Html template system using java annotations

Peter Speck

Supervisor: Bo Holst-Christensen
Computer Science, module 1, 2007
Roskilde University

Abstract

The problems that motivate this project are to (1) solve the lack of separation between html templates and java code when using existing template systems (e.g. embedded language or macros), to (2) solve the lack of scoped declaration of macros and java variables inside template loops, and (3) to solve the lack of validation of template macro definitions at compile time to help finding bugs before the web applications are deployed.

Annotations are used as metadata format for specification of the dynamic parts of the html template and 74 annotations are designed for the template system. The html templates don't contain embedded macros and are fully valid html files without any special tags or attributes, and well suited for editing in visual editors. An annotation-based template compiler and runtime are implemented and tested.

Strict type checking is performed by the template compiler when it processes the annotations. The compiler generates java source files which enables IDEs to show usages of template macro implementations, i.e. servlet methods and fields.

The system provides template looping constructs by using `Iterator`s and automatically instantiating the appropriate inner class for each object returned by the iterator. The inner classes provide scoping for temporary variables used by the loops to avoid polluting the global servlet scope with temporary variables.

Tests of annotations as specification format show that the new system provides very concise specifications for common web implementation design patterns and makes servlets easier to maintain.

Dansk abstrakt

Projektets mål er (1) at løse problemet med manglende separation mellem html-skabeloner og java-kode ved anvendelse af eksisterende systemer (f.eks. indlejrede sprog eller makroer), (2) at løse problemet med at definitioner af makroer og java variabler ikke er afgrænset i skabelonløkker, og (3) at løse problemet med manglende validering af skabelonmakroer på oversættertidspunktet, så fejl ikke først findes, når web-applikationen sættes i produktion.

Java-annotationer anvendes som meta-data format til specifikation af de dynamiske dele af html-skabelonen. Der designes 74 annotationer til det nye system, og der implementeres og testes en oversætter samt runtime hertil. Skabelonerne er fuldt valide html-dokumenter uden makro-referencer, specielle tags eller attributter og er derfor specielt velegnet til visuelle redigeringsværktøjer.

Makroerne type-checkes ved oversættelse af skabelonerne, og der genereres java kildekode, som gør det muligt for IDE-værktøjer at vise hvilke makro-definitioner, som anvendes af skabelonerne og hvilke, der ikke (længere) anvendes.

Tests af annotationer som metadata format viser, at det nye skabelonsystem giver mulighed for korte og præcise specifikationer af typiske web-implementationers 'design patterns' og at det gør det nemmere at vedligeholde servlets.

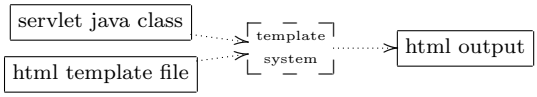
Contents

Chapter 1

Introduction

This project is about an html template system for java. The difference to the exuberance of existing systems is the use of java annotations to specify variable items in the template, the complete separation of the template from the java code and the use of a compiler to improve maintainability by supporting strict type checking.

The basic structure of information flow in a typical html template system is illustrated below. The system uses input from the template and from the servlet to generate the output. Some template systems have a more complex structure due to e.g. ability to call back into the servlet, embedded language in the template, or a compilation stage.



1.1 Motivation

The compiler implements the 4th template system which I have developed and thus builds on a history of experience and convictions. The major motivation behind the first 2 systems (one written in perl and the other in java) is to get the verbose html out of the program code as it obstructed the code clarity.

The first system uses macro-based replacements which supports optional arguments. The second system uses embedded program code for substitutions, assignments and conditionals. The two systems make it easier to create small and very simple pages and even though they covered the needs at the time, they can't handle the demands of modern web applications. One major problem is that webdesign is no longer created by programmers but by dedicated web designers who seldom have any programming background and therefore have difficulty editing templates which contain embedded programming code. Another major problem is that they don't scale in web application complexity as the embedded language is dynamic with variables defined in #include'd files and by a complex definition of variable and callback functions created by the servlets inheritance hierarchy. For complex sites, it is very difficult to determine if a variable or callback is used by anything and which role it has for the templates.

The 3rd system tries to solve these problems and comes pretty close: The html template is a completely valid html file with no special tags or attributes and it can contain mockup data. However, the specification of dynamic parts (i.e. the 'api' between the template and java) is made in xml, thus creating another file which has to be modified whenever

functionality is added/removed/changed in the template and the servlet. This makes it easy to build new pages, however, maintaining pages are more problematic as it is a manual process to determine which specifications in the xml are affected when a java method is deleted or parts of the html template is removed/changed.

1.2 Objectives

The aim of the 4th system—this project—is to create an improved system which doesn't have the 3rd systems' maintenance burden: The dynamic parts of the html templates are specified using java annotations instead of external xml files.

This report seeks to answer the following two questions:

> Are java annotations a suitable replacement for specifying the variable parts of the html templates?
>
> Which advantages and disadvantages does such a system have compared to the xml-based system?

1.3 Method

The structure of this report closely follows the method for answering the questions.

Chapter 2 describes the previous template systems to extract the experience from them. The purpose is to provide a requirement specification for the design of the 4th system. The rest of the report is split into two parts: the first part designs the annotations (chapter 4) and tests the design (chapter 5) by using it to implement common design patterns in template usage. The second part describes the implementation of the compiler (chapter 6) and tests the implementation (chapter 7). Finally, chapter 8 concludes on the main questions posed above and provides suggestions for further work to improve the template system.

Although the structure of the report makes it seem like the system was developed using the waterfall model, the complete history of the template systems points to the iterative model employed at non-static requirements since the requirements changed as a consequence of being able to build even bigger and more complex systems. If a waterfall model had worked, the 3rd system would have been perfect from the start and not prompting the urge to create an improved system.

1.4 Scope

Some interesting areas have to be considered being outside the scope of this report as space and time for this project are limited.

The reader is assumed to have a basic knowledge of html and template systems, and to be very familiar with java. A terse introduction to annotations is provided in chapter 3, however, it should be considered a summarization rather than an introduction.

Furthermore, code snippets are presented as illustrative material and not as working or complete code. Access modifiers, throws clauses, generic<> type markers and all non-essential lines of code have been omitted. The report doesn't contain Javadoc api documentation as the aim is to keep a more high-level view of the system. It tries to describe the forest instead of a number of trees: The forest is a different concept than the sum of detailed descriptions of each tree. However, the source for the system is listed in the appendix but

without any low-level documentation (such as Javadoc) or comments—listed as an item in "further work"—which is needed for releasing the template system to the public.

Only the implementation of the compiler is described, whereas the build system and the two 3rd party libraries (JTidy and JUnit) are omitted. The runtime system—including dynamically recompiling templates—is not described beyond what's needed to describe the output of the compiler.

1.5 Summary

A template compiler and a runtime for processing the compiled templates is implemented and tested. The sources are provided in the appendix.

This report provides a requirement specification based on previous work, the design of annotations for describing the variable parts of an html template, a high-level description of the implementation, the testing of it, and a conclusion based on the two questions which this report seeks to answer.

Chapter 2

Previous work and solutions

The purpose of this chapter is to provide a set of requirements for the template system by describing previous template systems.

2.1 The early days

Suppose we had the task of generating the following snippet of html where the logo url and the name of the user are variable and the rest is static:

```
L1  <div>
        <img src='/images/ruc.jpg'><br>
        You're logged in as Valentin Silvestrov.
    </div>
```

Early html pages were rather simple and as perl was very often used for creating web pages, all of the above html code was often included directly in the perl program, .e.g.

```
L2  print <<"END";
    <div>
        <img src='$logourl'><br>
        You're logged in as $firstname $surname.
    </div>
    END
```

More complex output was often made by some less template-ish code:

```
L3  print "<div>\n<img src='", $logo->url(), "'><br>\n",
      "You're logged in as ", $user->firstname(), " ", $user->surname(), ".\n",
        "</div>\n";
```

The need for templates arose when html design became more voluminous and advanced: The amount of html began to obscure the real code and the design was often transferred to a web designer who needed to be able to edit the html without being required to have the skill to edit perl scripts.

This was often solved by extracting most html snippets into external files and by using magic tags (e.g. @MACRONAME@) to mark text that should be replaced at runtime:

L4
```
<div>
    <img src='@LOGOURL@'><br>
    You're logged in as  @FIRSTNAME@   @SURNAME@
</div>
```

2.2 Contemporary macro-based systems

The 'collection of snippets' method evolved into having the full html page in one file (rather than a collection of snippets) as the visual html editors never handled partial html files well. This created the need for more complex templates as sections of the file sometimes needed to be omitted or needed to be reused several times when presenting dynamic lists of items, e.g. search results.

To reduce the amount of coding needed for handling variable data beyond simple substitutions most template systems added the ability to include code inside the template. This is the opposite case of the initial method which had html inside the program. To be able to use the visual editors with such templates, the editors had to be extended to handle such embeddings of non-html inside the html files.

The following snippet shows how this is done in JSP.[1] Instead of using simple variable substitution, it provides full java expressions using `<%= ...%>` tags and at runtime evaluates the contents and prints the resulting value in place of the tag:

L5
```
<div>
    <img src='<%= image.getUrl() %>'><br>
    You're logged in as <%= user.firstname() + " " + user.surname() %>.
</div>
```

To increase performance, many template systems compile the templates so they don't have to be parsed at runtime. The above snippet can be translated into the java statements below, assuming `pw` is an instance of ava.io.PrintWriter:

L6
```
pw.print("<div>\n\t<img src='");
pw.print(image.getUrl());
pw.print("'><br>\n\tYou're logged in as ");
pw.print(user.firstname() + " " + user.surname());
pw.print("\n</div>\n");
```

2.2.1 Drawbacks

- The special `<%= ...%>` tags are syntax errors when the html file is validated against the html standard. Thus the html validator and html tool must be able to recognize these tags, thereby restricting the set of usable tools.
- The html file contains JSP-tags instead of real data and it is very difficult to provide placeholder data which enable the designer to judge how the page looks when it is processed with real data.

The last point is important for web designers. When a browser displays the JSP Document, it will not be able to display the logo because "`<%= image.getUrl() %>`" is a non-existing file name. As the second `<%= ...%>` tag isn't a known/valid html tag, most browsers will ignore it and only display the text "You're logged in as ."

This has the effect of turning the html file into a piece of code: It is only possible to visually inspect the layout when the template has been 'executed' and the resulting html is

[1] Java Server Pages, http://java.sun.com/products/jsp/

displayed by a browser, thus damaging the most important aspect of a visual design tool: the ability to see the design visually.

This often results in the fact that the designer is not using the templates but a set of mock-up files which the programmer then has to translate into a template by adding the JSP tags. Whenever the designer changes the design, the designer either has to work in blind by editing the JSP file or by changing the mock-up and send it to the programmer who then has to figure out which modifications have been made since the last version and then reapply those to the JSP file—an error prone and tedious process.

2.3 DOM-based template systems

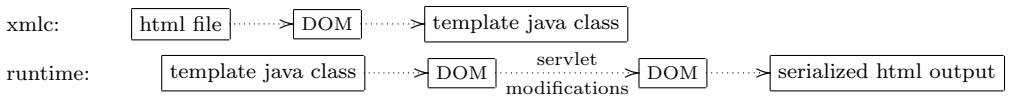
Some systems use a model based on DOM[2] instead of treating the template as a simple text file with macro-based substitutions. The systems have one major advantage compared to the macro-based systems: The html file is editable by most visual html editors as the templates don't contain syntax errors due to non-standard tags.

Another advantage is a much larger flexibility for writing libraries which modify documents (e.g. to make all URLs relative) as the systems provide access to the template thru an api, whereas macro-based systems either have to put macros everywhere something needs to be changed at runtime or by post-processing the generated html as a text string.

As the html template doesn't contain special tags, the primary difference between the systems is the provided api for modifying the template instance, e.g. how to populate a table with results from a database query. Some systems use the DOM-api at runtime, whereas others use a specialized api (and implementation) for performance reasons.

2.3.1 Enhydra xmlc

Enhydra xmlc[3] is the first well-known template system that is DOM-based. It uses a DOM at runtime. The DOM-api provides huge flexibility in modifying the document although the api is very low-level. The DOM tree is serialized as a java class. After the java class has created a DOM instance at runtime, the programmer can modify the DOM using a combination of plain DOM methods and convenience functions provided by xmlc for setting the text of elements with an `id` attribute. Finally, the DOM is serialized to a java `String` and sent to the browser:



XMLC has one very serious drawback: the performance. It has to create the gazillion java objects for representing the DOM and it has to convert those gazillion objects into a text representation to transmit the document to the browser. My experience of an Enhydra-based web application[4] is that for most pages much more cpu time was used for the processing of the template than the data retrieval. Furthermore, the access to the subparts of the DOM must be optimized as performance will otherwise suffer when elements are accessed by their `id` attribute as a full recursive search of the document tree is very slow.

[2]Document Object Model, an api for modifying the object tree in XML and html documents. See http://www.w3.org/DOM/

[3]XML Compiler, http://www.enhydra.org/tech/xmlc/

[4]Old version of http://jobfinder.dk/ which was developed for Ingeniøren A/S

Enhydra tries to solve the performance problem by using a specialized DOM implementation which keeps subtrees in a non-expanded format, i.e. serialized text format. However, this optimization only helps significantly when the document contains large subtrees which are not accessed at all, i.e. contains no variable data and don't require any post-processing to fix e.g. javascript links or image width/height tags.

Another drawback is that Enhydra xmlc doesn't support dynamical recompilation of the template, so whenever the template is changed, a xmlc recompilation followed by a javac compilation is required.

2.3.2 Reasonable Server Faces

Reasonable Server Faces[5] (RSF) not only implements the View part of MVC but a web framework which handles the complete request lifecycle, including object persistence in databases. The html template contains non-standard `rsf:id` attributes and it must be well-formed xml.[6] The template will therefore fail strict validation in many html validators. This is not a major problem as most visual html editors ignore unknown attributes and keep them as-is. The template system does not provide direct access to the DOM but uses a specialized interface which only provides modifiability of html elements which have a `rsf:id` attribute. The template api in RSF is focused on supporting the JavaBeans programming model.

Example:

```
L7  <form action="#" rsf:id="basic-form">
        <input type="password" rsf:id="login-password"></input>
    </form>
```

To set the input's `name` attribute, the following java method calls are used:[7]

```
L8  UIForm form = UIForm.make(notlogged, "basic-form");
    UIInput.make(form, "login-user", "#{logon.name}");
    // #{logon.name} is a macro for the 'name' property in the 'logon' bean.
```

The drawback of the optimization is the change to a non-standard api which is less powerful than the DOM api. A further drawback compared to Enhydra xmlc is that it is not possible to validate the `rsf:id` references at compile time as they are specified as `String`s in the java source and can therefore only be validated at runtime by e.g. manual testing or unit tests.

2.4 A 'prototype': my third template system

This section describes a template system which I created in 2005[8] because I was not satisfied with the macro-based template systems nor with the performance of Enhydra xmlc- based pages and RSF was not widely published nor production-ready until late 2006.[9] The reason why 'prototype' is in quotes is that the system was not conceived as a prototype but as 'the system'. For this report I will refer to it as the prototype as it has never been given a real

[5] http://www2.caret.cam.ac.uk/rsfwiki/

[6] Even though browsers support the syntax for empty elements (e.g. `<img.../>`), it is not valid html: http://webkit.org/blog/68/understanding-html-xml-and-xhtml/

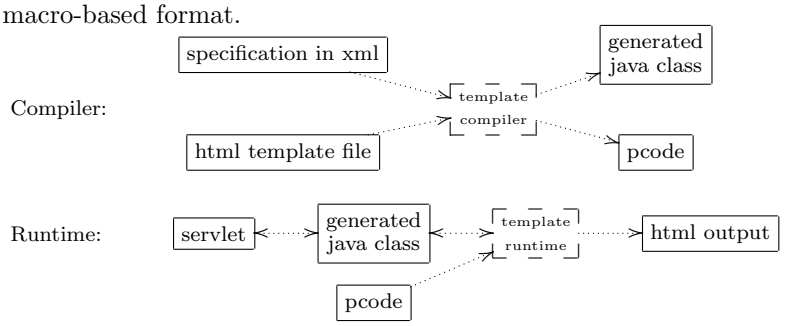
[7] http://www2.caret.cam.ac.uk/rsfwiki/Wiki.jsp?page=LogonTest_2

[8] for http://print.ruc.dk and http://faktura.ruc.dk

[9] RSF Timeline, slide 34, http://www2.caret.cam.ac.uk/rsfwiki/attach/Presentations/Introducing+RSF+-+31-05-06.ppt

name and it is the main inspiration behind this project. The system is strongly inspired by the design of Enhydra xmlc and fixes the performance problem by giving up access to a DOM api at runtime.

The template system compiles the html file into a java class which calls methods in the servlet for each dynamic part of the template. The callbacks include simple substitutions, conditionals and simple sub-tree loops (e.g. for table-rows). The dynamic parts are specified in an xml document and all specifications target html elements (or their attributes) using the elements id or class attribute. The specifications are therefore at the DOM level, whereas the implementation uses pcode (and generated code for the callbacks) similar to a compiled macro-based format.



Short example:

```
L9   <div>
       <img id=LogoImage src='/images/ruc.jpg'><br>
       You're logged in as <span id=PersonFullName>John Doe</span>.
       <div id=HasPrinters>You have access to at least one printer.</div>
     </div>
```

The xml specification:

```
L10  <transformations>
       <attr-src e='LogoImage'/>
       <content e='PersonFullName'/>
       <if e='HasPrinters'/>
     </transformations>
```

The children of `<transformations>` specify how the html file is compiled as they specify the transformations that are performed on the html document. The name of the child element specifies the type of transformation. The `e` attribute specifies which html elements the transformation applies to: In this example, it specifies the html element's `id` attribute.

`<attr-src e='LogoImage'/>` specifies that the `src` attribute of the image is made at runtime by calling the method `printLogoImage()` in the servlet (see interface below).

`<content e='PersonFullName'/>` specifies that the content of the element (all its DOM children) is made at runtime by calling the method `printPersonFullName()` in the servlet.

`<if e='HasPrinters'/>` specifies that the `<div id=HasPrinters>` subtree should only be included if the callback `testHasPrinters()` returns true.

The template system creates an (inner)interface in the generated java file which the programmer must implement to provide the functionality for generating the specified output:

```
L11  public interface CB { // "CallBack"
       void attrSrcLogoImage();
       void printPersonFullName();
       boolean testHasPrinters();
     }
```

2.4.1 Advantages and drawbacks

\+ Like Enhydra xmlc, the system uses clean html template files without any special tags and is well-suited to visual editors and checking the design in a browser before the java servlet is implemented. Mockup data are fully supported, however, unlike Enhydra xmlc they have no runtime overhead as they are omitted at compilation time.

\+ The performance is extremly good as the execution of the template is a mixture of simple interface-based method calls in java and calls to java.io.PrintWriter.print(String). At runtime nothing has to be resolved as everything is hardcoded in the generated java class. There is no callback method or variable lookups in e.g. hashtables.

≈ Not having DOM access at runtime seemed a problem in the beginning as I was used to post-process e.g. `<script>` and `<style>` URLs at runtime, but this was resolved by having the template compiler provide project-specific ability to modify the DOM after it had loaded the html template and before it was compiled into a java class.

\+ The compiled java class provides compile-time check for missing method implementations in the servlet. This cached a significant number of missing methods when I implemented new complex web pages—something which would otherwise have to be found during the testing phase. This feature saves both development and maintenance time.

\− The system uses an extra file (the xml specifications) which (1) must be maintained whenever the interface between the template and the java code is changed, and (2) which takes up valuable screen estate when editing the code and html template.

Maintenance is tedious because whenever a dynamic part of the html template is added/removed/renamed, the corresponding parts of the xml specification have to be changed. After that change is made, the corresponding java methods have to be located and modified. The inverse 2-step procedure is necessary when figuring out what parts of an html template are affected by a java method. Large design changes often cause extra work to ensure that all obsolete xml specifications and/or java methods are removed, i.e. 'manual garbage collection' of the xml specifications and java methods.

\+ It is possible to change the html template and have it recompiled while the server is running. This has the effect of making it possible to experiment with the design while the server is running and seeing how the resulting page will look.

It is not possible to change the xml specification file at runtime as this would add or remove methods. As the java 1.5 jvm doesn't support reloading classes with added or removed methods, a relaunch is required anyway and this makes the needed compilation a relatively small problem.

(−) The system only supports templates stored in the file system and not e.g. in a database. It is therefore not suitable for e.g. CMS systems which don't have a static set of page designs. As I don't develop this kind of web applications, I consider this an unimportant negative aspect (for me).

– No support for java scopes for html-subtree loops. See section below.

2.4.2 Pollution of scope by complex loops

Some html pages display complex tables with information retrieved from e.g. multiple database queries or java objects. This often results in a lot of temporary variables the value and definition of which are only relevant when emitting the html subtree for the row. However, as no support for an inner scope exists, the variables have to be defined as fields in the java servlet scope together with all other 'global' variables for the servlet. A terse example of a shopping cart:

L12
```
<h1>Number of items in cart: <span id=ItemCount>10</span></h1>
<table>
    <tr id=ItemRow>
        <td id=ItemId>42</td>
        <td id=ItemName>Motor</td>
        <td id=ItemPrice>100.00</td>
    </tr>
</table>
```

L13
```
class MyServlet {
    PrintWriter pw = ...; // servlet output
    List<Integer> foundItemIds = ...;
    Iterator<Integer> iter = foundItems.iterator();
    int itemId;
    String itemName;
    BigDecimal itemPrice;

    void printItemCount() { // the count in the header
        pw.print(foundItemIds.size());
    }

    boolean testHasItemRow() { // while-test for the <tr>
        if (!iter.hasNext())
            return false;
        itemId = iter.next();
        itemName = ...; // e.g. one database query
        itemPrice = ...; // e.g. calculation based on another query
        return true;
    }

    void printItemId() { pw.print(itemId); }
    void printItemName() { pw.print(itemName); }
    void printItemPrice() { pw.print(UtilityClass.formatPrice(itemPrice)); }
}
```

In this example, the servlet has the fields `itemId`, `itemName` and `itemPrice` even though they are only used in the methods which handle the contents of the `<tr>` subtree. The 'global' scope of the servlet is therefore polluted with the temporary variables for the loop. The pseudo-java code below illustrates an improved version which utilizes java's block scope for the 3 variables:

L14
```
class MyServlet {
    PrintWriter pw = ...; // servlet output
    List<Integer> foundItemIds = ...;
```

```
    void printItemCount() { // the count in the header
        pw.print(foundItemIds.size());
    }

    void loopItemRow() { // creates all <tr> rows including content.
        for (Integer itemId : foundItems) {
            String itemName = ...;
            BigDecimal itemPrice = ...;
            // pseudo-closures:
            void printItemId() { pw.print(itemId); }
            void printItemName() { pw.print(itemName); }
            void printItemPrice() { pw.print(UtilityClass.formatPrice(itemPrice)); }
        }
    }
}
```

Unfortunately, I have not been able to create a short textbook example which doesn't lead to the obvious objection that an `Item` class should contain the 3 fields. However, more complex pages often have rows that need information which does not fit well together for the rest of the application and the `Item` class would therefore only be created for the particular servlet. An improved template system should support this design pattern by reducing the amount of house-keeping code needed for such loops.

2.5 Generated code versus reflection at runtime

The prototype uses generated java source code (i.e. `*.java` files) instead of java bytecode (or reflection at runtime) to access fields and methods in the servlet. The reason is threefold: Firstly, it is very easy to generate java source. Secondly, I was already in the mindset as I already implemented another tool the sole purpose of which was to generate java source. Thirdly, I had already 2 utility classes for writing nicely formatted java source but no knowledge of libraries for generating bytecode.

In hinsigt, generated java source has some advantages:

1. As the generated source is a java class like all the other java classes the programmer creates, it enables modern java IDE environments to show which templates refer to a specific method as it's a standard "the methods of which refer to this field/method" dependency display.
2. It provides easy access for the template runtime to fields, methods and innerclasses in the servlet class. No need for obscure use of reflection.
3. It is faster for the jvm to access fields/methods by direct access instead of using reflection. However, this speedup is probably insignificant compared to the processing done in typical servlets.
4. The generated output is human readable and it is thus easier to verify and correct bugs in the compiler and easier to debug the execution of a template.
5. The output is checked for e.g. access restrictions by the java compiler when it is compiled into bytecode.

All but the last two advantages are true for bytecode too. However, bytecode is much more complex to generate and verify by manual inspection.

Using java source has significant disadvantages:

6. It requires the programmer to compile his own class, then run the template compiler, and then compile the generated java class. This turns a single-step 'make' into a 3-step procedure. This can be automated in modern IDEs and build scripts but is still an extra overhead. Using bytecode instead of generated source reduces this to a 2-step procedure as the final compile isn't needed.
7. If the template instance is created using a `new XXX()` expression, then the template and the servlet are interdependent. The servlet refers to the template using the `new` operator, and the template contains many callbacks to the servlet. If any of those callbacks are removed from the servlet, the template can't compile. Consequently, the template compiler cannot update the template to not refer to those methods. This problem can be solved by either manually outcommenting most of the generated java class or by instantiating the template indirectly by e.g. `Class.forName()`.
8. The template compiler has to check access permissions to annotated fields/methods as they might not be accessible from the package that the generated java source is located in. If they are inaccessible, the compiler can either generate an error (and forcing the programmer to make a lot of 'internal' fields/methods `public`) or the template system can fall back on using reflection to circumvent the access restrictions.

2.6 Requirements

The requirements for the new system are:

1. Clean html
2. No extra files
3. Reasonable performance
4. Design tinkering
5. Support serialized html-fragments
6. Catch bugs as soon as possible
7. Prioritize maintainability
8. Co-existence
9. Scoping for loops

The requirements are described in detail below.

2.6.1 Clean html

The html template must be valid html with no extensions whatsoever—no special tags, no special attributes. The template must be able to contain mock-up data such as a fake username and links to sample images. The problem with non-valid html is described in section 2.2.1 on page 5.

2.6.2 No extra files

No extra files with the specification of dynamic parts. The problem is described in section 2.4.1 on page 9. The specification must be embedded in the java source.

2.6.3 Reasonable performance

No major performance bottlenecks. Performance should be resonable without resorting to low-level optimizations. Enydra xmlc's overhead of expanding the template to a full DOM is not acceptable (see section 2.3.1 on page 6).

2.6.4 Design tinkering

It must be possible to recompile the template on-the-fly so that a "compile template, compile server, restart server" time-consuming procedure is avoided (see section 2.4.1 on page 9). The new system must support the design of the layout both by using the non-processed template only and by using the processed template.

2.6.5 Support serialized html-fragments

It must allow the servlet to generate html by writing raw html to a java.io.PrintWriter. Forcing the programmer to expand html fragments into a DOM subtree makes some functionality very tedious to implement, e.g. when html fragments are stored as text in an SQL database. The programmer has the full responsibility of generating valid html fragments as the output is not parsed for syntax errors.

I had to hack the DOM-to-textstream conversion in the Enhydra-based web application so that the content of CDATA[10] sections was output as-is without the `<![[ ]]>` marker. This is not an acceptable workaround for getting acceptable performance from a template system.

2.6.6 Catch bugs as soon as possible

The loose coupling between the template and the code in macro-based systems is (in my experience) a very common source of bugs due to typing errors and makes maintenance difficult. Therefore, the system must provide means for validating the use of the template by the servlet before the web application is deployed to save valuable developer time. Many developers use unit tests to catch errors, but—as chapter 7 shows—tests which catches all such errors takes often an extreme amount of time to implement and to run. It is therefore a huge advantage for a template system to be able to catch such errors at compile time.

An obvious solution is to provide a compiler which performs the validation and enables faster execution of the template at runtime by using a compiled form (e.g. pcode). However, the only template systems that I know of which use a separate compiler stage for checking template references are Enhydra xmlc and my prototype.

As part of the validation, the template compiler must make a warning if:

- the html file uses a non-defined 'macro'.
- a 'macro' is defined but is not used by the html template.

It is a very tedious job to manually check for the above problems and it is too easy to make spelling or typing errors. To avoid the warnings, it forces a programming approach of tidiness where deprecated/unused code is cleaned up rather than leaving it as bit rot even if it (initially) requires more work.

2.6.7 Prioritize maintainability

The main problem with the prototype is the externalization of specifications from the html and the java code (see section 2.4 on page 7). It is therefore a requirement that the specifications must be embedded in the java source: No other place is available as external files are to be avoided and the html template must be pure html.

[10] A type of DOM node which isn't valid/used for html documents

The system must use either generated java source or bytecode instead of pure reflection at runtime as the advantages described in section 2.5 on page 11 are considered much more important than the increased build time.

It is my experience that much more time is spent maintaining (and extending) complex web applications than is spent on the initial development—an aspect which has largely been ignored by software development literature ([LST78], [BDKZ93], [BP97]). Assuming this is a general pattern in development, supporting maintainability is considered more important than any reduction in initial development time.

Maintainability is often defined in very broad terms like "change effort" ([BDP06], [LC00]). Html templates are mostly used independently at the source code level (i.e. changing one template seldom has consequences for the use of other templates) so this report uses a more narrow definition of what constitutes maintainability:

- concise code, i.e. readability combined with non-verboseness. This means that e.g. the specifications should not be externalized in a file and they should be as terse as possible to avoid bloating the java source. The unreadability of 'html inside java source' of early template systems must be avoided.
- it must be easy to predict the consequences of changes, e.g. removing/renaming java methods and fields. It should moreover be easy to figure out how the java code must be changed if parts of the html are deleted/modified.

2.6.8 Co-existence

The template system must be able to co-exist with other template systems in the same java application as I will use the template system for web applications with huge amounts of code that cannot be migrated/reimplemented within a reasonable time frame. This implies that the template system must only implement the View-part of MVC and not try to be a complete web application framework—the template system should be usable by more than a single framework.

2.6.9 Scoping for loops

The new system must provide some support for scopes in the java servlet so that variables used inside a loop only are only defined when emitting the html subtree for the loop. This problem is described in section 2.4.2 on page 10.

2.7 Conclusion

Enhydra xmlc does not meet the performance requirement and is not easy to fix without changing the api and implementation so it is not DOM-based. The main problem with RSF—which solves Enhydra xmlc's performance problem—is the lack of compile time check of `rsf:id` attributes the consequence of which is that validation of the templates can only be performed at runtime. So RSF solved the performance problem but lost the compile time checks.

The prototype meets the requirements with the exception of the externalization of the specifications in the xml file and lack of scoping support for complex loops. The next chapter will therefore examine how the specifications can be embedded in the java source in an non-obtrusive and robust way.

Chapter 3

Annotations: Embedding structured metadata

The prototype stores the specifications of the dynamic parts of the html templates in external xml documents. The purpose of this chapter is to find a method for embedding the specifications in java source files.

3.1 Using comments

Before annotations were introduced in java 1.5, the typical method of embedding metadata in java source was to create more or less structured comments. The main problem with such 'comments' is that another language is introduced inside the java source and that the java compiler and IDEs treat the 'comments' as comments.

This leads to severe drawbacks:

1. Unless the java compiler has hardcoded support for the embedded language (e.g. Javadoc), the embedded code is not validated at compile time but simply ignored.
2. There is no support in the java runtime to access the embedded code. The comments must be processed by an external tool which compiles the embedded code and stores it elsewhere.
3. Simple comments are fragile. As it is 'just a comment', IDEs can't detect that the comment belongs to a method/field. When code is refactored, e.g. moved to another class, the IDE doesn't know that the comment should move together with the method/field.
4. As java doesn't provide any building blocks for creating the embedded language, it tends to become complex to parse. On the surface Javadoc seems simple ('it's just html') but it requires a specialized parser as it extends html with special syntax[1] for specifying arguments, exceptions, links, etc. and it is whitespace sensitive after newlines. E.g.
   */** @deprecated As of JDK 1.1, replaced by {@link #setBounds(int,int,int,int)} */*
5. Since the comments annotate the following method/field in the java source and often need to know e.g. the name of the method, the tool which processes the comments

[1] http://java.sun.com/j2se/javadoc/writingdoccomments/

must be able to parse a significant part of the java language syntax, thereby making the tool much more complex. Whenever the java language is extended, the tool must be extended before it can parse files using the new java constructs.

3.2 The annotation type

The main purpose of adding annotations to the java language was to formalize the specification of metadata and to make them available at runtime.[2] The purpose was not to replace Javadoc or specify a language which could replace the Javadoc 'html' but to provide a general base for building 'simple' blocks of metadata.

Java annotations are surprisingly simple and powerful as they define the metadata as immutable object-oriented data structures by using (a subset of) java interfaces. A complete description of java annotations is outside the scope of the project so this chapter only provides a very compact introduction. For a full description, see [MF04] (recommended) or [AGH06].

Annotations are defined using `@interface` and must not extend another annotation or interface. The parameters are defined as methods with some restrictions:

- No parameters are allowed.
- It must not be generic, i.e. template type.
- A throws clause must not be specified.
- Allowed return types are primitive types (boolean, byte, int, long, float, double); String; class (generic java.lang.Class or specific class); enum (i.e. value, not the class); an annotation type; array of any of these.
- Default values can be provided using the `default` keyword. Only non-null constant values are allowed.

Annotations are allowed to be defined without any methods and can as such be used as simple markers—just like normal interfaces. It is allowed to apply annotations to the annotation `@interface`, including the definition of the annotation itself.

Annotations can be made available at runtime, or only in `.class` files, or only in the source. This is controlled by annotating the annotation with the `@Retention` 'meta' annotation. The default is `.class` retention.

The `@Target` 'meta' annotation is used to specify if the annotation may be applied to class, constructor, method, field, parameter, local variable, annotation and/or package definitions. It is not possible to specify any other criteria, e.g. only methods which take no arguments, or only `public` methods.

3.3 An example

```
L15  import java.lang.annotation.Target;
     import java.lang.annotation.ElementType;
     import java.lang.annotation.Retention;
     import java.lang.annotation.RetentionPolicy;
```

[2] JSR 175: A Metadata Facility for the Java Programming Language: http://jcp.org/en/jsr/detail?id=175

```
@Target(ElementType.METHOD) // may only be applied to methods
@Retention(RetentionPolicy.RUNTIME) // available at runtime using reflection
@interface MyAnnotation {
    String value() default "";
    boolean funny() default false;
}

class MyClassUsingAnnotations {

    // all arguments specified.
    @MyAnnotation(value = "DocumentName", funny = false)
    void printPrintjobName() {
        pw.print(job.getDocumentName());
    }

    @MyAnnotation(value = "SupportEmail") // default value is used for 'funny'
    void printAdminEmail() {
        pw.print("print@ruc.dk");
    }

    // This is semantically the same as the previous, as
    // value=... is assumed if only a single value is specified.
    @MyAnnotation("SupportEmail") void printAdminEmail() {
        pw.print("print@ruc.dk");
    }

    @MyAnnotation // uses default values
    void printUserFullname() {
        pw.print("Michael Nyman");
    }
}
```

3.4 Runtime access

Provided the annotations are marked with RetentionPolicy.RUNTIME, it is easy to use reflection at runtime to access the specified annotations and the parameter values:

```
L16  public class DisplayMethodAnnotations {
        public static void main(String[] args) throws Exception {
            Class c = Class.forName(args[0]);
            for (Method m : c.getMethods()) {
                for (Annotation a : m.getAnnotations()) {
                    System.err.println(c.getName() + "." + m.getName() +
                            " has annotation: " + a.getName());
                    if (a instanceof MyAnnotation)
                        System.err.println(" is @MyAnnotation, value = <"
                            + ((MyAnnotation)a).value() + ">");
                }
            }
        }
    }
```

3.5 Lists

Annotations are not allowed to be applied to the same class/method/field etc. twice. A workaround for this problem is to create a container annotation (C) the single parameter of which is a value the type of which is an array of the annotation (A). A container array must be created for each annotation type as a generic list type isn't allowed due to the restrictions on parameter types:

L17
```
@interface A { int param(); }
@interface C { A[] value(); }
@C({@A(1), @A(2), @A(3)}) class Demo { ... }
```

3.6 The non-null problem

To make an annotation parameter optional, a default value is specified. For method parameters, it is typical to use `null` to indicate default or non-specified values, however, `null` is surprisingly not allowed as an annotation parameter value.[3] A possible workaround for complex types is to use a value which is distinct from semantically valid values for the parameter. For example, a css class name cannot be the empty string, so `String cssName() default ""` is a solution. As kind of workarounds requires a non-used value, it is often not usable for primitive types with a small set of values, e.g. `boolean` or `byte`. A workaround for `boolean`s is to use an enum for (true/false/not-specified).

3.7 Template specification example

This section demonstrates how annotations can be used as the specification format for a html template system. Assume we want to display the user's current printer. The output should be something like:

L18
```
<div>Selected printer: HP LaserJet 4200</div>
```

Because the printer name must be addressable using simple DOM operations, the name must be enclosed in a single html element, e.g. a `<span>` element with an `id` attribute:

L19
```
<div>Selected printer: <span id=PrinterName>HP LaserJet 4200</span></div>
```

A servlet method provides the actual name of the printer:

L20
```
String getPrinterName() {
    return "42-1-02-hp4250-drift-dat"; /* e.g. database query */
}
```

The specification that the content of the `<span>` element should be replaced by the output from the method can be made as an annotation directly on the method—providing a high degree of locality for the implementation and the specification, unlike the xml-based prototype:

L21
```
@ProvidesUnencodedElementContent(id = "PrinterName")
String getPrinterName() {
    return "42-1-02-hp4250-drift-dat"; /* e.g. database query */
}
```

[3] The language reference [AGH06] which is the result of the JSR 175 process doesn't state the reason for disallowing null.

An annotation (@interface) is defined for each possible action for an element, e.g. element content, attribute value or element conditional. The above annotation specify dynamic generation of element content, but the name and parameter specification are verbose and takes up quite some space. The definition of the annotations are an important part of the template system's design because it has a large impact on how easy it is to understand how the servlet and the html template is tied together.

3.8 Conclusion

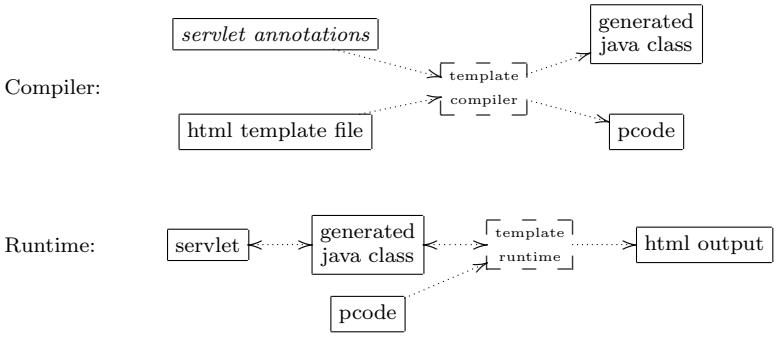
Annotations provide a method for embedding metadata which solve the major problems that plague comment-based metadata.

Annotations are well suited to implement metadata which can be structured as (possible nested) containers of simple types. For metadata which lends to parsing (e.g. SQL or html snippets) it is difficult to use annotations beside embedding the data encoded in a `String` parameter.

The prototype uses simple xml-based specifications which can be implemented using annotations without any major problems because of the simple structure of the xml documents: They contain a flat list of elements with the data specified as the element's name and its attributes. The only problem is the need to be able to apply the same annotation several times on the same method/field, but—as next chapter shows—this can be solved by using array parameters instead of multiple annotations.

Annotations are therefore a very suitable replacement for xml as specification format for the template system. It is important for the readability of the servlet code that the annotation's name and parameters are as concise as possible. The next chapter provides the design of all the annotations in the template system.

The structure of the new system is very similar to that of the prototype (see page 8) as the only difference is the replacement of the xml specification file with the servlet annotations. The implementation is very different, however, due to support of scopes and of methods with return values and fields instead of only void methods—concepts which are introduced in the next chapter.



Chapter 4

Design of template annotations

This chapter documents the design of the annotations in the new template system. Refer to the next chapter for small examples of how the annotations are used in servlets.

The annotation based directives can be divided into 4 groups:

1. Specification of html element content which is generated by callbacks.
2. Specification of html element attributes which are generated by callbacks. This is subdivided into simple attributes, attribute collections and sub-attributes.
3. Specification of flow control: conditionals and loops.
4. Specification of which html files should be compiled and global options for a file.

To avoid name clashing with 3rd party annotations, the annotation names are prefixed with a single letter whereby resorting to use the full package name is avoided. The prefix is `A` for plain attributes, `S` for sub-attributes, `E` for elements and `T` for general template attributes.

4.1 Targeting html elements

The first 3 groups all specify an action for one or more html elements. In order for the declarations to be as terse as possible, an explicit declaration is not required whenever a resonable default value can be determined—aka "convention over configuration".

The elements are resolved by the following prioritization:

1. The `id`, `className` or `tag` annotation parameter in any combination. The default parameter for the annotation must not be specified in this case.

   The `id` parameter specifies the value of the element's `id` attribute. The `className` parameter specifies one of the names listed in the elements `class` attribute. The `tag` parameter specifies the name of the element without angle brackets (e.g. '`head`', '`title`', '`form`').

2. The default parameter of the annotation, which matches both `id` and `class` attributes but generates a warning if it matches both kinds.

3. If applied to a method: The name of the method, stripped of an [a-z] `+` prefix, e.g. "printSomething()" can refer to `<span id=Something>...</span>`. Has the same id/class attribute behaviour as rule 2.

4. If applied to a field: The name of the field, e.g. "String userName;" can refer to `<span id=UserName>...</span>`. Has the same id/class attribute behaviour as rule 2.

To support the prioritization, annotations which target html elements have 4 parameters shown below, which unfortunately can't be specified in a common super annotation as annotations can't derive from other annotations.

As described in section 3.5 on page 18 it is not possible to apply the same annotation multiple times on the same method/field. The solution is to allow the same annotation to target multiple id/class/tag names instead of a single name by declaring the parameters as arrays.

L22
```
@Target({ElementType.METHOD, ElementType.FIELD})
@interface EHtml {
    String[] value() default ""; // empty string means "not specified"
    String[] id() default "";
    String[] className() default "";
    String[] tag() default "";
}
```

Since it isn't possible to specificy any constraints in the annotation declaration for the methods/fields they are applied to, it must be checked by the template compiler. Most of the template annotations constrain the (return) type to a specific type and require methods to take no arguments.

4.2 Annotation reference

Table 4.1 on page 22 lists all the 74 annotations.

As regards plain attribute annotations (i.e. those prefixed with `A`): It is not allowed to have several instances of the same annotation to match the same element as an attribute's value must only be specified once. Attribute collections generate several attributes, so neither `@ASrc`, `@AWidth` nor `@AHeight` must match the same element as `@ASrcWidthHeight`.

Multiple sub-attribute annotations (i.e. those prefixed with `S`) for the same element must not specify the same sub-attribute name. This requirement for unique sub-attributes corresponds to the requirement for unique plain attributes. If a sub-attribute is specified, the corresponding plain attribute must not be specified, i.e. combining `@SDisplay` and `@AStyle` is illegal.

Multiple annotations marked with the same numerical legend must not match the same element. For example, combining `@EWhile` and `@EIterate` is illegal, whereas combining `@EWhile` and `@EIf` is legal. The loop is only executed if `@EIf` returns true and it is not retested inside the loop, i.e. the conditionals enclose the loops.

Multiple flow control annotations (*) are allowed for the same element. They are combined using the boolean `and` operation.

Multiple `@EZap` annotations (**) are allowed to match the same element.

String attributes

Attribute	Annotation
action	@AAction
align	@AAlign
class	@AClass, @AClassPost
clear	@AClear
coords	@ACoords
enctype	@AEncType
for	@AFor
height	@AHeight
href	@AHref
id	@AId
name	@AName
onblur	@AOnBlur
onclick	@AOnClick
ondblclick	@AOnDblClick
onfocus	@AOnFocus
onkeydown	@AOnKeyDown
onkeypress	@AOnKeyPress
onkeyup	@AOnKeyUp
onload	@AOnLoad
onmousedown	@AOnMouseDown
onmousemove	@AOnMouseMove
onmouseout	@AOnMouseOut
onmouseover	@AOnMouseOver
onmouseup	@AOnMouseUp
onreset	@AOnReset
onselect	@AOnSelect
onsubmit	@AOnSubmit
onunload	@AOnUnload
src	@ASrc
style	@AStyle, @AStylePost
type	@AType
valign	@AValign
value	@AValue
width	@AWidth

Integer attributes

Attribute	Annotation
colspan	@AColSpan
rowspan	@ARowSpan
tabindex	@ATabIndex

Boolean attributes

Attribute	Annotation
checked	@AChecked
disabled	@AEnabled
multiple	@AMultiple
ismap	@AIsMap
readonly	@AReadOnly
selected	@ASelected

Attribute-collections

named list	@ACollection
src+width+height	@ASrcWidthHeight

Sub-attributes

@SValue	@SDisplayBlock[1]
@SCollection	@SDisplayInline[1]
@SHasClass	@SDisplay[1]

Element content

@EText[2]	@EHtml[2]
@ETextPre[3]	@EHtmlPre[3]
@ETextPost[4]	@EHtmlPost[4]
@EReplace[2+3+4]	

Flow control

@EIf*	@EWhile[5]
@EIfNot*	@EScopingWhile[5]
@EContentIf*	@EIterate[5]
@EContentIfNot*	@EScope[5]
@ETagIf*	
@ETagIfNot*	
@EZap**	

Template compilation

@TCompile	@TIgnore
	@TOverrides

Table 4.1: Annotation reference. See section 4.2 on page 21 for legends.

4.3 Template options

Compilation of an html template is triggered by applying the `@TCompile` annotation to a class (inner or outer):

L23
```
@Target(ElementType.TYPE)
@interface TCompile {
    String filename() default "";
    String fragment() default "";
    String tpname() default "";
}
```

filename specifies the name of the html file. The .html/.htm suffix may be omitted. The default name is the name of the java source file. The html file must be located in either the same directory as the java source or in a sub-directory named 'templates'.

fragment specifies which part of the html template should be output. Usually it is the complete html document (denoted by the default empty string), however, if e.g. only the contents of the subtree `<div id=PrinterInfo>...</div>` is required, it can be selected using fragment="PrinterInfo" (see rule 2 on page 20).

tpname specifies the simple class name of the generated java class. The default value is the simple name of the contained java class suffixed by 'Tp'. If a sub-directory named 'templatedata' exist in the directory where the html source is located, the generated source is saved in that package/directory. Otherwise it is saved in the same 'package' as the html source.

4.4 Generated element content

The `@EHtml` and `@EText` directives specify that the content of the elements is generated by a 'callback' which can either be:

- a field of type String
- a method without arguments and which returns a String
- a void method without arguments (`@EHtml` only)

The difference between the two annotations is that `@EHtml` outputs the result `String` as-is, whereas `@EText` html-encodes it. The two annotations correspond to MSIE-Javascript's Element.innerHTML and Element.innerText properties. The output completely replaces the content specified in the html template, but not the tag or attributes. If the method returns void, it is assumed that the method prints the content directly to the java.io.PrintWriter.

4.4.1 Pre- and postcontent

Instead of replacing the complete value, the system supports inserting output before and/or after the element. This is useful for e.g. adding an `<input type=hidden>` to a `<form>` element.

- `@EHtmlPre` and `@ETextPre` insert content before the template content
- `@EHtmlPost` and `@ETextPost` insert content after the template content

Both types can be specified on an element.

4.4.2 Complete replacement of element

Output from `@EReplace` replaces the complete element including tag and attributes.

4.4.3 Examples

L24
```
@EHtml void printPersonFullName() {
    pw.print("Elliott Smith");
}

@EText String getPersonTitle() {
    return "Student";
}

@EReplace("Logo") void printLogoImageTag() {
    pw.print("<img src=... width=42 height=10>");
}

@EHtmlPre(tag = "form") String sessionInfo() {
    return "<input type=hidden name=session-id value=" + xxxx + ">";
}
```

4.5 Simple attributes

The value of simple attributes is generated in a way which is similar to the way element content is generated by `@EText`. The value specified in the html template is completely replaced by the value provided by the callback. The template system generates the attribute name, equal sign and single quotes, so the callback must only generate the value. A specialized annotation is provided for each common html attribute. For the 'class' and 'style' attributes, a post-content annotation is provided as the content of those two attributes are lists.

4.5.1 String-based attributes

The string-based annotations are very similar to `@EText` and can be applied on String fields and methods returning a String. Example:

L25
```
@Target({ElementType.METHOD, ElementType.FIELD})
@interface AValue {
    String value() default ""; // element selection, empty string is "not specified"
    String id() default "";
    String className() default "";
    String tag() default "";
}
```

4.5.2 Integer-based attributes

Integer-based attributes are handled like String-based attributes except that fields and method return types must be `int` or `Integer`. No quotes are generated in the output html as they are not needed for plain html.

4.5.3 Boolean attributes

Boolean attributes such as 'checked' for checkboxes are handled using boolean methods or fields instead of generating output. The `disabled` attribute is handled as an inverted `enabled` attribute to avoid having double negations in the programmer's code (aka "I do not want this to be disabled"). Fields and methods return types must be `boolean` or `Boolean`.

L26
```
@AChecked boolean selectedPrinterCheckbox() {
    return p.id == 42;
}

@AEnabled boolean enabledPrinterCheckbox() {
    return p.selectable();
}
```

4.6 Attribute collections

This type has 2 purposes:

- To support uncommon attributes for which no specialized annotation exists.
- To add several attributes using one method.

The latter is useful when the attributes are provided as pre-made by some other utility. E.g. for image tags, a simple cache might return the src, width and height tags as one string, e.g. "`src=...width=200 height=152`". This is supported by `@ASrcWidthHeight`. The programmer must generate the full attribute list, including attribute name and equal sign.

Extra attributes can be added using `@ACollection`. The only mandatory argument 'attributes' is the list of attributes that the programmer adds and are therefore suppressed by the template compiler to avoid having the attributes specified twice in the output.

L27
```
@Target({ElementType.METHOD, ElementType.FIELD})
@interface ACollection {
    String value/id/className/tag element targeting quadruplets ...
    String[] attrNames(); // name of the attributes
}
```

4.7 Sub-attributes

For the `style` and `class` attributes, the system supports generating specific parts of the attribute value instead of replacing the complete value. Sub-attributes for the `style` attribute can be considered "attributes within attributes". The `class` attribute is a list of names, so it is a set rather than a map.

4.7.1 CSS class

It is often useful to control the display of html elements by adding/removing a css class name from the element's `class` attribute. The `@SHasClass` annotation takes the parameter '`classes`' which is a list of those class names that are output if the method/field returns true. The parameter name clashes somewhat with the '`className`' parameter which specifies which elements the annotation targets, however, this should not be too problematic as the former is mandatory.

L28
```
@Target({ElementType.METHOD, ElementType.FIELD})
@interface SHasClass {
    String value/id/className/tag ...
    String[] classes(); // name collision with className, but this parameter is mandatory
}
```

4.7.2 CSS style declarations

Adding/removing CSS classes doesn't always provide the needed flexibility, or it adds too many single-purpose CSS classes. More flexibility is obtained by directly modifying the individual specifications in 'style' attributes. This is a more fine-grained approach than replacing the complete attribute value using `@AStyle` as the template compiler keeps all the other specifications, thereby keeping whatever the web designer might have added. The control of style specifications is analog to the control of attributes in an element—attributes within an attribute. Unlike element attributes, very few specialized annotations are provided for the `style` attribute.

Non-specialized declarations

Use `@SValue` to generate a full declaration for a single style or multiple styles. The compiler generates the name, colon and semicolon. The programmer only generates the value. If multiple styles are specified, each of them is output using this sequence. Any definition of that particular declaration in the html template is ignored, corresponding to simple element attribute annotations such as `@AValue`.

L29
```
@Target({ElementType.METHOD, ElementType.FIELD})
@interface SValue {
    String value/id/className/tag ...
    String name(); // css name of style
}
```

Use `@SCollection` to generate multiple or extra css style declarations. It is similar to `@ACollection` for adding attributes to an element. It takes the argument 'styles' which is the list of names of style declarations which the compiler should suppress.

L30
```
@Target({ElementType.METHOD, ElementType.FIELD})
public @interface ACollection {
    String value/id/classNAme/tag ...
    String[] attributes(); // name of the attributes which the callback generates
}
```

Display

The display annotations control the visibility of an element. The methods and fields that they are applied to must be boolean.

`@SDisplayBlock`**:** depending on the return value of the field/method, a "display: block" or "display: none" declaration is added to the style attribute. This overrides any (default) display value assigned to the element using style sheets.

`@SDisplayInline`**:** like `@SDisplayBlock` but uses "display: inline".

`@SDisplay`**:** if the method/field returns false, "display: none" is added. Otherwise, nothing is added. This can be used to control the visibility of table rows and cells (provided they are not hidden using style sheets) as MSIE fails to recognize the proper `display` values of the elements, thus requiring browser-specific values for the visible state.

4.8 Conditionals

Conditionals enable the omission of the tag, the content or complete html subtrees in the output. Inclusion is determined at runtime by the return value of a method or field value.

4.8.1 @EIf, @EIfNot and @EContentIf, @EContentIfNot

The `@EIf` and `@EIfNot` annotations handle omission of an element and all its children. The seldom-used `@EContentIf` and `@EContentIfNot` directives omit the children of the element only but not the tag itself.

The `@EIf` and `@EContentIf` annotations have the parameter '`inverse`' which can be used to negate the test, i.e. include the subtree if the method/field returns false. The `@EIfNot` and `@EContentIfNot` annotations are a convenience notation for specifying `inverse = true` and helps avoid the problem of not being able to apply the same annotation to a method/field twice:

L31
```
// Not valid:
@EIf({"xx", "zz"}) @EIf(value = "yy", inverse = true) boolean invalidTest() { ... }

// Valid:
@EIf({"xx", "zz"}) @EIfNot("yy") boolean validTest() { ... }
```

4.8.2 @EZap

`@EZap` works like an `@EIf` which always returns false, except that the omission is done at compile time and the subtree is completely ignored by the compiler similar to a pre-processor directive. It can be applied to any method/field/class.

4.8.3 @ETagIf and @ETagIfNot

`@ETagIf` is complementary to `@EContentIf`: it controlls omitting the element tag instead of the content. For the input `<div><a id=Link href='...'>test</a></div>` the output of `@ETagIf boolean testLink;` is:

- if true: `<div><a href='...'>test</a></div>`
- if false: `<div>test</div>`

Like `@EIfNot`, `@ETagIf` has an `inverse` parameter and `@ETagIfNot` uses the negated test condition.

4.9 Simple loops

Simple loops are handled using `@EWhile` which must be applied to a method which returns a boolean. Each time the method returns true, the subtree is processed. It thus works exactly like: **while** (method()) emitSubtree();

4.10 Scoping

Scoping is an important building block of advanced loops. A scope is created by using an inner class when processing the subtree. Annotations on methods/fields in the parent class are still used, so the scope can be considered a search path, however, the compiler will warn if duplicates are found as they are very often a symptom of a programmer bug.

The `@EScope` annotation is made on an inner class. An instance of the class will be created when the element is processed. Instantiation is done using either:

- by calling a method whose name is the name of the inner class and has the prefix 'new'. For an inner class named 'PrinterRow', the method must be named 'newPrinterRow', must not take any arguments, and must return a 'PrinterRow'.

- alternatively, it is instantiated using reflection.

The inner class is allowed to be static or non-static but must not be nested inside another inner class. `@EScope` outputs the subtree once and can be considered a one-time loop.

4.11 Advanced loops

Advanced loops are based on scopes. An instance of the inner class is created for each iteration of the loop. The annotations are:

- `@EWhileScoped` which can be applied to a boolean method. The inner class' constructor must not take any arguments.
- `@EIterate` which is applied to a method/field which returns an Iterator/Iterable. The inner class' constructor must take exactly one argument—the objects returned by the iterator.

`@EScope` must not be specified on the inner class as it is implicit in the above looping annotations because the scoping implementation is different for iteration.

The reason for having separate `@EWhileScoped` and `@EIterate` instead of overloading `@EWhile` is a wish to make the programmer's code more robust by avoiding a change in semantics if the programmer by chance defines an inner class with a matching name.

The inner class can be specified using the '`scope`' annotation parameter. If no scope parameter is specified, the inner class is found by using the name determined by:

1. if any element-targeting value for the annotation has been specified, 'Emit' is added as a prefix.
2. else, if the annotation is specified on a field: the name of the field with first letter uppercased and prefixed with 'Emit'.
3. else, if the annotation is specified on a method: the name of the method with an `[a-z]+` prefix replaced by 'Emit'.

4.12 Miscellaneous annotations

`@TIgnore` makes the compiler ignore all (other) annotations on the class/field/method. It doesn't have any parameters. This can be used as a simple commenting-out which is easier to add than using `/* ... */`

`@TOverride` makes the compiler suppress warnings for annotations in sub-scopes which override annotations in parent scopes with the same element-selecting parameters. As overriding annotations are seldom intended in inner scopes, the compiler defaults to produce a warning if one annotation shadows another one.

4.13 Build system

As the build system is outside the scope of this report, it will only be described very briefly here. The system scans a directory subtree for .java files and tests if the files contain `@TCompile` by simple text search (`String.contains()`) thus giving false positives for files which only contain "@TCompile" inside a comment and not as a class annotation.

The compiler is executed for each matching file/class. If the class isn't annotated with `@TCompile`, it returns without doing anything. In addition to any generated java class, the output is a list of dependent source files and their modification dates. The build systems use this information in the following runs to quickly determine which files need

recompilation—similar to a unix makefile. False positives are not treated differently by the build system—they just don't depend on any other files.

4.14 Summary

This chapter provides documentation of the annotations without examples of how they are used to implement common constructs on web pages. The next chapter tests how usable the design is to implement common web design patterns.

Chapter 5

Test of the design

This chapter tests how usable the design of the annotations is. The testing is performed by implementing common web design patterns. The success criteria is concise servlet code, especially that the annotations don't reduce the maintainability and readability of the java code.

A lot of books on Design Patterns [GHJV95] have been published for common programming languages. For web design, the amount of material is unfortunately quite small and based on user-level/usability patterns (e.g. [Wel]) rather than implementation patterns. Consequently, the tests used in this chapter are based solely on patterns which I have identified while implementing web applications. The patterns are described in terms of DOM elements because the previous two systems which I've used[1] are DOM based at the specification level. The patterns are:

content substitution: A simple example is to output the real user's name instead of a placeholder in "`<div>You're logged in as John Doe</div>`". Sometimes only parts of the content should be replaced.

Conditional: The ability to omit parts of the template. The parts are defined at compile time, however the inclusion/omitting is determined at runtime. Some web-based shops require customers to login before they can submit orders but allows the customers to collect items in the shopping basket while not logged in. If the customer is not logged in, the "You're logged in as..." message `<div>` element must be omitted, i.e. a conditional.

Tag conditional: Administrative users often have more links available on a page, e.g. to edit information or to display internal company information for an item. For the normal user the links must be omitted and only the plain text should be displayed.

Attribute value: Link destination address (`<a href=...>`), image source address (`<img src=...>`), `input` name/value, etc. are often determined at runtime.

Sub-attribute: Sometimes only a single name in a `class` attribute should be added (or omitted) and sometimes the `style` attribute needs to be modified. Changing the complete attribute value makes the template fragile with regard to modifications made by the web designer after the initial development and deployment.

[1] Enhydra xmlc and the prototype.

Simple loop: Most web applications make database queries and display the results in a table or list. For each query result, the template system should loop over a designated sample row/list-item and emit the content for each query result, including performing all the replacements etc. inside the sub-template.

Complex loop data: Some rows contain complex information, e.g. each row requires separate database queries or complex calculations. Sometimes summary views display several pieces of information which are not combined into a single entity in the rest of the application. It is therefore useful to be able to easily create a local model for each row/list-item in template loops.

Multiple templates in the same servlet: The typical example is the 'print friendly' version of a page which cannot be created using only style sheets but must be created using a different template. Usually such pages share almost all dynamic parts and the required template is indicated by e.g. a 'print=1' URL argument which makes it difficult to use a specific servlet for each template.

The inverse pattern is to use the same template in several servlets although two completely different implementations for the same page is quite uncommon: I only use this inverted pattern for very simple pages, e.g. 'quick and dirty' pages which are not displayed to normal users but only to developers/admins, e.g. debug dumps. The only dynamic parts of those simple pages are the title and a single content substitution in the main body.

Partial template: Parts of AJAX based pages are updated using Javascript's innerHTML. When the page is designed, the sub-part is designed as an integral part of the complete page rather than a separate template. This creates the need for generating only the sub-part of a template without the overhead of generating the complete page and then chopping off the unwanted (pre and post) parts.

5.1 Element content

The code below shows how to perform simple substitutions by creating a servlet that outputs "You're logged in as John Doe (jdoe)." using the real persons full name and username. The two dynamic texts are enclosed in a `<span>` so that they are separated from the rest of the text, corresponding to how replacements are typically made using client-side Javascript.

L32
```
<div>You're logged in as <span id=PersonFullName>John Doe</span>
        (<span id=Username>jdoe</span>).</div>
```

The 'servlet' class is:

L33
```
@TCompile
public class DesignTest1 {
    PrintWriter pw = new PrintWriter(System.out); // servlet output

    @EHtml
    void printPersonFullName() {
        pw.print("Elliott Smith");
    }

    @EText
    String username = "es";
```

```
    void process() {
        DesignTest1Tp tmpl = new DesignTest1Tp(this);
        tmpl.emit(pw);
    }

    public static void main(String[] arg) {
        DesignTest1 servlet = new DesignTest1();
        servlet.process();
        servlet.pw.flush();
    }
}
```

The overhead of simulating a servlet takes up a large part of the java code: The declaration of the PrintWriter and the `main` method. The `process` method has 2 lines: the first one instantiates the template using the servlet instance with `this` specified as the root scope. The scope is used by the template runtime to access the `username` field and to call the `printPersonFullName` method. The second line calls the `emit` method which processes the template and outputs the result to the PrintWriter passed as the argument.

Specifying the substitutions takes very little code. No parameters are needed for the `@EHtml` annotations as the method and field names match the `id` attributes in the html template. Combined with the `@TCompile` line to trigger the compilation, the total template overhead is 3 simple lines of code—the 3 annotations.

5.2 Conditionals

This test extends the previous test by adding a conditional message which tells the user if he/she has access to any printers:

```
L34  <div>You're logged in as <span id=PersonFullName>John Doe</span>
            (<span id=Username>jdoe</span>).</div>
     <div id=NoPrinters1>You don't have access to any printer</div>
     <div id=NoPrinters2>Please ask your local system administrator to give you access.</div>
     <div id=HasPrinters>Congratulation. You have access to at least one printer.</div>
```

The output should contain either the two `<div id=NoPrintersX>` messages or the `<div id=HasPrinters>` message. The 'servlet' code is:

```
L35  @TCompile
     public class DesignTest2 {
         List<Printer> accessiblePrinters = ...; // the list of printers available to the user
         static class Printer { ... }

         // the interesting method:
         @EIf
         @EIfNot({"NoPrinters1", "NoPrinters2"})
         boolean testHasPrinters() {
             return !accessiblePrinters.isEmpty();
         }

         // the code below is the same as the previous test:
         PrintWriter pw = new PrintWriter(System.out);
         @EHtml void printPersonFullName() { pw.print("Elliott Smith"); }
         @EText String username = "es";

         void process() { new DesignTest2Tp(this).emit(pw); }
         public static void main(String[] arg) { ... }
     }
```

The two annotations applied to the `testHasPrinters` method control the inclusion of the texts. The `@EIfNot` annotation controls two different elements by specifying an array parameter. Another solution is to use `class=NoPrinters` instead of `id` attributes as this enables targeting several html elements by using a single parameter: `@EIfNot("NoPrinters")`.

5.2.1 Tag conditionals

Users with administration privileges often have more functionality available, e.g. links to display technical information for a printer, whereas normal users only see the name of the printer. For example, admin users should be presented with a link for display of administrative information for the printer:

L36
```
<div>Selected printer: <a href=...>HP LaserJet 4200</a></div>
```

while normal users should only have the plain printer name:

L37
```
<div>Selected printer: HP LaserJet 4200</div>
```

Creating this output using plain element conditionals requires two different element subtrees:

L38
```
<div>Selected printer:
        <a id=PrinterInfoLink href=...>HP LaserJet 4200</a> <!-- for admins -->
        <span id=PrinterName>HP LaserJet 4200</span> <!-- for normal users -->
</div>
```

and the servlet needs to specify whether the `<a>` element or the `<span>` element should be output:

L39
```
@EIf("PrinterInfoLink") @EIfNot("PrinterName")
boolean testIsAdmin() {
    return user.isAdmin();
}

@EText({"PrinterInfoLink", "PrinterName"})
String getPrinterName() {
    return ...;
}
```

This is tedious and damages the visual editing of the html template as the printer name is displayed twice when the template is edited: Firstly as a link and secondly as plain text. Fortunately, the `@ETagIf` annotation makes this much simpler and prettier. The improved template does not contain the `<span>` element but only the `<a>` element and the template system suppresses the link *tag* (but not the content) if the user is not an admin. The template is much simpler:

L40
```
<div>Selected printer: <a id=PrinterInfoLink href=...>HP LaserJet 4200</a></div>
```

and the servlet just uses `@ETagIf` for the conditional:

L41
```
@ETagIf
boolean testPrinterInfoLink() {
    return user.isAdmin();
}

@EText
String getPrinterInfoLink() {
    return ...;
}
```

The output is one of the two lines below. If `testPrinterInfoLink()` returns true, the first line is output, otherwise the second line is output:

L42
```
<div>Selected printer: <a href=...>HP LaserJet 4200</a></div>
<div>Selected printer: HP LaserJet 4200</div>
```

This makes it much easier to add links for admin users without making the html template or java servlet overly complex.

5.3 Attributes

Simple attributes like 'value' are very similar to element content using `@EText`, so this test focuses on more complex attributes and sub-attributes. Assume the html snippet below is placed inside a `<li>` list item or a table row:

L43
```
<input type=radio class=PrinterRadio>
<span class=PrinterName style='font-size: 90%'>HP LaserJet</span>
```

The requirements of the servlet are: The `radio` elements' `value` and `checked` attribute must be specified with the proper values for the printer. The `<span>` needs the content set and needs the style attribute updated so that it displays the name in gray if the printer is unselectable. If the printer is a color printer, the span must have `ColorPrinter` specified in the `class` attribute in addition to `PrinterName`.

The code for the servlet is:

L44
```
@TCompile
class DesignTest3 {
    Printer p = ...;
    static class Printer { String name; int id; boolean selected, selectable, color; }

    @AValue String nPrinterRadio() { // sets value=xxx in <input...>
        return Integer.toString(p.id);
    }

    @AChecked boolean cPrinterRadio() { // might add 'checked' to <input...>
        return p.selected;
    }

    @AEnabled boolean ePrinterRadio() { // might add 'disabled' to <input...>
        return p.selectable;
    }

    @EText String tPrinterName() { // sets content of <span class=PrinterName>
        return p.name;
    }

    @SValue(styles = "color") String cPrinterName() {
        return p.selectable ? "black" : "gray"; // sets color=xxx inside 'style' attribute
    }

    @SHasClass(classes = "ColorPrinter") boolean pPrinterName() {
        return p.color;                  // might add ColorPrinter to class='...'
    }

    // main-method etc...
}
```

Specifying attributes is just as easy as specifying the element content. Even complex sub-attributes such as class and style are easily updated with dynamic values as the java programmer doesn't have to take into account what else the web designer has specified.

The system is very robust when it comes to revising designs: Suppose that the `<span>` element didn't have a `style` attribute in the initial design. If the java programmer is using a template system which doesn't support sub-attributes, he would typically generate the complete `style` attribute value. When the web designer revises the design and adds the `font-size` rule, it is simply ignored—which probably surprises the web designer. However, as this template system supports sub-attributes, all style rules except `color` are kept as-is. The same advantage exists for the `class` attribute: It doesn't matter if the web designer manually adds the `ColorPrinter` class to check the layout as the template system automatically suppresses it—something which is quite difficult to obtain with macro-based template systems. Only the rules which the programmer explicitly overrules are changed, thereby keeping the suprises at a minimum: The template system is more robust for collaboration development of web applications.

Another aspect worth noting is the seamless transition from simple getter methods (e.g. `tPrinterName`) to more complex methods (e.g. `cPrinterName`) which dynamically calculate the value. Unlike JSP and many other macro-based template systems, there is no transitioning step when the value has to be calculated. Some might argue that all those methods are overhead, and that it should be replaced with a specification of which datastructures contain the desired values—often as an xml specification. My counter-argument is that a non-java format has all the disadvantages of the prototype (and adds a transition step when values have to be calculated) and would probably take almost the same amount of 'code' anyway.

The only negative aspect is the need to manually convert the printer's integer `id` value to a `String`. This test shows that it makes sense to allow `@AValue` to be applied to a method returning an integer—but not boolean or float or other simple types which must typically be formatted.

5.4 Simple loops

The example below displays the list of printers available to the user:

L45
```
<div>You have access to the following printers:</div>
<table>
    <tr class=PrinterRow>
        <td class=PrinterName>My printer</td>
    </tr>
</table>
```

The java code:

L46
```
@TCompile
class DesignTest4 {
    List<Printer> accessiblePrinters = ...;
    static class Printer { String name; int id; }

    int printerIndex = -1; // current list index into accessiblePrinters

    @EWhile
    boolean testPrinterRow() {
        return ++printerIndex < accessiblePrinters.size();
    }
```

```
    @EText
    String getPrinterName() {
        return accessiblePrinters.get(printerIndex).name;
    }

    // PrintWriter, main-method, ... as previous examples
}
```

This example shows that iterating thru simple structures requires very little code—just a single method and list index. However, if the iteration was done using an `Iterator`, it becomes slightly more complex:

L47
```
@TCompile(filename = "DesignTest4") // same html as previous example
class DesignTest5
{
    List<Printer> accessiblePrinters = ...;
    static class Printer { String name; int id; }

    Iterator<Printer> iter = ...;
    Printer currentPrinter;

    @EWhile
    boolean testPrinterRow() {
        if (!iter.hasNext())
            return false;
        currentPrinter = iter.next();
        return true;
    }

    @EText
    String getPrinterName() {
        return currentPrinter.name;
    }

    // PrintWriter, main-method, ...
}
```

The class scope is polluted with variables used only in the loop scope: `currentPrinter` and `iter`. If the page has several complex loops, the programmer easily ends up with many temporary variables which are easy to mix up and therefore often have their named prefixed with the name of the looop. This often makes the code harder to read due to overly long variable names or names that are too easy to mistake for another name.

E.g. assume that (1) calculating the price for using the printer requires a temporary variable and (2) the user can select a default printer. This would add a `currentPrinterCost` variable for the former, and a `selectedPrinter` variable for the latter. The programmer can by accident write `currentPrinter` instead of `selectedPrinter` without the java compiler being able to catch the bug. If the variable is used before the loop, the variable is `null` and results in a NullPointerException. If the variable is initialized, it can by luck point to the correct printer and thus result in the "it works for me, so there is no bug" situation which often leads to time consuming debug sessions.

The next section shows how scopes are able to solve this problem as they enable `currentPrinter` to be renamed to `p` and `currentPrinterCost` to `cost`. The variables are confined to the inner class which enables the java compiler to catch all usage of the variables outside their valid scope.

5.5 Complex loop using scope

This example implements the same printer list as the previous example but uses an inner class for scoping:

L48
```
@TCompile(filename = "DesignTest4") // see previous section for the html template
class DesignTest6 {
    List<Printer> accessiblePrinters = ...;
    static class Printer { String name; int id; }

    @EIterate
    Iterable<Printer> iterPrinterRow() {
        // List<x> implements Iterable<x>, so the list can be returned as-is.
        // The declaration of the list (i.e. the field) could have been annotated,
        // but a method is usable even if the field is declared in a superclass.
        return accessiblePrinters;
    }

    class EmitPrintRow {
        final Printer p;

        EmitPrintRow(Printer p) {
            this.p = p;
        }

        @EText
        String getPrinterName() {
            return p.name;
        }
    }

    // main-method etc.
}
```

The above example shows the basic structure but is too simple to really show how much better this approach is to handle complex loops. The `currentPrinter` variable is scoped to the inner class which makes it easier to use a simple name: `p`. There is no servlet-programming overhead to handle the iteration or instantiating the inner class besides declaring the class.

5.5.1 A more complex list

The following example tries to show the advantage of using inner classes when the data needed for each list row are more complex. The html template is:

L49
```
<form>
    <div>You have access to the following printers:</div>
    <table>
        <tr>
            <th>Default<br>printer</th>
            <th>Official name</th>
            <th>Nickname</th>
            <th>Prints left</th>
        </tr>
        <tr class=PrinterRow>
            <td><input type=radio class=PrinterSelected name=selected></td>
            <td class=PrinterName>rub-26-1-prt-2300</td>
            <td><input type=text class=PrinterNickname value='library'></td>
```

```
                <td class=AccountStatus>42</td>
            </tr>
        </table>
        <input type=submit value='Update'>
</form>
```

The java code:

```
L50  class UserPrinterAccount {
        // many-to-many relation between user and printer:
        //          user-specific information for a single printer
        int balance;
        String nickname; // null if no user-specific name for the printer.
     }

     class User {
        String fullname, username;
        UserPrinterAccount defaultPrinter;
     }

     @TCompile
     class DesignTest7 {
        User user = ...;

        @EIterate("PrinterRow")                    // see previous example
        List<Printer> accessiblePrinters = ...;

        static class Printer { String name; int id; }

        class EmitPrinterRow {
            final Printer p;
            final UserPrinterAccount acc; // null if no account

            EmitPrinterRow(Printer p) {
                this.p = p;
                this.acc = findAccountUsingDatabaseQuery(p);
            }

            @AChecked
            boolean isDefaultPrinter() {
                return acc != null && acc.equals(user.defaultPrinter);
            }

            @AEnabled
            boolean allowDefaultPrinter() {
                return acc != null;
            }

            @EText
            String getPrinterName() {
                return p.name;
            }

            @AValue
            String getPrinterNickname() {
                return acc != null ? acc.nickname : null;
            }
```

```
        @AName
        String namePrinterNickname() {
            return "nick-" + p.id;
        }

        @EText
        String getAccountStatus() {
            return acc != null ? Integer.toString(acc.balance) : "-";
        }
    }
}
```

5.6 Specification of templates

As seen in the previous examples in this chapter, if only a single html file is used and its basename is the same as the simple class name, a simple `@TCompile` is sufficient. If the html template is named differently, a file name argument is needed:

L51
```
@TCompile(filename = "DesignTest4")
        // this makes it use DesignTest4.html instead of DesignTest5.html
class DesignTest5 {
    ...
}
```

5.6.1 Template fragments

A common use of template fragments is ajax and self-refreshing `<iframe>`. Splitting the 2 parts into 2 separate html templates makes the visual design difficult. The example below shows the user's current list of print jobs. The template is designed as a plain single page without any refreshing, however, the `<table>` part should be refreshed using an `<iframe>`.

L52
```
<div>You're logged in as <span id=PersonFullName>John Doe</span>.</div>
<table id=JobTable>
   <tr class=JobRow>
      <td class=JobName>My document</td>
   </tr>
</table>
<div>Some more information here.</div>
```

The servlet to implement the 'outer' page is:

L53
```
@TCompile(filename = "DesignTest8")
class DesignTest8Page {
    @EHtml
    void printPersonFullName() {
        pw.print("Elliott Smith");
    }

    @EReplace
    void replaceJobTable() {
        pw.print("<iframe src=... width=... height=...></iframe>");
    }
}
```

The `replaceJobTable` method replaces all of the `<table>` subtree with an `<iframe>` which loads its content using the servlet below. This servlet implements the `<iframe>` and is very similar to the example in section 5.5 on page 37:

L54
```
@TCompile(filename = "DesignTest8", fragment = "JobTable")
class DesignTest8IFrame {
    static class PrintJob { String name; }

    @EIterate("JobRow")
    List<PrintJob> printjobs = ...;

    class EmitPrinterRow {
        final PrintJob job;

        EmitPrinterRow(PrintJob job) {
            this.job = job;
        }

        @EText String getJobName() {
            return job.name;
        }
    }
}
```

The template for the latter servlet emits only the `<table>` html fragment without any html header section or `<body>` tag. The user can add such header (e.g. a doctype) when the fragment is used for an `<iframe>`. When updating the page using Javascript's innerHTML (e.g. ajax), no such additional header must be added.

This example shows that it is easy to split a template into several parts so that the web designer can design the page using visual tools and the web application can serve the parts without needing to break the template into bits by a pre-processor. The overhead is very small—only the declaration of the `replaceJobTable` method, an overhead which is difficult to avoid as the `<iframe>` has to be defined somewhere.

5.7 Conclusion

The main conclusion is that using annotations has very little coding overhead and integrates very tightly with the java code. The specification of the dynamic parts of the template is located very close to the implementing method/field in the java class. The "convention over configuration" approach has been very successful when it comes to reducing the overhead of most specifications to a parameterless annotation, e.g. the method/field name is in most cases reused for the html `id/class` attribute name instead of requiring yet another name.

The support for scopes using inner classes solves the problem of having the global scope polluted with temporary variables—a significant benefit for complex pages that is obtained with minimal overhead (as L50 on page 38 demonstrates).

The tight integration is also the main disadvantage as it might require two-step compiles when annotated fields/methods are removed as the generated java class still refers to them. The template system must therefore support some method of disabling/out-commenting those parts of the generated code as the main java class cannot be compiled until the generated class is updated—which requires the main class to be compiled, a catch-22.

The only minor usability problem which the test revealed was the need to manually convert an `int` to a `String` in `nPrinterRadio()` in L44 on page 34. It seems like a good idea to allow some—but probably not all—annotations to allow `int/Integer` in addition to String. This must be considered for future versions of the template system.

Chapter 6

The compiler

Rather than performing the full processing of the template at runtime, the template system is split into a compiler and a runtime part. This chapter describes the implementation of the compiler.

6.1 Introduction

The compiler translates the html template into a format suitable for fast execution at runtime. The input to the compiler is the (servlet) java class and the html template. The output is a generated java class (for accessing the servlet at runtime) and pcode/string table (for execution of the compiled DOM).

The compiler differs in two major ways from traditional compilers such as C and pascal compilers:

1. The input consists of two files in different languages which have to be combined. It is not a serial list of files in the same language.[1]
2. A significant part of traditional compilers is tokenizing and parsing the input into abstract syntax trees. The template compiler uses the JVM to extract annotations from java `.class` files and parses html files by using the JTidy[2] library. This part of the compiler is therefore only a few simple lines of code. The difference from traditional compilers is not that this part doesn't exist, but that the implementation is so small and simple that the interesting parts of the compiler are in all the other areas.

The compiler is implemented in java because the JVM provides easy access to the annotations using reflection and because it makes it easier to support dynamical partial recompilation of templates which are changed while the web application is running. This enables the template system to allow the web designer to tinker with the layout without requiring a compilation and restart of the server for each change.

[1] The java compiler (javac) reads two different kinds of input files (source and `.class` files), however, they contain (almost) the same information (just 'encoded' differently) so javac matches the traditional model. It is common for traditional compilers to generate several heterogenous *output* files (e.g. object files and symbol files with debugging information).

[2] http://jtidy.sourceforge.net/

6.1.1 Interfacing the build system

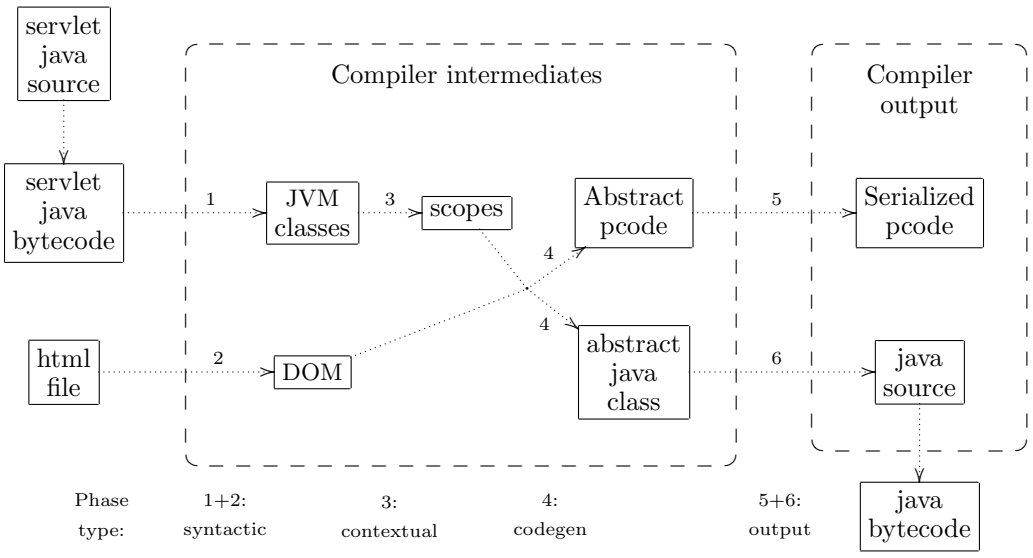
The build system calls the compiler with name of the input java class, the path of its source file and the root directory of output files. The source path is used to load html templates. The compiler returns a list of generated output files and dependent input files to the build system. This enables the build system to detect when templates must be recompiled.

A more fine-grained detection is to check if the annotations and method/field declarations have changed in the input java class. This has not been implemented as it still requires loading the java class and parsing the annotations—which takes up a significant amount of the total compilation time.

6.2 Phases

JTidy and the JVM introduce the first intermediate representations of the source files. The compiler uses additional phases because it is difficult to generate the output formats directly from these initial intermediate formats and because it is easier to implement the compiler when it is broken into smaller well-defined chunks.

The figure below shows the input, intermediate and output formats represented by square boxes. The labelled arrows show the processing phases:[3]



Phase 1 loads the compiled servlet class. Described in section 6.3 on page 43.

Phase 2 loads the html template into a DOM using JTidy. See section 6.4 on page 43.

Phase 3 creates the scope tree which represents the java class and some of its inner classes. Described in section 6.5 on page 44.

Phase 4 generates the pcode and java class from the DOM and the scopes. Described in section 6.6 on page 46.

[3]The output phases (5+6) would normally [ASU86] be called 'codegen' and phase 4 'intermediate codegen'. I've chosen to use the simple 'output' name for phases 5+6 as the phases have very little complexity compared to phase 4, esp. compared to a x86 C-compiler's very complex conversion of the intermediate format into optimized machine-code.

Phase 5 writes the serialized pcode from the abstract representation. This is a very simple phase as it is just a manual serialization of a java class. Consequently, it is not described further.

Phase 6 writes the java source file from the abstract representation. This is a simple phase that is described in section 6.8 on page 50.

Phases 3 and 4 are described in greater detail because they are the complex and major phases in the compiler. The reason for having phases 5 and 6 is that abstracting the output formats simplifies phase 4 and because the format of the output files can't be generated serially, e.g. it is not possible to declare new global variables while writing the body of a method and it is not possible to add `import` statements inside the class declaration either.

6.2.1 Implementation details

Phase 1 is implemented in the build system (see section 6.3 below).

Phase 2 is implemented in the dk.vitality.util.jat.translator.compiler.htmlLoader package. Source: appendix H on page 101.

Phase 3 is implemented in the dk.vitality.util.jat.translator.compiler.application package. Source: appendix F on page 73.

Phase 4 is implemented in the dk.vitality.util.jat.translator.compiler.codeGeneration package. Source: appendix G on page 92.

Phase 5 is a single method in the TemplateData class in the dk.vitality.util.jat.runtime.lib package. Source: appendix D.3 on page 68.

Phase 6 is implemented by the ProgramEmit class and a single method in the ProgramImpl class in the dk.vitality.util.jat.translator.compiler.codeGeneration package. Source: appendix G.9 on page 94 and appendix G.10 on page 98.

6.3 Loading the compiled servlet class, phase 1

This is an extremely simple phase as it consists of a `Class.forName(...)` call which loads the class without initializing it (i.e. the static initializers are not run).

The build system scans the `.java` file hierarchy instead of a `.class` hierarchy because some development tools writes the compiled classes directly to a `.jar` archive without creating `.class` files. As a single `.java` source file contains the code for the outer class and all its inner classes, the build system must use the modification date of this file for all inner classes. The build system therefore needs to know which inner classes that a `.java` file defines. Because the compiler needs the loaded `java.lang.Class` anyway, the build system loads it—i.e. this stage is (currently) performed by the build system.

6.4 Parsing the html template, phase 2

The parsing of the html template is performed using the JTidy library which reads the html file and produces a DOM tree as a result. The compiler clones the tree into a Xerces based

implementation because JTidy uses its own very primitive, broken and non-conforming DOM implementation.

After the html template has been loaded into a DOMtree, it is possible to manipulate it by subclassing the main compiler class. This feature can be used to e.g.

- modify links (in e.g. `<a>`, `<script>`, `<style>`) to handle differences between the URL design and the templates path hierarchy.
- change the layout by adding/removing/changing the html. This feature is used in print.ruc.dk to add the site menus/logo so that the individual html template files only contain the specific content part of the page. This makes it easier to change the overall site design and keeps the size of the individual template files small.

6.5 Scopes, phase 3

The DOM tree is traversed in phase 4 to generate the output for each node in the tree. This is similar to plain DOM-to-text serialization of html documents except that the template compiler's output is more complex: It consists of static text intermixed with callbacks specified by the annotations that target the particular element. Because each element is processed one-by-one, phase 4 must be able to quickly find the list of annotations that target the current element. Phase 3 creates the scope tree: the datastructure which provides fast lookups.

Each scope represents one java class. The root scope is the outermost java class. A sub-scope is created for each inner class that:

- is annotated by `@EScope`
- or is referred to by the `scope` parameter of `@EIterate`/`@EScopingWhile`.
- or is annotated by `@TCompile` or encloses an inner class annotated by `@TCompile`.

Sub-scopes are not created for other inner classes to avoid making erroneous warnings.

Annotations can target elements using the `id` attribute, the `class` attribute and the tag name in any combination (see section 4.1 on page 20). The scope contains 3 maps from the id/class/tag value to the application of the annotation. If multiple id/class/tag names are specified for an annotation, the annotation is added to the map for each specified name.

Finding the annotation-applications for an element is performed by 3 steps. The result is the combined list of applications of all 3 steps:

- `id` attribute (if any): a simple lookup in the id map.
- `class` attribute is a white-space separated list of class names. A lookup in the class map is performed for each name and the result is the combined list.
- tag name: a simple lookup in the tag map.

6.5.1 Application class hierarchy

The application of an annotation (i.e. specifying it on a specific field/method/...) is unfortunately called 'an annotation'. Therefore the name 'Application' has been used in the implementation to avoid overloading the 'annotation' name. The relationship annotation/application corresponds to class/object in oop: an 'application' is an instance of an annotation applied to a specific field/method/...

The JVM uses the `java.lang.reflect.Annotation` class to represent the application of an annotation in a loaded class. The compiler encapsulates it in an `Application` class because the compiler needs to keep track of annotations which are not used by the html

template and to handle extracting the 4 element targeting parameters in one single place.[4] A single application of an annotation can match several html elements. As each match requires state information (e.g. pcode addresses which need to be patched), a match is represented by an `AbstractEmitter` subclass.

A main design principle for implementing the compiler is to use OO to represent different types of annotations (both value types and action types) instead of hardcoding it as one huge phase 4 method. This results in the need for a class hierarchy for each of the 2 types:

1. One hierarchy which specializes on the *value type* of field/method the annotation is applied to, e.g. `boolean`, `int`, `String`. This centralizes checks for field/method types, generation of pcode to call the methods/fields at runtime and getting the sub-scope for the element subtree (if any). This hierarchy primarily handles how the annotations are applied and therefore 'belongs' to phase 3. The base class is `Application` which has subclasses such as `AbstractStringApplication` and `AbstractIntegerApplication`.

2. One hierarchy which specializes on the *action-type*: how the annotation modifies the html element (is it a conditional, does it generate the content, an attribute, a sub-attribute, etc). This hierarchy is used by phase 4 to generate (emit) the output for the element using strictly typed lists of actions for an element and makes the central emitting methods small as all details are delegated to this class hierarchy. The base class is `AbstractEmitter`. Most sub-classes are defined as inner-classes in the `Application` that creates the emitter.

The scope must return the appropriate instance of the specialized `Application` class. Unfortunately, this cannot be done by the (pseudo) code below as java only permits to switch on simple types:

L55
```
Application get(Annotation a) {
    switch (a.annotationType()) { // annotationType() returns the @interface java.lang.Class
        case EHtml: return new EHtmlApplication(...);
        case EHtmlPre: return new EHtmlPreApplication(...);
        case SDisplay: return new SDisplayApplication(...);
        ....
    }
}
```

The problem is solved by using the trick to generalize the switch statement to a map of lambda expressions. However, java doesn't support lambda expressions, so the factory pattern is used instead:

L56
```
interface ApplicationFactory {
    Application create(Annotation a);
}

class AnnotationApplicationMap { // simplified
    Map<Class, ApplicationFactory> map = new ...;
```

[4] Because annotations can't inherit from interfaces, the extraction is performed using reflection as it would otherwise require specialized code for each of the 70 template annotations!

```
    AnnotationApplicationMap() {
        map.put(SDisplay.class, new ApplicationFactory() { // adds a 'case' factory
            Application create(Annotation a) {
                return new SDisplayApplication(a);
            }
        });
        // many more factories here, one for each template annotation (70 total)
    }

    Application get(Annotation a) {
        return map.get(a.annotationType()).create(a); // this executes the switch statement
    }
}
```

This makes it easy and fast to obtain the specific `Application` instance for an instance of the `Annotation` class returned by the JVM. The drawback is the tedious declaration of all the factory classes.

6.5.2 Creating the scope tree

The scope tree is created by a recursive algorithm which processes the classes fields, methods and inner classes using reflection. To be able to make proper warnings about overridden annotations, the algorithm is split into 2 parts/phases so that all annotations on the class' fields and methods are processed before the subscopes are processed. Parsing is started with the root scope. The first part is the steps below:

1. If `@TIgnore` is present, return.
2. Recursively parse super classes.
3. Process annotations on fields not marked with `@TIgnore`.
4. Process annotations on methods not marked with `@TIgnore`.

The second part:

5. Post-process `@EIterate` and `@EScopingWhile` annotations by creating a sub-scope by executing the full algorithm on the specified inner class.
6. For each inner class marked with `@EScope` but not with `@TIgnore`, create a sub-scope by executing the full algorithm.

Finally, if the `@TCompile` is placed on an inner class, the Inner class and all its enclosing classes are processed using the above algorithm, starting with the outermost class.

6.6 Code generation, phase 4

The code generation is performed as simple recursive processing of the DOM tree. Codegen for an element is independent of (annotations for) its children and a parent element can only change the processing of its children in two ways: (1) it may skip the children (e.g. due to `@EHtml`) and (2) it may make the children use a subscope (e.g. due to `@EScoped`). The recursion and main processing of the elements are performed by the `TreeEmit` class. To manage the emitters, an instance of the `ElementEmitters` helper class is created for each html element: The role of `ElementEmitters` is to find out what should be done for an element, whereas the role of `TreeEmit` is to perform the actions.

`ElementEmitters` obtains a list of matching annotation `Application`s from the current scope and each `Application` creates an instance of a subclass of `AbstractEmitter` to

handle codegen for the element. The list below represents all the possible modifications of an element at runtime. The emitter adds itself to the appropriate list or sets the singleton instance. `ElementEmitters` checks that no conflicting emitters are added (e.g. `@AStyle` in addition to `@SDisplay`) and that only one 'element singleton' is created for each type:

- list: element conditionals, e.g. `@EIf`
- list: tag conditionals: `@ETagIf`
- list: content conditionals: `@EContentIf`
- list: pre/post-content, e.g. `@EHtmlPre`
- element singleton: looping, e.g. `@EWhile`
- element singleton: replacing element: `@EReplace`
- element singleton: scope: `@EScope`
- element singleton: content, e.g. `@EHtml`
- list: simple attributes, e.g. `@AValue`
- list: post-content attributes, e.g. `@AClassPost`
- list: sub-attributes, e.g. `@SDisplay`

In addition to the above, `ElementEmitter` contains:

- the scope used for the children
- flag indicating if the `id` attribute must be suppressed: for elements inside loops, it is suppressed to avoid duplicate definitions of the same id: They are inhibited for children of html elements that use a loop emitter.
- flag indicating if the element should be processed at all: false iff it matched an `@EZap`.

6.7 The generated output

This section describes the generated output to provide a base for understanding the output parts of the compiler, i.e. phases 5, 6 and (implementation parts of) phase 4.

The output class hierarchy (for implementing the two "abstract pcode" and "abstract java class" data structures in the figure on page 42) is designed using the Facade Design Pattern [GHJV95]. The reason/consequences are:

1. It provides the emitter classes with a higher-level interface which is much more suitable for the emitters than the lower-level data structures.
2. Thus, a larger degree of separation between the emitter classes and the output data structures.
3. It enables simple validation of the generated output (e.g. no undefined references).
4. However, the facade and implementation provide both the "abstract pcode" and "abstract java class" data structures. Classes which implement the facade pattern and provide a significant part of the implementation can easily evolve into the blob anti-pattern[5] over time.[BMMIM98]
5. The combination of both the output data structures enables automatic joining adjacent static output strings while still keeping the required splits due to e.g. callbacks. An alternative approach is to add a phase which post-processes the pcodes and optimizes adjacent strings. This requires analyzing all goto-destinations to detect strings that can't be joined.

Using the facade pattern enables simpler code in the emitter classes and removes the need for a pcode-optimizing phase at the cost of making the high-level design of the classes less clean.

[5] An anti-pattern is a pattern with a negative effect, e.g. spaghetti code.

The facade is provided by the `Program` interface, whereas the `ProgramImpl` class provides the implementation. The interface has methods for static text output, creating forward and backward gotos, getting field values in the servlet class, calling methods in the servlet class and entering/leaving scopes.

6.7.1 Generated java code

The generated class contains 4 major parts:

1. references to the scopes, i.e. class instances. The programmer passes the root-scope instance as a parameter to the constructor. It also contains a reflection reference to the constructor if the instance can't be created directly (e.g. inner classes without provided newMyClass() methods).

2. temporary variables for iterators, i.e. reference to the iterator and the current object.

3. an `executeCallback(int idx)` meta-method which contains the generated code to call methods, access fields and create inner class instances in the programmer's class. Instead of generating a method for each function, it is dispatched thru a `switch(idx)` statement so that the template runtime can execute the generated code using a single method.

4. by inheritance from `TemplateBase`, it contains the 3 registers in the pcode runtime, the variables: String stringValue; int intValue; boolean boolValue. Whenever the `executeCallback()` method accesses a field or executes a non-void method, the result is stored in the appropriate register variable. The boolValue variable is used for conditional gotos.

With a few non-interesting parts removed, the generated code for the PrinterListNick example (see L50 on page 38) is:

L57
```
class PrinterListNickTp extends TemplateBase
{
    PrinterListNickTp(PrinterListNick scopeRoot) {
        this.scopeRoot = scopeRoot;
    }

    void emit(HtmlPrintWriter pw) throws Exception {
        if (v1cnstr == null) {
            v1cnstr = PrinterListNick.EmitPrinterRow.class.getDeclaredConstructor(
                PrinterListNick.class, Printer.class);
        }
        super.emit(pw);
    }

    // scopes:
    final PrinterListNick scopeRoot;
    PrinterListNick.EmitPrinterRow scopePrinterRow;

    // callback fields for @EIterate Iterable iterPrinterRow() in PrinterListNick
    Iterator<Printer> v1iter;
    Printer v1item;
    static Constructor<PrinterListNick.EmitPrinterRow> v1cnstr;
```

```
    void executeCallback(int idx) throws Exception {
        switch (idx) {
            case 0: stringValue = scopePrinterRow.getAccountStatus(); break;
            case 1: stringValue = scopePrinterRow.getPrinterName(); break;
            case 2: stringValue = scopePrinterRow.getPrinterNickname(); break;
            case 3: stringValue = scopePrinterRow.namePrinterNickname(); break;

            case 4:
                {
                    Iterable<Printer> able = scopeRoot.iterPrinterRow();
                    if (able == null) {
                        throw new NullPointerException(
                            "PrinterListNick.iterPrinterRow returned null");
                    }
                    v1iter = able.iterator();
                    if (v1iter == null) {
                        throw new NullPointerException(
                            "PrinterListNick.iterPrinterRow.iterator() returned null");
                    }
                }
                break;

            case 5: scopePrinterRow = v1cnstr.newInstance(scopeRoot, v1item); break;

            case 6:
                boolValue = v1iter.hasNext();
                if (boolValue) {
                    v1item = v1iter.next();
                } else {
                    v1item = null;
                    v1iter = null;
                }
                break;

            case 7: scopePrinterRow = null; break;
            case 8: stringValue = scopeRoot.getPersonFullName(); break;
            case 9: stringValue = scopeRoot.getUsername(); break;
            case 10: boolValue = scopeRoot.testHasPrinters(); break;
            default: throw new Exception("Unknown callback idx = " + idx);
        }
    }
}
```

The cases in the switch statement are:

0..3: Call simple `String xxx()` methods and store the return value in stringValue which is used by the pcode.

4: Call the `@EIterate Iterable<Printer> iterPrinterRow()` method to obtain an `Iterator<Printer>`.

5: Create an instance of the EmitPrinterRow inner class using reflection as no instance-creating method has been supplied and inner classes need the hidden 'this' argument for its enclosing class.

6: Get the next object from the iterator, if any. Update `boolValue` with the state as it is used in a conditional goto to the start of the loop. If no more objects are available, references are nulled to catch erroneous usages of the variables outside the scope.

7: Null the reference to the scope created in case 5. This is executed when leaving the scope.

8..9: Call simple `String xxx()` methods.
10: Call testHasPrinters() to test if the subtree should/should not be included.

Section A.4 on page 59 shows how the callbacks are used from the pcode.

6.8 Serializing generated java source, phase 6

The input to the source generating class `ProgramEmit` is a list of the scopes/iterator info and a list of the callbacks. It uses two utility classes: `JavaClass` which handles overall class definition (e.g. package imports, superclass and implemented interfaces) and JavaPrinter which is a `PrintWriter`-like class which handles simple Line indention for blocks and if/for/while statements.

Besides simple straight-forward codegeneration, `ProgramEmit` handles:

- detecting how to call static methods versus instance methods,
- declaration of and obtaining reflection constructors for inner classes
- declaration of variables for iterators, obtaining an `Iterator<>` from an `Iterable<>`, and getting next object from the iterator.

6.9 Conclusion

Using the JVM to parse the java class (and JTidyto parse html) removed almost all the complexity and code from the parsing phase of the compiler. After this description of the compiler, the remaining work is to test the implementation for bugs.

Chapter 7

Implementation test

For completeness, this chapter describes how the template system is tested.

The general purpose of testing is to ensure something works. For compilers, the definition of 'works' includes both producing the correct output for valid input and to produce warnings/error messages for problematic/invalid input. This means that the testing must include tests for both valid and invalid input, and they must check both the generated output and the generated warnings/error messages.

7.1 System tests versus unit tests

The template system consists of two parts: the compiler and the runtime. The testing can be performed by testing either the individual parts (unit tests) or all parts combined (system tests). The difference between the testing methods is only if they test the complete system or a part of it, but not how the test is performed.

Unit testing simplifies testing and debugging as each individual part is examined without the complexity that arises from interacting with other parts, and because it often is easier to produce tests for a single unit than for a complete system. For the template system it is easier to perform system testing as the runtime format (pcode and generated java class) is not easy to validate. It is easy to provide basic validation of the generated java source as it must be compilable by the java compiler, but errors at a higher level is difficult to test for because many different runtime formats are valid. Some validation can be performed by manual inspection of the generated java source.

Testing the full system (i.e. compiler and runtime) as a single entity is a much better approach as both the input and the output are well defined (i.e. chapter 4). A system test package consist of a java class (servlet), an input html file and files which contain the expected html output and expected warnings/errors. Performing the test consists of compiling the template (and the produced java class), running the servlet and checking that the generated output matches the expected output.

7.2 Limitations of testing

The major problem with validating software using tests is the number of tests needed to prove the software works correctly for all valid and invalid input. To test all permutations

of inputs for a simple 32-bit integer adder, 2^{64} different inputs must be tested. Even if 100 billion permutations were tested each second, it would take almost 6 years to run the tests.

The input to the template compiler is complex: It has 71 element targeting annotations which can match none, a single or multiple html elements. The element targeting annotations and the 2 compiler directive annotations (`@TIgnore` and `@TOverrides`) can be specified on fields, methods and/or inner classes. Each of the 71 element targeting annotations have at least 4 parameters. The huge number of permutations of the input (html template and annotations) makes it impossible to perform testing of all valid and invalid input. Only a very small subset can be tested.

7.3 Writing tests

There are two major methods for designing tests:

automatically generated tests. Input is generated by a (more or less) random generator and the testing framework has to determine if the template compilers output is correct for the actual input. Testing the correctness can be performed by (1) implementing another template compiler inside the testing framework (expensive, and prone to the same programmer bugs), or (2) only checking aspects of the output, e.g. if the compiler reports errors and/or warnings.

Furthermore, a plain random generator produces input where all input permutations are equally likely. As the number of important corner cases is very small compared to the full input space, they are seldom tested, if at all. This means that most of the performed tests are redundant. The solution is to use test generators which focus on producing test inputs which tests the corner cases, but creating such test generators is difficult.[AHLL06]

manually written tests. The implementor designs a number of tests, typically to cover both simple cases (to provide easy testing and debugging of basic functionality) and corner cases. Writing good test cases which covers important corner cases is surprisingly difficult.[Mye79]

Often a combination of the above tests are used, e.g. manually written tests to ensure correct operation for valid input and random tests to ensure the software doesn't crash for invalid input.

For the template system, it is difficult to use automatically generated tests to find a significant number of bugs because it is difficult to implement a method to validate the generated output. The implementation is tested using manually written tests due to the limited time available for this project. Furthermore, only positive tests are implemented (no testing of invalid input) and only the output html is tested (warnings are ignored).

The tests are written as typical web implementation design patterns similar to the examples presented in chapter 5. The first test is a very simple "Hello, World" to provide the most basic debugging example. The other tests (moreContent, simpleAttributes, attributeCollections, hasClass, styles, conditional, conditionalContent, conditionalTag, simpleLoop, scope, scopingWhile, iterate and splitTemplate) all focus on a major group of annotations and often use several combinations of the annotations in one test to cover any problem with the implementation of the annotation. The drawback of using complex group tests only is the more difficult debugging when the output is incorrect and the lack of group interdependency testing.

A sample test (including expected output) is provided in appendix B.

7.4 Conclusion

System tests are well-suited for ensuring that the template compiler implements the design correctly because the input and output files are well-defined and the translation does not depend on anything else—there is no user interaction or network timing involved introducing unpredictable or non-validable values and the expected output from the template compiler is easy to specify. The major problems are (1) the difficulty in implementing good test cases, and (2) the amount of time required for implementing the tests—something which is often considered being a problem or waste of time—but which is probably more an indication of the complexity of the tested software and of testing.

Chapter 8

Conclusion

This report tries to evaluate the suitability of using java annotations in an html template system to specify the dynamic parts. The method used to answer the question is to build such a system and to test it using common implementation patterns in web design.

The design test chapter concluded that annotations create very concise specifications and the answer to the above question is therefore a strong acknowledgement of the suitability. Furthermore, the implementation and unit test chapters show that it is possible to implement such a compiler without major problems and that it actually works. The only negative aspect compared to the prototype is the need for the template compiler to be able to load the servlet class. This minor problem can be solved by using a dedicated class loader or a bytecode library for reading the class definitions.

However, the material in this report only allows validation of the suitability from a design-wise perspective: It can conclude the annotation-based system is very well-suited for implementing the small web-design examples, however, it cannot tell if the system is suitable for implementing complex pages nor if it is suitable for complex web applications. As the real purpose of the tool is to be usable for such applications, answering the extended question of suitability is therefore interesting.

After the compiler was implemented and before writing most of this report, I upgraded two web applications from the xml-based prototype to the annotation-based system. One web application[1] consists of 1 very complex page and 2 derivations thereof (e.g. printfriendly version). The other web application[2] consists of 6 pages of which 3 are moderately complex due to a lot of conditionals and 1 html email.

My experience is that the new system makes it much easier to maintain the pages: Several dead methods were found because the annotations make it very obvious if a method is used by the template system or not. Writing new servlets was easier too because the template information is embedded in the source next to the implementation and because screen real estate is not wasted on an xml document—a reducement from 3 to 2 open windows is large in relative terms.

The only negative experience is the need for the 3-step procedure (see item 2.5 on page 12) when callbacks are removed. The problem is solveable by instantiating the template

[1] Display of rentable holiday homes in 6 languages. All pages are dynamically generated even though the URLs seem to display static pages. Example of house 27093:
http://www.dancenter.dk/hus/27093 http://www.dancenter.de/haus/27093
http://www.dancenter.no/hus/27093 http://www.dancenter.se/hus/27093
http://www.dancenter.nl/huis/27093 http://www.dancenter.co.uk/house/27093

[2] Booking system for the rental system, including creditcard payment.

using reflection so the servlet is not directly dependent on the generated template java class and is therefore compilable even if the java source generated by the template system isn't.

8.1 Further work

There are several areas where the template system can be improved:

1. The current system requires the callback methods to be `public` if the generated java class is created in the '`templatedata`' sub-package. Most of those methods should only be accessible from the generated template class and java lacks C++'s `friend` concept which is the simple solution to the problem. Circumventing the access restrictions by using reflection causes lack of (javac) compile-time check of methods and the template compiler thus has to fully check the defined methods for type errors to avoid unpleasant surprises at runtime. As the overhead of using reflection is unknown, it should be measured—including the current compiler's usage of reflection for instantiating objects of inner classes.

2. The JTidy library is non-maintained and its parsing of html performs 'optimizations' which are not valid for a DOM-based template system.[3] The whatwg[4] working group has produced a modern html parser[5] which should be used as a replacement.

3. Due to the limited time available for this project, almost no Javadoc documentation is provided as the time has been spent writing this report. The Javadoc low-level documentation is still needed, especially for the annotations and main compiler classes as they are easily accessible in IDEs. Many java projects provide the low-level documentation only and have difficulty communicating the high-level concepts and easy examples for beginners. This project provides high-level documentation only, which isn't a perfect solution either because it makes it difficult to dive into the compiler's low-level internals.

4. As seen in the opcode example (section A.4 on page 59) the pcode sometimes calls the same boolean callback method several times. `@ETagIf` calls the method twice: once for the open element tag and once for the close element tag. For non-trivial callback methods, it would be a nice performance optimization to cache the returned value so the method is only called once. However, this optimization is not simple as the user expects the methods to be called for each iteration of a loop and whenever the scope has changed. This can be implemented at runtime by caching the results and purging the cache for each iteration or scope change, however, this method has quite an overhead for simple callback methods. The better—but more complex implementation—is to make the compiler detect when cached values are valid and when they have to be purged.

5. Many larger commercial European websites are multi-language sites. The site used for testing this template system provides *all* pages in 6 Germanic languages. This is implemented by using a single html template for all translations and inserting the translated text at runtime by accessing a shared translation database. The annotation-based template system requires each text needing translation to be put inside a `<span`

[3] E.g. removal of empty `<div>` elements.
[4] http://www.whatwg.org
[5] http://code.google.com/p/html5lib

`id=...>` element and a `@EText String getXXX() {...}` method to access the translations.

As many European sites are only provided in Indo-European languages which do not require a specialized layout due to e.g. right-to-left writing direction and therefore allows using the same html template for all languages, an improvement of the template system is a way of providing translated texts without the need for a manually written java method.

A solution to this problem requires design of the html macro and of a generic interface to translation systems. One possible design of the html macro is to use `id` attribute values which start with a double underscore and use the rest as the (relative path) name of the text in the translation system.

Appendix A

The P-code format

A RISC-style 32-bit integer with fixed-size encoding of command and parameter number is used to simplify the pcode encoding. The least-significant 4 bits contain the command number and the remaining 28 bits the parameter value.

A.1 Opcode enum

The Opcode enum handles encoding and decoding the format:

L58
```
enum Opcode
{
    PRINT_STRINGTABLE, // see "Opcode description" section below
    PRINT_STRING_VAR,
    PRINT_HTML_STRING_VAR,
    PRINT_ATTR_STRING_VAR,
    PRINT_INT_VAR,
    CALLBACK,
    GOTO, CONDITIONAL_GOTO,
    NEGATE,
    RETURN;

    final static int OPCODE_BITS = 4;
    final static int OPCODE_MASK = (1 << OPCODE_BITS) - 1;

    int encoded() {
        return ordinal();
    }

    int encoded(int param) {
        return ordinal() | (param << OPCODE_BITS);
    }

    static Opcode decodeOpcode(int encoded) {
        return opcodes[encoded & OPCODE_MASK];
    }
    private static Opcode[] opcodes = values();

    static int decodeParam(int encoded) {
        return encoded >>> OPCODE_BITS;
    }
}
```

A.2 Opcode execution

The execution of the pcodes is handled by `TemplateBase` which the generated java source inherits from. See L57 on page 48 for the `void executeCallback(int idx)` method:

L59
```
abstract class TemplateBase
{
    String stringValue;
    int intValue;
    boolean boolValue;
    ...

    void emit(HtmlPrintWriter pw) throws Exception
    {
        final int[] opcodes = ...;
        final String[] strings = ...;
        int pc = 0;
        while (true) {
            int encoded = opcodes[pc];
            Opcode opcode = Opcode.decodeOpcode(encoded);
            int param = Opcode.decodeParam(encoded);
            pc++;
            switch (opcode) {
                case PRINT_STRINGTABLE: pw.print(strings[param]); continue;
                case PRINT_STRING_VAR: pw.print(stringValue); continue;
                case PRINT_HTML_STRING_VAR: pw.htmlEncode(stringValue); continue;
                case PRINT_ATTR_STRING_VAR: pw.attrEncode(stringValue); continue;
                case PRINT_INT_VAR: pw.print(intValue); continue;
                case CALLBACK: executeCallback(param); continue;
                case GOTO: pc = param; continue;
                case CONDITIONAL_GOTO: if (boolValue) pc = param; continue;
                case NEGATE: boolValue ^= true; continue;
                case RETURN: return;
                default:
                    throw new TemplateException(
                        "Invalid opcode = " + encoded + " at addr = " + pc);
            }
        }
    }

    abstract void executeCallback(int idx) throws Exception;
}
```

A.3 Opcode description

print_stringtable prints a string from the stringtable indexed by the parameter. The string is printed as-is without encoding as it contains html.

print_string_var prints the stringValue field without encoding. This is used for e.g. `@EHtml String xxx()` methods

print_html_string_var prints the stringValue field using normal html encoding (e.g. '<' is converted to '`<`', newlines to '`<br>`', 16-bit characters is encoded as '`&#1234;`'). The `@EHtml String getUsername()` method results in a callback-opcode followed by print_html_string_var.

print_attr_string_var is similar to **print_html_string_var**, but does not encode newlines as '<br>'. Used for `@AName String namePrinterNickname()`.

print_int_var prints 'intValue'. Used for integer attributes, e.g. `@AColSpan`.

callback calls the `void executeCallback(int idx)` method, thereby executing the dynamically generated code. Used whenever the class has to access the programmers java class.

goto changes the program-counter to the address specified in the parameter.

conditional_goto changes the program-counter if `boolValue` is true. The variable is set by executing a callback for a method/field returning a boolean, e.g. for `@EIf` and `@EWhile`.

negate inverts the value of `boolValue`, which is used for inverted conditional goto. An alternative approach is to have an **inverted_conditional_goto** instead of **negate**.

return stops execution of the program. It is only emitted once and only as the last pcode. It is redundant as the loop could return when the end of the pcode array was reached, but using an explicit opcode as an sentinel ensures an exception if the array is only partially generated or contains errors such as goto addresses beyond the end of the array.

A.4 Opcode example

The generated pcode for the PrinterListNick example is:

address	opcode(param)	stringtable text / callback
0	print_stringtable(0)	`<html>\n<head>\n<title></title>\n</head>...`
1	callback(8)	`stringValue = scopeRoot.getPersonFullName();`
2	print_html_string_var	
3	print_stringtable(1)	`</span> (<span id=Username>`
4	callback(9)	`stringValue = scopeRoot.getUsername();`
5	print_html_string_var	
6	print_stringtable(2)	`</span>).</div>\n`
7	callback(10)	`boolValue = scopeRoot.testHasPrinters();`
8	conditional_goto(10)	
9	print_stringtable(3)	`\n<div id=NoPrinters>You don't have ac...`
10	callback(10)	`boolValue = scopeRoot.testHasPrinters();`
11	negate	
12	conditional_goto(34)	
13	print_stringtable(4)	`\n<form action=.  class=HasPrinters>\n<d...`
14	callback(4)	*get iterator from* `iterPrinterRow()`
15	goto(31)	
16	callback(5)	*create new instance of EmitPrinterRow*
17	print_stringtable(5)	`\n<tr class=PrinterRow>\n<td><input cla...`
18	callback(1)	`stringValue = scopePrinterRow.getPrinterName();`
19	print_html_string_var	

20	print_stringtable(6)	`</td>\n<td><input class=PrinterNicknam...`
21	callback(3)	`stringValue = scopePrinterRow.namePrinterNickname();`
22	print_attr_string_var	
23	print_stringtable(7)	`' type=text value='`
24	callback(2)	`stringValue = scopePrinterRow.getPrinterNickname();`
25	print_attr_string_var	
26	print_stringtable(8)	`'></td>\n<td class=AccountStatus>`
27	callback(0)	`stringValue = scopePrinterRow.getAccountStatus();`
28	print_html_string_var	
29	print_stringtable(9)	`</td>\n</tr>\n`
30	callback(7)	`scopePrinterRow = null;`
31	callback(6)	*get next object from iterator*
32	conditional_goto(16)	
33	print_stringtable(10)	`</table>\n<input type=submit value=Upd...`
34	print_stringtable(11)	`</body>\n</html>\n`
35	return	

Size of pcode is 36 * sizeof(int) = 144 bytes. Total size of the 12 strings in the stringtable is 681 chars.

A.5 Saving pcode

The pcode and stringtable is saved and loaded using the `TemplateData` class. It is saved in a separate binary file instead of being embedded in the generated java source to support dynamic recompiling of the template and because the java bytecode format is very space-inefficient for initializing arrays.

Appendix B

System test example

This example tests the `@ETagIf` annotation. The html template is:

```
L60  <html>
     <body>
          <div><a id=Keeper href="http://www.ruc.dk">Roskilde University</a></div>
          <div>Goto: <a id=DropMe href="http://example.com">No place in
                    particular</a>.</div>
     </body>
     </html>
```

The java code is:

```
L61  package dk.vitality.util.jat.junit.tests.iConditionalTag;

     import dk.vitality.util.jat.junit.lib.TestBase;
     import dk.vitality.util.jat.runtime.annotation.flowControl.ETagIf;
     import dk.vitality.util.jat.runtime.annotation.global.TCompile;

     @TCompile
     public class TestConditionalTag extends TestBase {

         protected void emitTemplate() throws Exception {
             new TestConditionalTagTp(this).emit(pw);
         }

         @ETagIf public boolean testKeeper() { return true; }

         @ETagIf public boolean wantsKeeper() { return true; }

         @ETagIf public boolean testDropMe() { return false; }

         @ETagIf public boolean failedDropMe() { return true; }
     }
```

The first two methods match the `id=Keeper` element and the last two methods match the `id=DropMe` element. Multiple matches are used to test that the compiler correctly handles nested conditionals.

The `TestBase` class provides JUnit encapsulation such as collecting the output written to `pw` (a PrintWriter instance) and compares it to an html file which contains the expected output.

The tag of the first element is included in the output as both methods returns true while the tag of the second element is omitted as the third method (`testDropMe`) returns false. The expected output is:[1]

[1] JTidy adds an empty title tag if no title was specified in the output. The doctype is omitted for brevity.

L62 <**html**>
<**head**>
<**title**></**title**>
</**head**>
<**body**>
<**div**><**a** href='http://www.ruc.dk' id=Keeper>Roskilde University</**a**></**div**>
<**div**>Goto: No place in particular.</**div**>
</**body**>
</**html**>

Bibliography

[AGH06] Ken Arnold, James Gosling, and David Holmes.
The Java^{TM} Programming Language, Fourth Edition. Addison-Wesley, 2006.

[AHLL06] James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. Tool support for randomized unit testing. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 36–45, New York, NY, USA, 2006. ACM Press.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BDKZ93] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, 1993.

[BDP06] Manfred Broy, Florian Deissenboeck, and Markus Pizka. Demystifying maintainability. In *WoSQ '06: Proceedings of the 2006 international workshop on Software quality*, pages 21–26, New York, NY, USA, 2006. ACM Press.

[BMMIM98] William H. Brown, Raphael C. Malveau, Hays W. 'Skip' McCormick III, and Thomas J. Mowbray. *AntiPatterns. Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc., 1998.

[BP97] Ira D. Baxter and Christopher W. Pidgeon. Software change through design maintenance. In *Proceedings of the IEEE Conference on Program Comprehension*, pages 250–259. IEEE Press, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[LC00] Young Lee and Kai H. Chang. Reusability and maintainability metrics for object-oriented software. In *ACM-SE 38: Proceedings of the 38th annual on Southeast regional conference*, pages 88–94, New York, NY, USA, 2000. ACM Press.

[LST78] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[MF04] Brett McLaughlin and David Flanagan.
Java 5.0 Tiger. A Developer's Notebook^{TM}. O'Reilly, 2004.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., 1979.

[Wel] Martijn van Welie. Web design patterns. `http://www.welie.com/patterns/`.