LEIDEN UNIVERSITY MEDICAL CENTER

The Template Attribute Language

Jeroen F. J. Laros

Leiden Genome Technology Center

Department of Human Genetics

Center for Human and Clinical Genetics

Introduction

The *Template Attribute Language* templating language used to generate dynamic HTML and XML pages.

- Simplify the collaboration between programmers and designers.
- Works by embedding TAL statements inside *valid* HTML or XML tags.

Introduction

The *Template Attribute Language* templating language used to generate dynamic HTML and XML pages.

- Simplify the collaboration between programmers and designers.
- Works by embedding TAL statements inside *valid* HTML or XML tags.

Interesting attributes:

- Conditional output.
  - Show or hide parts of a page.
- Repeating (looping).
  - Tables.
- Macros.
  - Code reuse.
  - Nesting.

Introduction

Allows for more *modularity*.

- Separating the interface from the implementation.
  - Allow for separate development of the interface.
  - Changes can be made to the interface without touching the code.
  - Changes can be made to the code without affecting the interface.

Introduction

Allows for more *modularity*.

- Separating the interface from the implementation.
  - Allow for separate development of the interface.
  - Changes can be made to the interface without touching the code.
  - Changes can be made to the code without affecting the interface.

Because the interface is separated, *multiple* interfaces can be easily added.

- Interfaces don't interfere.
  - The core is not written with one interface in mind.
- No need to keep track everywhere in the code which interface is used.
- Communicate via internal variables.

Generating a table

Generate a table from the following matrix:

```
1   exonInfo = [
2       [  5809,  5860,   '1',  '52' ],
3       [  6758,  6874,  '53', '169' ]
4   ]
```

Listing 1: Example input

Generating a table

Generate a table from the following matrix:

```
1   exonInfo = [
2       [  5809,  5860,   '1',  '52' ],
3       [  6758,  6874,  '53', '169' ]
4   ]
```

Listing 1: Example input

The output should be something like this:

Number	Start (g.)	Stop (g.)	Start (c.)	Stop (c.)
1	5809	5860	1	52
2	6758	6874	53	169

Table 1: Example output

Common implementations

This is a frequently used solution.

```
1   print "<table class = \"raTable\"><tr>" \
2         "<td>Number</td><td>Start (g.)</td>" \
3         "<td>Stop (g.)</td><td>Start (c.)</td>" \
4         "<td>Stop (c.)</td></tr>"
5   for i in range(len(exonInfo)) :
6       print "<tr><td>" + str(i + 1) + "</td>"
7       for j in exonInfo[i] :
8           print "<td>" + str(j) + "</td>"
9       print "</tr>"
10  #for
11  print "</table>"
```

Listing 2: Common implementation

Common implementations

This is a frequently used solution.

```
1   print "<table class = \"raTable\"><tr>" \
2         "<td>Number</td><td>Start (g.)</td>" \
3         "<td>Stop (g.)</td><td>Start (c.)</td>" \
4         "<td>Stop (c.)</td></tr>"
5   for i in range(len(exonInfo)) :
6       print "<tr><td>" + str(i + 1) + "</td>"
7       for j in exonInfo[i] :
8           print "<td>" + str(j) + "</td>"
9       print "</tr>"
10  #for
11  print "</table>"
```

Listing 2: Common implementation

Problems with this implementation:

- Parts of the HTML code has to be escaped.
- The output is a mess.

Common implementations

The output works, but has no indentation.

```
<table class = "raTable"><tr><td>Number</td>
<td>Start (g.)</td><td>Stop (g.)</td>
<td>Start (c.)</td><td>Stop (c.)</td></tr>
<tr><td>1</td>
<td>5809</td>
<td>5860</td>
<td>1</td>
<td>52</td>
</tr>
<tr><td>2</td>
<td>6758</td>
<td>6874</td>
<td>53</td>
<td>169</td>
</tr>
</table>
```

Listing 3: Output of the code in Listing 2

Common implementations

This is a slightly better approach.

```
1   print "<table class = \"raTable\">"
2   print "  <tr>"
3   print "    <td>Number</td>"
4   print "    <td>Start (g.)</td>"
5   print "    <td>Stop (g.)</td>"
6   print "    <td>Start (c.)</td>"
7   print "    <td>Stop (c.)</td>"
8   print "  </tr>"
9   for i in range(len(exonInfo)) :
10      print "  <tr>"
11      print "    <td>" + str(i + 1) + "</td>"
12      print "    ",
13      for j in exonInfo[i] :
14          print "<td>" + str(j) + "</td>",
15      print "\n  </tr>"
16  #for
17  print "</table>"
```

Listing 4: Fix indentation

Common implementations

The output looks fine now.

```
1   <table class = "raTable">
2     <tr>
3       <td>Number</td>
4       <td>Start (g.)</td>
5       <td>Stop (g.)</td>
6       <td>Start (c.)</td>
7       <td>Stop (c.)</td>
8     </tr>
9     <tr>
10      <td>1</td>
11      <td>5809</td><td>5860</td><td>1</td><td>52</td>
12    </tr>
13    <tr>
14      <td>2</td>
15      <td>6758</td><td>6874</td><td>53</td><td>169</td>
16    </tr>
17  </table>
```

Listing 5: Output of the code in Listing 4

Common implementations

```
1   <table class = "raTable">
2     <tr>
3       <td>Number</td>
4       <td>Start (g.)</td>
5       <td>Stop (g.)</td>
6       <td>Start (c.)</td>
7       <td>Stop (c.)</td>
8     </tr>
9     <?
10      print("\r");
11      for ($i = 0; $i < count($exonInfo); $i++) {
12        print("  <tr>\n");
13        print("    <td>" . ($i + 1) . "</td>\n    ");
14        for ($j = 0; $j < count($exonInfo[$i]); $j++) {
15          print("<td>" . $exonInfo[$i][$j] . "</td>");
16        }//for
17        print("\n  </tr>\n");
18      }//for
19    ?>
20  </table>
```

Listing 6: PHP example

Doing the same with TAL

A TAL template is pure HTML or XML.

```
1   <table class = "raTable">
2     <tr>
3       <td>Number</td>
4       <td>Start (g.)</td>
5       <td>Stop (g.)</td>
6       <td>Start (c.)</td>
7       <td>Stop (c.)</td>
8     </tr>
9     <tr tal:repeat = "i exonInfo">
10      <td tal:content = "repeat/i/number"></td>
11      <td tal:repeat = "j i" tal:content = "j"></td>
12    </tr>
13  </table>
```

Listing 7: TAL example

Doing the same with TAL

A TAL template is pure HTML or XML.

```
1   <table class = "raTable">
2     <tr>
3       <td>Number</td>
4       <td>Start (g.)</td>
5       <td>Stop (g.)</td>
6       <td>Start (c.)</td>
7       <td>Stop (c.)</td>
8     </tr>
9     <tr tal:repeat = "i exonInfo">
10      <td tal:content = "repeat/i/number"></td>
11      <td tal:repeat = "j i" tal:content = "j"></td>
12    </tr>
13  </table>
```

Listing 7: TAL example

- This template can be viewed with a browser.
- Modifications to the template are easy.

Doing the same with TAL

Here we see an example of how to compile a template.

```
1   from simpletal import simpleTALES
2   from simpletal import simpleTAL
3   from cStringIO import StringIO
4
5   context = simpleTALES.Context()
6   context.addGlobal("exonInfo", exonInfo)
7
8   handle = open("TAL_example.html", 'r')
9   template = simpleTAL.compileHTMLTemplate(handle)
10  handle.close()
11
12  io = StringIO()
13  template.expand(context, io)
14
15  print io.getvalue()
```

Listing 8: Compiling the template

Doing the same with TAL

Here we see an example of how to compile a template.

```
1   from simpletal import simpleTALES
2   from simpletal import simpleTAL
3   from cStringIO import StringIO
4
5   context = simpleTALES.Context()
6   context.addGlobal("exonInfo", exonInfo)
7
8   handle = open("TAL_example.html", 'r')
9   template = simpleTAL.compileHTMLTemplate(handle)
10  handle.close()
11
12  io = StringIO()
13  template.expand(context, io)
14
15  print io.getvalue()
```

Listing 8: Compiling the template

We write this only once...

Doing the same with TAL

We put the functionality of Listing 8 in a function.

```
1   args = {
2       "exonInfo" : exonInfo,
3       "version"  : 2.0,
4       "welcome"  : "Hello"
5   }
6
7   processTal("TAL_example.html", args)
```

Listing 9: Template compiling function

Doing the same with TAL

We put the functionality of Listing 8 in a function.

```
1   args = {
2       "exonInfo" : exonInfo,
3       "version"  : 2.0,
4       "welcome"  : "Hello"
5   }
6
7   processTal("TAL_example.html", args)
```

Listing 9: Template compiling function

For each page, we can now make a template, and call the processTal() function with the associated parameters.

Demonstration

http://www.mutalyzer.nl/

Conclusions

Abstract the interface from your code.

- Write your code independent from the interface.
- And the other way around.
  - No mixed languages.
  - Opens the road for other interfaces.
  - No interference with other interfaces.

Conclusions

Abstract the interface from your code.

- Write your code independent from the interface.
- And the other way around.
  - No mixed languages.
  - Opens the road for other interfaces.
  - No interference with other interfaces.

Divide expertise.

- Allow other people to write / modify the interface.

Conclusions

Abstract the interface from your code.

- Write your code independent from the interface.
- And the other way around.
  - No mixed languages.
  - Opens the road for other interfaces.
  - No interference with other interfaces.

Divide expertise.

- Allow other people to write / modify the interface.
- Work with templates.
  - Modify your template easily.
  - Templates can be viewed without compilation or a web server.
  - Templates can be nested.
    - A menu structure as a template, with a template inside.

Questions?

Some implementations:

Java – **http://christophermrossi.com/jpt/**
Perl – **http://search.cpan.org/dist/Petal/**
PHP – **http://phptal.org/**
Python – **http://www.owlfish.com/**