar to lambda functions and LISP's `map` operator.

My goal here is to show that strict enforcement of separation is not only a worthy design principle, but that it is realistically achievable through the appropriate choice of template constructs. Section 2 describes how template engines evolved from previous strategies to solve very real design issues. Section 3 explicitly states the benefits of separation. Section 4 shows how the model-view-controller pattern very naturally applies to server design and how it helps visualize separation of logic (model) and display (view). Sections 5 and 6 formally define templates and three restricted template classes. Sections 7 and 8 define strict model-view separation and how to avoid entanglement. Finally, Section 9 demonstrates the `StringTemplate` engine solving some nontrivial HTML tasks.

2. EVOLUTION OF TEMPLATE ENGINES

Generating dynamic pages implies a server no longer maps a URL to an HTML file on the disk. Instead the server must map URLs to a chunk of code that spits out the appropriate HTML containing data and associated display instructions. I will describe the evolution of Java engines in particular here, though all the concepts apply to Perl, VisualBasic, and so on.

Java began supporting server development with Servlets [15], which amounted to methods that respond to HTTP `GET` and `POST` commands by generating HTML via print statements. For example, here is the core of a simple servlet that generates a page saying "hello" to the URL parameter called `name`.

```
out.println("<html>");
out.println("<body>");
out.println("<h1>Servlet test</h1>");
String name = request.getParameter("name");
out.println("Hello, "+name+".");
out.println("</body>");
out.println("</html>");
```

The problem is that specifying HTML in Java code is tedious, error-prone, and cannot be written by a graphics designer. To improve the situation, programmers can try to factor out common HTML output elements into Java rendering objects like `Table` and `BulletList`, but they are still embedding HTML all through their servlets.

The next stage of evolution was the introduction of Java Server Pages (JSP) [6], which at first glance seemed to be a big step forward. JSP files are essentially inside-out servlets, HTML files with embedded Java code. JSP files are automatically translated to servlets by the server. The same "hello" page looks like the following in JSP:

```
<html>
<body>
<h1>JSP test</h1>
Hello, <%=request.getParameter("name")%>.
</body>
</html>
```

JSP files may start out as HTML files with simple references to data, as in this example, but they quickly degenerate into fully entangled code and HTML specifications like Servlets. In practice, graphics designers also cannot modify JSP files. Most importantly JSP encourages bad object-oriented design. For example, an include file is a poor substitute for class inheritance. JSP pages may not be subclassed, which makes it hard for programmers to factor out common code and HTML. Hunter [5] summarizes a few other problems with JSP such as JSP's inelegant looping mechanism to display lists.

While JSP was not the answer, it did put the idea of a template into programmers' minds. A template is an HTML document with "holes" in it that can be filled with data or the results of simple actions. Unfortunately, just about every template engine repeats the mistakes of JSP; they provide a Turing-complete tool-specific programming language embedded in HTML. As with JSP, then, designers are forced to imagine the emergent behavior of a program rather than looking at a page exemplar.

While template engines have fixed some of the annoying issues with JSP, most fail to address the primary reason why JSP is a disaster for large systems.[1] A template should merely represent a view of a data set and be totally divorced from the underlying data computations whose results it will display. If the template language is too powerful, template designers risk entangling template and business logic. The following section details why such entanglement should be avoided.

3. MOTIVATIONS FOR SEPARATION

The primary goal behind using a template engine is to separate logic and data computations from the display of such data in both thought and mechanism. In the case of web site development, this means roughly that there should be no code in HTML and no HTML in code. Here some really good reasons why programmers and designers want the separation:

1. **encapsulation**: the look of a site is fully contained within templates and, likewise, the business logic is located entirely in the data model. Each specification is a complete entity.

2. **clarity**: a template is not a program whose emergent behavior is an HTML page; a template *is* an HTML file that a designer or programmer can read directly.

3. **division of labor**: a graphics designer can build the site templates in parallel with the coders' development efforts. This reduces the load on programmers not only by having a (usually less expensive) designer building the templates, but by reducing communication costs; designers can tweak the HTML without having to talk to a programmer. At jGuru.com we repeatedly verified our ability to have our graphics designer operate independently from our coders.

4. **component reuse**: just as programmers break large methods down into smaller methods for clarity and reusability, designers can easily factor templates into various sub-templates

[1] While the page URLs at jGuru.com end in `.jsp`, they are not JSP files at all. After departing from JSP, we left the page names for backward link compatibility.

such as gutters, navigation bars, search box, data lists, and so on. Entangled templates tend to be difficult to factor and cannot be reused with other data sources.

5. **single-point-of-change**: by being able to factor templates, the designer can abstract out small elements like links and larger items like user record views. In the future, to change how every list of users, for example, appears on a site, the designer simply changes one template file. This also avoids the errors introduced by having multiple places to change a single behavior. Having a single place to change a single behavior in the model is also important. Logic in the template is to be avoided because "is admin user" logic is repeated in many templates.

6. **maintenance**: changing the look of a site requires a template change not a program change. Changing a program is always much riskier than changing a template. In addition, template changes do not require restarting a live, currently-in-use server application whereas code changes normally require recompilations and restarts.

7. **interchangeable views**: with an entangled data model and display, the designer cannot easily swap in a new look for such things as site "skins." At jGuru.com, each site look is a collection of templates called a *group*. Some groups are radically different (as opposed to just changing some color preferences and font sizes). A simple pointer tells the page controller which template group to use when displaying pages.

8. **security**: templates for page customization are commonly made available to bloggers but, like macros in Microsoft Word, unrestricted templates pose a serious security risk. After suffering numerous attacks related to bloggers invoking class loaders, squarespace.com switched from using Velocity [19] to using `StringTemplate`. One could also imagine a simple, but effective attack in the form of an infinite loop. Strictly isolating the model from the view and disallowing control structures, as championed in this paper, equals security.

Programmers often view strict separation as costing more time and hassle. While this may be true in the small, such as adding a new piece of data to a view, my experience is that projects progress much faster in the long run and result in much more flexible, robust code specifically because of the significant benefits listed in this section.

4. MODEL VIEW CONTROLLER PATTERN

The well-known *model-view-controller* pattern [9] applies nicely to web server design and identifies three general "realms" where various pieces of the page generation process belong. At the coarsest level, the *view* represents a page's template or templates, the *controller* represents both the server's dispatch infrastructure that maps a URL to a code snippet and the code snippet itself, and the *model* represents an application's data (the "state"), most of the business logic, and any model-related computations.

Unfortunately, programmers cannot agree precisely where to draw the lines separating the three realms. A mailing list thread from 2002 entitled "separating C from V in MVC" [12] is representative of these arguments. This uncertainty arises mostly because the boundaries between controller and model are usually blurred. In contrast, there is widespread agreement that keeping the view as simple and free from entanglements with model and controller as possible is a worthwhile ideal to strive for.

In my experience, the controller should be as lightweight as possible. The controller acts like an event manager, dictating *when* a page's code is executed. The various page code snippets are also part of the controller and should limit their activities to pulling the appropriate data from the model, session, or parameters then pushing that data into the view. If the page is the target of an HTTP `POST` or other processing page, the page triggers actions in the model. A page may include some data processing specific to that page, but in general it is best to factor out code from the page into the model that affects the model or embodies a common operation. Long sequences of method calls to the model from a page should also be extracted and refactored as a single method in the model. For example, a page can do simple filtering of data (probably using a general purpose filter mechanism provided by the model) but an operation like "process forum entry submission" should be coded in the model and merely invoked by a processing page.

The model contains all of the business logic, computations, and "state" (the persistence layer such as a database). I have seen student projects and commercial programmers' applications with SQL statements in their controllers and, shockingly, in their views. Any change to the schema forces changes to all SQL spread throughout pages and views. SQL tables are rarely the appropriate level of abstraction anyway; a model should process raw database data into objects and relationships between objects. The model also encapsulates commonly-executed operations such as "purchase book" or "register and send email."

The view should only specify how to display data processed by the model and made available by the controller. A graphics designer should feel very comfortable writing and manipulating view specifications. The view should not be part of the program, meaning that it should not alter the model nor process data. A good test of how well the view is isolated is whether or not the view could be used in another application. For example, if a template displays a list of elements in a table, could the template be reused it in a bookstore application or a forum hosting site?

Where to draw the line between the controller and model is sometimes fuzzy and depends on the application, but the line separating them from the view is clear.

5. TEMPLATE DEFINITION

A literature search does not reveal a formal definition of a template and, indeed, Hunter [5] lists this as one problem with the template engine market. While the notion of a template is straightforward, a formal treatment will provide a common nomenclature and a means of classifying template engines and their generational power.

Fundamentally, an output template differs from a program that generates output in that the template is an exemplar whereas the program's computation results in the output. A template is an output document with embedded actions that the engine evaluates when rendering the template. A JSP file is example of an *unrestricted template*.

DEFINITION 1. *An* unrestricted template, T, *is an alternating list of output literals,* t_i*, and action expressions,* e_i*:*

$t_0 e_0 ... t_i e_i t_{i+1} ... t_{n_t} e_{n_e}$

where any t_i *may be the empty string and* e_i *is unrestricted computationally and syntactically. If there are no* e_i *in* T *then* T *is just a single literal,* t_0.

THEOREM 1. *An unrestricted template is equivalent to an unrestricted Chomsky type 0 grammar and, hence, may generate the recursively enumerable languages.*

PROOF. Because e_i is unrestricted computationally, it may embody a Turing machine and, thus, may generate the type 0 languages all by itself. □

This result is self-evident and not very interesting, but it is worth stating explicitly that unrestricted templates are the most powerful and can generate any desirable output (i.e., anything a Turing machine can compute).

Generating a document in some language L may require multiple, nested templates, therefore, individual literals t_i and the output of individual templates may or may not conform to L or even subsentences of L.

Most often the literals are separated lexically from the expressions with special symbols. For example, here is a simple template using `HTML::Template` [18] that generates an HTML body:

```
<p>Hello, [% user %].
```

where t_0 = "`<p>Hello, `", e_0 = "`$user$`", and t_1 = "`.`". The expressions are embedded between literals and lexically-delimited via `[%...%]`, in this case.

A template may encode expressions using the surrounding language's lexicon in order to conform to L. For example, XMLC [2], uses the HTML `SPAN` tag:

```
<p>Hello <SPAN id="user">Sample name</SPAN>.
```

Unrestricted templates can alter the model and can compute any computable function. They can also use context information to alter the output. For example, a template could specify that the color of a table row i be a function of $sin(i)$ with $sin(i)$ being computed using a Taylor series approximation coded inside the template itself.

Interestingly, by this definition of a template, XSL [20] style sheets are not templates at all because style sheets specify a set of XSLT tree transformations whose emergent behavior is an XML or HTML document. XSL style sheets are programs like servlets, albeit declarative in nature rather than imperative.

It is worthwhile to point out finally that, just because templates are specified with a textual specification, templates are not limited to generating text. For example, before generating text, many source-to-source translators perform a series of tree transformations. Here is a template for an assignment tree using (an experimental version of) ANTLR's [14] LISP-like tree notation:

```
#(ASSIGN <ID> <expr>)
```

where `ASSIGN` is a literal root with two expressions for children, `<ID>` and `<expr>`, whose values are computed and filled in before constructing the tree.

6. RESTRICTED TEMPLATE CLASSES

Unrestricted templates are extremely powerful, but there is a direct relationship between a template's power and its ability to entangle model and view. The more powerful a template, the closer it becomes to a Turing-complete language. Further there is an inverse relationship between power and template suitability for non-programmers such as graphics designers.

One of the biggest entanglement problems faced by unrestricted templates is that a template can affect the model. If a template can modify the model, it is part of the program, hence, separating model from view should begin by restricting a template to operate on a set of incoming read-only data values, called *attributes*, that are computed by the model. Attributes may be single-valued like `4521`, multi-valued like as `[Jim, Frank, John]`, or aggregates like `(name=Tom,ext=5322)`. References to undefined/unset attributes evaluate to the empty string.

Once templates are restricted to operate on a set of attributes to prevent side-effects, the similarity to generational grammars leaps out. Attributes are just the terminals (vocabulary) of a generational grammar and templates are productions (rules). This connection yields some interesting and useful results by leveraging work from formal languages. For example, XML document type definitions (DTDs) are essentially context-free grammars [8], therefore, a template with the power of a context-free grammar can generate any XML document definable with a DTD. It is satisfying to note that a very simple template engine emulating a push down automaton can generate XML documents and by design, as shown in section 8, enforces strict separation of model and view.

This section defines restricted classes of templates corresponding to the Chomsky type 1..3 generational grammars [1]: context-sensitive, context-free, and regular. Let's start with the weakest template class, type 3 regular grammars [16].

DEFINITION 2. *A regular template is restricted to 0, 1, or 2 symbols. The first symbol is either a literal t or an attribute reference a. The second symbol, if present, must be a reference to a regular template. A regular template may be the empty string, ϵ. In other words, templates are of the form t, a, $t\tau$, $a\tau$, or ϵ where τ is a template reference action. Attribute and template references are side-effect free.*

THEOREM 2. *Regular templates generate the regular languages.*

PROOF. Mirrors proof of Theorem 3 for context-free templates; just restrict the derivation tree shape to have rule references and corresponding template references only at the right edge of subtrees. □

The more common regular expression equivalent provides an easier way to think about regular templates. Templates may iterate over a set of attributes and literals: a_i and t_i. In practice, templates loop over a multi-valued attribute to display lists. The surprising fact is that these minimal templates can do a lot. For example, the following example, in the pseudo-regular expression notation of a fictional template engine, generates a `<br>` separated list of users:

```
( $names "<br>" )+
```

where the `(...)+` "one or more" operator would know to walk through the multiple values of the `names` attribute. Most template engines would actually use a FOR-loop type structure like the following Velocity [19] template:

```
#foreach( $n in $names )
$n<br>
#end
```

To apply a template to each name in the list, however, requires that the programmer reference another template in a loop in order to apply the template to the list elements. That is a template must be invoked at a non-right edge position. Imagine a designer wants to factor out a list item template called `item`:

```
<li>$attr$</li>
```

where `$attr$` is a reference to the attribute to which `item` is being applied (`$attr$` analogous to the `this` pointer in object-oriented methods). A template can then apply the `item` sub-template to each attribute in `names` with the following `StringTemplate` notation:

```
<ul>
$names:item()$
</ul>
```

While one can express this particular output language using a regular template, in general, there are output languages that regular templates cannot generate, since we know from formal languages [16] that there are languages that are inherently nonregular. A template that supports general references to other templates corresponds to the context-free generational grammars.

DEFINITION 3. *A* context-free template *is a template whose actions are limited to referencing attributes, a_i and templates. A context-free template may be empty. Actions are side-effect free.*

THEOREM 3. *Context-free templates generate the context-free languages.*

PROOF. (*sketch*) One can show equivalence of a context-free grammar's derivation tree for any sentence to a nested template tree structure. The derivation tree for a context-free grammar has terminal references at the leaves and nonterminals at the subtree roots. A subtree corresponds to a rule application and has children that are either terminals or other nonterminal subtrees. A template is a list of literals, attribute references, and template references. By equating a template's attributes and literals to terminals and template references to nonterminals, one sees that there is always an equivalent template for a derivation tree and vice versa. If a context-free template is equivalent to a context-free grammar, the template generates the context-free languages. □

To get an idea of the generational power of context-free grammars, recall that parsers for most programming languages are implemented via grammars accepted by LL- and LR-based parser generators, which are subsets of the context-free grammars [17]. Therefore, a context-free template is sufficient to generate output conforming to the syntax of common programming languages.

Of more interest to the reader is that context-free templates can generate any XML file conforming to a DTD, as DTDs are context-free grammars. What about SGML? With its DTD *exceptions*, an SGML DTD may indicate when a named element may appear or not appear [8]. In a template, what if the designer needs to alter output based upon context information? For example, a submenu should only appear if the parent menu is active. Allowing context to dictate when a template element may appear adds a great deal of generational power and brings us to the *context-sensitive template* class analogous to the context-sensitive grammars.

DEFINITION 4. *A* context-sensitive template *is context-free template augmented to allow predicated template application; that is, a template augmented to allow template references or inclusion of subtemplates only in certain grammatical context. Actions are attribute references and template references. Predicates operate on a_i and the template tree structure itself. Actions and predicates are side-effect free.*

By limiting predicates to operations on a_i and surrounding template tree structure, predicates are limited to testing grammatical context. I suspect one could show an exact equivalence to the context-sensitive grammars given the proper computation limitations on actions, but these templates are of theoretical interest only. In practice, it is best to avoid entanglement by performing all computations in the model and pushing results into the template via attributes. Once in the model, computations are unrestricted, therefore, template power moves directly from the context-free class to the unrestricted class in practice. Milton and Fischer [11] claim an analogous result that their predicated $LL(k)$ parsers are Turing complete because any predicate is Turing complete.

In the next section, I lay out the strict rules of separation and illustrate illegal template constructs. Then in Section 8 I show which template classes inherently enforce separation by design.

7. FORMAL SEPARATION

While programmers value simplicity, they more often value powerful functionality and amazing "one-liners," incurring the cost of complexity. I cite the 29 parameters to Linux's copy program, `cp`, as an example. Existing templates engines burst with features and power. Some [10] have tags built into the view that let programmers send email and most others even provide (indirect) SQL access to a database! These are obvious cases where the view has merged with the model. Beyond the obvious, however, are numerous seemingly-innocuous template features that entangle the view with the model just as effectively.

Due to lack of formal definition of separation, template engines provide functionality that allows programmers to violate separation routinely and, hence, template engines do not solve the problem that drove their development, losing all benefits of separation described above. For example, to support harmless functionality such as "make all negative numbers red," engines support attribute value testing, which leads to egregious violations such as "if user is James and host is wraith". While many template features have both harmless and improper uses, these features will inevitably be used to violate separation either unwittingly or as an expedient.

Most engines provide only, admittedly useful and convenient, organization of dynamic pages at the cost of learning a new programming language or tool. Without strict enforcement of separation, a template engine provides tasty icing on the same old stale cake. Consequently, do not consider an engine's expressivity and power. Consider, rather, how well it enforces separation. My goal over the past five years building sites as well as the template engine that drives them has been to toss out the stale cake, providing sufficient power while ruthlessly enforcing separation even though it is annoying sometimes.

This section lists the rules that ensure separation of view from model and controller, derived from an academic's eye slugging through the development of a large commercial server application. While a programmer or designer may follow these rules using any template engine without relying on enforcement, it requires extreme discipline and exacts a high cognitive load.

Before formalizing my intuition, I kept a few simple rules and tests in mind to evaluate entanglement. Could I reuse this template with a completely different model? Could a designer understand this template? If it looks like a program, it probably is. If the template can modify the model, it's part of the program. If order of execution is important, it's part of the program. If the template has computations or logic dependent on model data, it's part of the program. If types are important, the template is a program.

My approach to defining separation is to first dictate generally which pieces belong in the model and which in the view in order to make them self-contained or *encapsulated*. Then, I define the rules concerning the interaction of model and view that prevent entanglement.

DEFINITION 5. *A model is* encapsulated *if all logic and computations dependent upon data values are done in the model.*

Notice what this does *not* say. The view is not precluded from computing $sin(i)$ for row i in the *template*, for example. Model

encapsulation simply means that the model must contain all computations that are dependent on model data.

DEFINITION 6. *A view is* encapsulated *if all layout and display information is specified in the view.*

There can be nothing in the model that indicates how the data will be displayed by the view.

DEFINITION 7. *Model and view are strictly separated if both are encapsulated and, further, if the view cannot modify the model and the view makes no assumptions about the types of model data.*

This definition is distilled to the essential restrictions preventing entanglement, but in practice, strict separation has widespread implications about what can and cannot be done in a model and template. In more concrete terms, here are the rules implied by strict separation:

1. **the view cannot modify the model either by directly altering model data objects or by invoking methods on the model that cause side-effects**. That is, a template can access data from the model and invoke methods, but such references must be side-effect free. This rule arises partially because data references must be order-insensitive. See Section 7.1.

2. **the view cannot perform computations upon dependent data values** because the computations may change in the future and they should be neatly encapsulated in the model in any case. For example, the view cannot compute book sale prices as "`$price*.90`". To be independent of the model, the view cannot make assumptions about the meaning of data.

3. **the view cannot compare dependent data values**, but can test the properties of data such as presence/absence or length of a multi-valued data value. Tests like
`$bloodPressure<120`
must be moved to the model as doctors like to keep reducing the max systolic pressure on us. Expressions in the view must be replaced with a test for presence of a value simulating a boolean such as
`$bloodPressureOk!=null`
Template output can be conditional on model data and computations, the conditional just has to be computed in the model. Even simple tests that make negative values red should be computed in the model; the right level of abstraction is usually something higher level such as "department x is losing money."

4. **the view cannot make data type assumptions**. Some type assumptions are obvious when the view assumes a data value is a date, for example, but more subtle type assumptions appear: If a template assumes `$userID` is an integer, the programmer cannot change this value to be a non-numeric in the model without breaking the template. This rule forbids array indexing such as `colorCode[$topic]` and `$name[$ID]` The view further cannot call methods with arguments because (statically or dynamically) there is an assumed argument type, unless one could guarantee the model method merely treated them as objects. Besides graphics designers are not programmers; expecting them to invoke methods and know what to pass is unrealistic.

5. **data from the model must not contain display or layout information**. The model cannot pass any display information to the view disguised as data values. This includes not passing the name of a template to apply to other data values.

By *dependent values*, these rules refer to not only direct model data references like `$user` or `$model.getUser()`, but also to local template variables altered by logic dependent on model data. For example,

```
#set interest = 0.08
#if ( $risk < 0.5 )
  #set interest = 0.065
#endif
#set interestRate = interest*100
```

Here, variables `interest` and `interestRate` depend on model data value `$risk`. Dependencies can clearly sneak up on you.

What about URLs? Is it proper to hard code URLs into templates? Strictly speaking, the URL structure of a site belongs to the controller and should not be referenced as literals in templates.

The next section describes how a common strategy for template data access is inherently unsafe, providing hidden "land mines" for the template designer to step on inadvertently.

7.1 Pull Strategy Violates Separation

Consider a simple HTML page where the application needs to pull a list of names from a database and display them; afterwards it prints the number of elements in the list. The following template pulls data from the model:

```
<html>
<body>
Names:
<p>
$model.getNames()
<p>
There were $model.getNumberOfNames() names.
</body>
</html>
```

where `$model.getNames()` invokes a method on the `model` that computes the list of names and pulls it into the template. The `$model.getNumberOfNames()` reference asks the model to compute and return the number of names. Because `getNames()` was invoked first, the list and its length are available and `getNumberOfNames()` can simply return the number. Now, what if the designer wants to have the number of names in the `<title>` of the page? That is, what if `getNumberOfNames()` must be called first? The model has not yet computed the list. At best the template gets an invalid number; at worst, an exception. The only way to remove this order dependency is to make `getNumberOfNames()` call `getNames()` also.

So, this simple data reference move in the template, which a graphics designer would make without thinking, forces a dramatic implementation change in the model. Similarly, changes in the dependency graph caused by model code changes can break previously-correct views. Unless a development team has extremely good testing, which is very difficult for dynamic web sites, changing the model will inevitably and secretly break some site pages.

To avoid data reference order dependencies, use a *push* strategy.

DEFINITION 8. *A template uses the* push *strategy if all data used by the template is computed prior to template evaluation and is available as a set of attributes* a_i.

Attributes are "pushed" into a template like a stream of tokens. In contrast, an unrestricted template may *pull* from a data model via functions.

DEFINITION 9. *A template uses the* pull *strategy if any data used by the template is computed on demand by invoking model functions.*

THEOREM 4. *A safe pull strategy degenerates to push in the worst case.*

PROOF. Model data values, hereafter referred to simply as attributes, often depend on the values of other attributes. These dependencies form a DAG, a directed acyclic graph. The graph is acyclic in order to be well-defined; for example, attribute a_i cannot depend on itself. To preserve correctness (to be *safe*), the view may not request attributes from the model in an order that violates a dependency in the graph.

In the worst case, the topologically sorted attribute dependency DAG has a unique order. That is, the DAG forms a chain where attribute a_i depends on a_{i-1} ($a_{i-1} \rightarrow a_i$) and, hence, indirectly a_n depends on all $a_1..a_{n-1}$:

$a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_{n-1} \rightarrow a_n$

To compute a_n means the model must compute all other attributes to preserve correctness. Because the designer might request a_n first thing in the view, the model must first compute $a_1..a_{n-1}$ and then a_n. If a_n is the first template expression e_0, a_n is also computed before the template generates any output text. Hence, all attributes are computed *a priori*. Pull degenerates to push in the worst case. □

7.2 Entanglement Index

When evaluating template engines, it is useful to rank them according to the level of entanglement allowed by the tool.

DEFINITION 10. *A template engine's* entanglement index *is the number of separation rules that a template may violate. There are 5 separation rules and therefore the index runs from 1..5. The minimum index is 1 because there is no way to prevent attributes from containing display and layout information.*

`StringTemplate` makes it impossible to violate (decidable) separation by design and has the minimum index of 1. While working on a template survey paper, I found that only `HTML::Template` [18] and XMLC [2] have an index of 1. In contrast, the following template engines have entanglement indices greater than 1: Tapestry, WebMacro, Velocity, PTG, UniT, Tea, WebObjects, FreeMarker, ColdFusion, Template Toolkit, and Mason.

FreeMarker has an entanglement index of 5 because it allows arbitrary method calls into the model, violating rules 1 and 4. It allows business logic in the template, violating rule 3. Finally, it allows computations on attributes, violating rule 2.

Struts, a popular web application control framework, has no entanglement index because it is not itself a template engine; it is designed to work with other view systems like XSLT, Velocity, JSP, and JSP Tag Libraries. A Struts-`StringTemplate` duo could prove a useful combination.

Interestingly, there does not seem to be a middle ground. I have concluded that engines are polarized either at index 1 or 5. This implies adding engine features is a "slippery slope" and must be done with care. Once an engine allows a violation, it must violate others. For example, to allow method calls to the model (potentially violating the side-effect free rule), an engine must violate the type independence rule for the arguments.

8. ENFORCING SEPARATION

The rules of separation are strict, far-reaching, and seemingly extremely confining. Two questions immediately come to mind: 1) can we enforce the rules and 2) is the resulting template engine useful?

In principle, an engine can enforce all rules except for one: it is impossible to police the incoming attributes to determine whether a model or controller is passing display information or links into a template. For example, is attribute value `Red` a color or a man's name?

As for the other four separation rules, one could retrofit unrestricted template engines with a combination of static and run-time checks in an attempt to catch violations and warn the programmer, but my intuition is that there are lots of undecidable problems. A better approach is to start with an overly-restricted template and gradually add power ensuring constructs adhere to the separation principle. This section mirrors the evolution path followed by `StringTemplate`.

Start with a text document filled with "holes" where a programmer or designer can stick single or multi-valued read-only data values called attributes. This single assumption, that templates operate purely on attributes, allows us to relate templates to formal language grammars, provides data reference order independence since data references cannot have side-effects, and ensures templates cannot modify the model. By definition, this template allows no computations. Attributes are inert stones dropped in a soup. To use an analogy from chemistry, data should *mix* and not *react* with the surrounding template. A document with attribute references is a regular template.

Factoring a large template into subtemplates is useful and clearly safe, so engines can support includes. A more sophisticated version of an include is a macro invocation because macros have named parameters that fill holes in the included template. The parameters naturally are just another scope of attributes sent into the subtemplate. As a trivial example, consider defining a template called `bold`:

```
<b>$attr$</b>
```

The template can call `bold` like a macro: `bold(attr=name)`. A slight notation change allows "template application" functionality, leading to the equivalent operation: `name:bold()`. If `name` were multi-valued, `bold` would be applied to each element in the list resulting in another list of, altered, attributes like LISP's `map`. This construct eliminates the need for explicit FOR-loops.

Add recursion so that a template can invoke itself directly or indirectly and the template arrives at the context-free class. With just attribute reference, template application, and template recursion, an application can, thus, generate any XML document that has a DTD without violating any rules of separation.

Do we need more power? That is, do we need the power of context-sensitive templates? Context-sensitive templates are useful for tasks such as switching in a logged in or logged out version of a gutter and, more importantly, for handling conditionally included literals (or subtemplates) such as

```
$if(title)$
<h1>$title$</h1>
$endif$
```

The `<h1>` tags should appear only if the `title` attribute is present.

A template can only test the presence or absence of attributes rather than full expressions because tests against attribute values violate separation. These restricted IF constructs definitely pull templates beyond context-free templates into the context-sensitive range, but how far? Surely full expressions are more powerful. Indeed they are, but programmers can always compute whatever they want in the model and set an attribute or leave it empty, simulating a boolean with the same result as if they had computed the expression in the view.

By adding an IF construct that can test results of unrestricted computations in the model, it is likely that the separated model-view duo is as powerful as an unrestricted template. So, in answer

to the second question regarding usefulness, yes: a model-view duo achieves essentially unrestricted power without violating any rules of separation.

8.1 Attribute Rendering

Benjamin Geer [4], the original author of FreeMarker [3], posed a seemingly unresolvable conundrum for an engine enforcing separation. Consider the escape of "<" characters to HTML format, "`<`", exemplifying a common class of rendering operations such as string abbreviation, date formatting, and case conversion. The model may not escape a data string *a priori* and send it to the template without violating separation because "`<`" is clearly display information. The template may not examine an incoming string looking for "<" characters to do the necessary replacements. The template also may not invoke a method on the model that computes escaped strings. How then does one escape strings, a common and necessary operation?

The inescapable fact is that there must be some Java code somewhere that computes HTML escape sequences because the computation cannot be done in the template. Besides, Java already provides heavy support, such as the `java.text` package, for dealing with text, numbers, dates, and so on. Since neither the model nor the controller may contain this computation, the code must exist elsewhere. It turns out that there is a silent partner in the MVC pattern at work already where these types of attribute rendering operations properly belong.

To expose the silent partner, consider that for every integer attribute passed to a template, the template engine implicitly calls the attribute's `toString()` method with our tacit approval. The collection of `toString()` methods of all incoming attribute types embody an aspect [7], perhaps suggesting a more complete design pattern name when referring to HTML page generation: MVCR (*model-view-controller-renderer*) where the *renderer* encapsulates the conversion of objects to displayable character strings.

There is a subtle, but critical distinction between view and renderer. A template is mainly concerned with page and attribute layout plus text properties like color and size. The renderer, on the other hand, is concerned with converting attributes (or, in the case of aggregates, the constituent components) to displayable strings of characters. For example, the renderer decides whether to render negative integers as `(23)` instead of `-23`, but the template decides where on the page the number goes, how big it is, what color it is, whether it is a link, etc...

To solve the HTML character escape paradox, then, the controller can simply wrap string attributes with an object whose `toString()` method escapes special characters. When the template renders to text, strings will be escaped properly. For other attributes, wrappers might delegate to standard Java libraries. The controller acts like a station in an assembly line from the model to the view that adds a paint job to certain objects. The `toString()` methods in standard types and in all wrapper objects used by an application embody the renderer. It makes sense to encapsulate wrapper objects in, say, an enclosing `HTMLRenderer` class. The fact that this class can be reused with just about any web application lends credence to the fact that the rendering code is not part of a particular application's model.

Each template may have its own renderer. The controller can assign different renderers in response to page requests from different locales. In addition, a view-renderer pair could automatically perform routine operations like escaping strings, thus, often relieving the controller from having to specifically wrap attributes.

Asking an attribute to render itself fits nicely within our object-oriented mindset and establishes a boundary that cleaves the conventional view component into both view and renderer, yielding the MVCR pattern. The renderer neatly solves Geer's conundrum and comparable problems. Just as a template may access an unrestricted model computation through the decoupling mechanism of testing the presence/absence of an attribute, templates may access unrestricted rendering computations performed in the renderer via the narrow conduit of the `toString()` method without violating model-view separation.

Some readers will argue that the MVCR pattern is merely a cleverly worded disguise, opening a potential loophole for violations of model-view separation. A renderer will contain code using string literals containing display information and, worse, a renderer could potentially contain code that modified an application's model. First, recognize that there is no way around allowing objects to render themselves with the `toString()` method. Either integers get converted from 32-bit binary words to strings of digit characters or servers cannot generate web pages, using templates or otherwise. Preventing abuse of this mechanism is undecidable and engines already take a hit for this vulnerability, accounted for in the minimum entanglement index of 1; though, one should consider factoring out HTML literals even in the renderer into fine-grained templates to completely avoid HTML in code. Second, formalizing the renderer and encapsulating rendering operations into an application independent service allows the renderer for a given target language such as HTML to be supplied with an engine. Because the renderer will not be part of a specific application, it cannot constitute an entanglement of an application's model and view. Technically a programmer could alter the HTML renderer or attach a "malicious" renderer that altered a specific model's data, but that risk already exists: someone could easily alter the source for a strict engine to allow violations.

8.2 Enforcement vs Encouragement

Programmers crave flexibility. Whether they sacrifice a little flexibility in order to enforce valuable principles depends on their experience and philosophy derived from that experience. If they have learned the hard way that sometimes even geniuses make mistakes or do the expedient thing, they will fight to make mechanisms work correctly by design. Remember that Murphy's original law, "if it can be done wrong, it *will* be done wrong," arose from someone installing accelerometers the wrong way during a rocket sled test, unnecessarily stressing the test pilot's body. The reader may also recall computers in the 1980's where the keyboard and modem connectors were identical; many people fried their keyboard by accident.

It is wiser to enforce rather than merely encourage good behavior even if the imposed restrictions chafe a little. The main reason programmers have moved to new template engines in the first place is to avoid the entanglements found with original systems like JSP and the ilk. I submit that re-implementing JSP and adding a comment in the manual encouraging un-JSP like behavior is insufficient. My philosophy with `StringTemplate` has been to build outwards from safety, adding functionality while adhering strictly to my principles.

9. STRINGTEMPLATE

`StringTemplate` [13] is a template library carefully designed by myself and Tom Burns (CEO, jGuru.com) over many years of experience building commercial sites including jGuru.com, knowspam.net, and antlr.org. `StringTemplate` evolved from a simple "document with holes" to a sophisticated template engine that, much to my surprise, has a distinctly functional programming flavor. While functional programming languages have never been my

forte, StringTemplate's language evolved naturally according to my needs: side effect-free expressions, expression order independence, template application to a list of data objects, nested (that is, recursive) application of templates, and simplicity to make templates accessible to my graphics designer. StringTemplate's design is optimized for enforcement of separation not for Turing completeness nor amazingly-expressive "one-liners". The simple name StringTemplate reflects my minimalist approach (its jar is about 120k with source and binaries).

StringTemplate enforces separation of model and view with the exceptions that it cannot enforce the undecidable problem of passing display information to the view via attributes. We repeatedly verified that a nonprogrammer graphics designer can operate independently from the site coders. Once our designer knew what templates she could fill in and what attributes she could reference, she was able to build multiple site looks (branding for customers) totally independently.

In addition to the power of its template expression language, StringTemplate promotes heavy site component reuse, supports multiple site "skins" (a skin is simply a group of templates), provides template inheritance between skins so the designer can build new site skins as they differ from existing skins, allows site designs to be updated on a live server, and alleviates the need for controller page code to manually generate HTML or subtemplates.

When discussing template engines, programmers will ask how a particular engine handles some interesting output tasks. The next sections demonstrate how StringTemplate expresses filling a table, alternating the row colors in a table, and generating hierarchical menus. The examples illustrate the two crucial features that alleviate the need for looping constructs and just about every other common language construct: template application to a multi-valued attribute (like LISP's map) and template recursion.

9.1 Fill a table example

Imagine we have objects of type User that we will pull from a simulated database:

```
public class User {
    String name;
    int age;
    ...
}
static User[] list = new User[] {
    new User("Boris", 39),
    ...
};
```

Push the user list as attribute users into a template, say st, with the following Java code:

```
st.setAttribute("users", list);
```

To generate a table like this:

```
<table border=1>
<tr><td>Boris</td><td>39</td></tr>
<tr><td>Natasha</td><td>31</td></tr>
...
</table>
```

apply an anonymous template to each user object:

```
<table border=1>
$users:{
<tr><td>$attr.name$</td><td>$attr.age$</td></tr>
}$
</table>
```

where attr is the default attribute name that varies automatically as StringTemplate iterates over users. attr.name accesses the name property of the incoming User object via the getName() method. The anonymous row template can also be factored out into a template called row:

```
<tr><td>$attr.name$</td><td>$attr.age$</td></tr>
```

reducing the main template to the more readable:

```
<table border=1>
$users:row()$
</table>
```

The result of a template application is another list. Templates can be applied to the results of other template applications. In the following example, each element of a list is bolded and then made a <ul> bullet item.

```
<ul>
$aList:{<b>$attr$</b>}:{<li>$attr$</li>}$
</ul>
```

9.2 Alternating table row colors

Graphics designers often format lists with alternating element background color:

```
<table border=1>
<tr><td>Boris</td><td>39</td></tr>
<tr><td bgcolor=#F6F6F6>Natasha</td><td>31</td></tr>
<tr><td>Jorge</td><td>25</td></tr>
...
</table>
```

Rather than resort to a FOR-loop with a nested IF, just provide a comma-separated list of templates to apply round-robin to the list elements.

```
<table border=1>
$users:rowBlue(), rowGreen()$
</table>
```

9.3 Hierarchical menus

Why would anyone ever need recursion in a template? Consider doing hierarchical menus. Without recursion, the template needs complicated nested FOR loops and IF structures. A hierarchy is a recursive structure and cries out for recursion to walk the tree. An engine without recursion will simply not deal naturally with recursive data structures. For our purposes, assume each item in a menu has a title, a url, a possible list of submenu items, and whether a particular submenu is active:

```
public class TreeMenuItem {
    String title;
    String url;
    boolean active = false;
    List submenu;
    ...
}
```

In the gutter of the overall page template, apply template menuItem to the main list of choices:

```
$choices:menuItem()$
```

A non-recursive template that generates only the top layer as a list of links looks like this:

```
<a href=$attr.url$>$attr.title$</a>
```

Output might look like:

```
<a href=/>home</a>
<a href=/news>news</a>
...
```

To handle submenus, have the template iterate (tail-recursively) over the list of submenu items if that menu is active.

```
<a href=$attr.url$>$attr.title$</a>
$if(attr.active)$
$attr.submenu:menuItem()$
$endif$
```

While the table filling examples use only context-free template constructs, this recursive menu example requires context-sensitivity in order to show only active submenus. Also note that no display information needs to be passed in as an attribute.

Programmers often ask how to encode a complicated algorithm in a template. The key lesson shown in this hierarchical menu example is that messy computations and control-flow constructs are easily avoided in a template through judicious choice of data-structure, mirroring the relationship, say, between a state machine's trivial engine and its highly-interconnected state network. For example, language parser state machine construction is not easy, but walking the resulting states to actually parse sentences is very easy. Indeed the code to construct states dwarfs the tiny parsing engine. Parser construction is analogous to the controller-model pair organizing a data structure or structures; the template engine is analogous to the parsing engine.

10. CONCLUSIONS

Strictly separating model and view is an extremely desirable goal because it increases flexibility, reduces maintenance costs, and allows coders and designers to work in parallel. Template engines evolved in response to previous technologies such as JSP and the ilk that entangle model and view. Unfortunately, engines merely encourage separation while still supporting constructs that allow programmers to violate separation. Programmers will often use these constructs as an expedient or because of lack of experience, thus, current engines provide unclear advantages over JSP (although most engines provide very handy frameworks besides templates for building web applications).

Existing engines do not enforce separation because there was no formal definition of separation nor list of rules to guide them. It is also against a programmer's nature to limit functionality, fearing that enforcing separation implies a critically weak engine. I also wonder if engine designers get caught up having fun implementing languages without regard to overall principles.

I have shown that we can, in fact, build a template engine that strictly enforces all decidable rules of separation by design. `StringTemplate` is sufficiently powerful empirically to generate numerous complicated sites such as jGuru.com and, moreover, the pages are simple and sufficiently unentangled to be built by a graphics designer not a programmer.

By showing the relationship between restricted templates and generational grammars, I conclude that restricted templates can generate interesting classes of languages such as those described by XML DTDs and beyond into the context-sensitive and unrestricted languages.

Given a clear definition of model-view separation, template engine designers may no longer claim enforcement of separation without an entanglement index of 1. More importantly, given theoretical arguments and empirical evidence, engines no longer have an excuse to support entanglement, fearing weak generational power.

I am currently investigating the use of template engines to support powerful and reusable language translators for the next generation of the ANTLR parser generator. `StringTemplate` is released under the BSD license at `http://www.stringtemplate.org`

11. ACKNOWLEDGEMENTS

I would like thank to John Witchel for motivating me to formalize my thoughts regarding MVC separation and Monty Zukowski for planting the *template* "meme" in my head back in 1995. I thank Chris Brooks and David Galles for their detailed reviews. Matthew Ford provided very valuable feedback on both the ideas contained herein and the source code for my engine. I thank Benjamin Geer for his review work and for pointing out flaws in my arguments.

12. REFERENCES

[1] N. Chomsky. *Aspects of the Theory of Syntax*. MIT Press, 1965.

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

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

[4] B. Geer. Private communications.

[5] J. Hunter. The problems with JSP. http://www.servlets.com/soapbox/problems-jsp.html.

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

[7] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[8] P. Kilpelainen and D. Wood. SGML and XML document grammars and exceptions. *Information and Computation*, 169(2):230–251, 2001.

[9] G. Krasner and S. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[10] Macromedia. Coldfusion. http://www.macromedia.com/software/coldfusion.

[11] D. R. Milton and C. N. Fischer. LL(k) parsing for attributed grammars. In *International Conference on Automata, Languages, and Programming*, pages 422–430, 1979.

[12] Separating C from V in MVC. http://mathforum.org/epigone/modperl/jilgygland, 2002.

[13] T. Parr. StringTemplate documentation. http://www.antlr.org/stringtemplate/index.tml, September 2003.

[14] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[15] Servlets. http://java.sun.com/products/servlet/.

[16] S. Sippu and E. Soisalon-Soininen. *Parsing Theory I*. Springer-Verlag, 1988.

[17] S. Sippu and E. Soisalon-Soininen. *Parsing Theory II*. Springer-Verlag, 1988.

[18] S. Tregar. HTML::Template. http://html-template.sourceforge.net.

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

[20] XSL style sheets. http://www.w3.org/Style/XSL/.