

Jostraca: a Template Engine for Generative Programming

Position Paper for the ECOOP'2002 Workshop on Generative Programming

Richard J. Rodger
InterComponentWare AG, Otto-Hahn-Straße 3, 69190 Walldorf, Germany
Telephone: +49 6227 385124, Mail: richard.rodger@intercomponentware.com

Abstract

Generative Programming (GP) in practice often faces the problem that specialists are required to produce and maintain the GP infrastructure. The concepts of templates and domain models are shunned by many programmers who are not familiar with the GP approach. However, implementations of these concepts are readily accepted by working programmers who are quite at home with Java Server Pages (JSP) or XML. The problem lies in the ergonomics of GP, and it is this problem that the Jostraca template engine attempts to solve.

1. Basic Concepts

Following [We01], GP has two primary aspects: representation of the problem domain, and implementation of the solution domain. Jostraca[Js] confines itself strictly to the implementation aspect. In this sense, Jostraca is orthogonal to any given Domain Specific Language (DSL). Furthermore, the template engine attempts to maintain independence from any given domain model. This independence from the representational aspect of GP is a central feature of the system.

Focusing on the implementation aspect helps surmount a major obstacle on the road to wider acceptance of GP: the problem of credibility[Be]. Programmers have a natural

and justified abhorrence of excessive complexity and unfamiliar syntax. Many template systems immediately present this outward appearance. However, in order to use GP effectively, programmers must learn to work with templates, since template manipulation remains the primary mode of producing both original and custom implementations. For this reason, Jostraca explicitly chooses to mimic the familiar syntax of JSP. In this manner we flatten the learning curve. By basing our syntax on proven technology we solve an entire class of adoption problems.

The following example (figure 1) demonstrates a Jostraca template which creates a simple value object in Java.

```
<% String[] fields = new String[] { "FirstName", "LastName" }; int i; %>
public class Person {
  <% for( i = 0; i < fields.length; i++ ) { %>
    private String m<%=fields[i]%> = "";

    public String get<%=fields[i]%>() {
      return m<%=fields[i]%>;
    }

    public void set<%=fields[i]%>( String p<%=fields[i]%> ) {
      m<%=fields[i]%> = p<%=fields[i]%>;
    }
  }
  <% } %>
}
```

Figure 1. Jostraca Template. Note: some details omitted for clarity; we assume familiarity with JSP syntax.

This outputs the generated code in figure 2.

```
public class Person {
    private String mFirstName = "";
    public String getFirstName() {
        return mFirstName;
    }
    public void setFirstName( String pFirstName ) {
        mFirstName = pFirstName;
    }

    private String mLastName = "";
    public String getLastName() {
        return mLastName;
    }
    public void setLastName( String pLastName ) {
        mLastName = pLastName;
    }
}
```

Figure 2. Generated Code. Note: some modifications for clarity.

For the sake of precision, we define the following terms: *template script* consists of the programming language statements within the script markers (<% and %>) and *template text* consists of the plain text source code outside the script markers. The strength of this syntax lies in its ability to leverage existing knowledge of the template scripting language (template script) for the purposes of manipulating the source code of the target language (template text) in a manner which, for the most part, preserves the appearance of the target source code in a readable format. That the template scripting language is a general purpose programming language is a key strength of this approach. The programmer is not required to learn a new templating language. This advantage very much offsets the disadvantage that the JSP-style syntax is aesthetically displeasing. In addition, the JSP-style syntax requires no knowledge of Abstract Syntax Trees (AST) or other higher-level language descriptions.

The actual code generation is a 2-phase process. First, in the template compilation phase, the template is compiled into valid source code of the template scripting language, into a program known as the *CodeWriter*. Second, in the code generation phase, the CodeWriter is then itself compiled and executed to output the generated source code of the target language. This architecture is somewhat similar to the standard JSP implementation [Tomcat], but with CodeWriters replacing servlets. It also allows clean and complete separation of template

compilation from code generation, so that this simple design allows Jostraca to use any external programming language as the template scripting language.

Naturally the CodeWriter is itself produced from a template. At present a simplistic substitution syntax is used to merge the compiled template script and text into the CodeWriter template. There is no reason why this cannot be done using the standard JSP syntax in future.

What is important to note is that the programmer is also free to define the CodeWriter. That is, the very environment of code generation can itself be modified and customized. For instance, the basic CodeWriters supplied with Jostraca are merely glorified *HelloWorld* programs. But they could just as easily be integrated as components into a larger application.

The Jostraca implementation differs from existing JSP implementations in that it is focused on solving the practical problems of template-based GP. Usability for software developers must be an important goal for any GP system. Usability in this sense means small, fast, single purpose tools that can be easily integrated into automated software construction environments. It also means tools that can be abandoned (by adopting the generated code as-is) when necessary. The template mechanisms provided by popular application servers have very different usability requirements and goals. On the

implementation level, the services they provide (for example, multi-user session handling) are superfluous to GP. In addition, while the CodeWriter concept is clearly defined by Jostraca, the code generation environment of application server is implementation dependent.

2. Domain Model and Language Independence

There is a trade-off to this approach. Accessing the domain model is no longer natural, as in other systems, such as [CI], or [Ra]. However, as previously stated, we explicitly solve this problem by defining it out of scope; that is, Jostraca remains independent from the representational aspect of GP. Jostraca is thus *not* a complete GP system. However, this trade-off gives us enormous freedom. We gain domain model independence, template scripting language independence and we can make the code generators into independent components.

Accessing an external domain model is, of course, not so easily brushed aside. Some suggestions are offered below, but ultimately, in order to preserve domain model independence, this problem is left to the implementors of DSLs and other domain model representations. For small scale problems a simple domain model expressed as a formatted text file is often sufficient. This follows the appeal to simplicity in [PP]. The power of a simply parsed, plain text domain model should never be underestimated. For larger scale problems, XML provides a convenient way to represent a domain model. XML structures can be readily accessed from many languages using third party components. In particular, access using [XPath] has proven to be a very convenient approach in practice. For even larger problems, or domain specific representations, we expect that any given DSL provides reasonable programmatic access to its AST, as a minimum, and probably, if mature, provides a user friendly API. It is here that the language independence of Jostraca is advantageous, since the templates can then be written in the same language that is used to parse the DSL.

The primary goal of the Jostraca system is to exist as a component in the GP infrastructure. This allows programmers the freedom to use other tools better suited to domain analysis for the solving the domain representation problem. For instance, UML diagrams can be exported to [XMI], which can then be used by Jostraca templates. Even further up the chain, requirements modelling tools such as Catalyze[Ca] can be used with Jostraca to produce complementary artefacts such as design documentation. Note that Jostraca exists alongside these tools and can be used without vendor support.

In practice, the most immediate value of Jostraca is to provide a consistent implementation of the data model, from generating SQL table definitions to [EJB] objects. For the most part this can be done in isolation from the rest of the system, given sufficient clarity in the definition of component interfaces. Further scope exists for the integration of Jostraca templates with such tools as [Fr], or [Fj]. Again, such integration is not tool dependent, so long as the domain representation is available.

3. Future Directions

Practical experience using Jostraca templates for code generation has provided some ideas for extending the concept. The main weakness of a system such as Jostraca is that it is text based. This means that template manipulation is awkward. Unfortunately, template manipulation (which is, ultimately, customization of the domain implementation), has shown itself to be necessary in almost every application of the system. It is important to distinguish between customization at the domain model level, which is easily solved because the degrees of freedom are well-defined in advance, and customization at the implementation level, where standard implementation choices may not suit the problem at hand.

Use of an object oriented language for the target implementation certainly helps to solve a number of customization problems. Object inheritance and mixins provide reasonable

approaches. Programmatic inclusion of code snippets into the templates at template compilation or code generation time also provides some scope for customization. However, unless carefully controlled, these ad hoc approaches quickly become unmanageable. This is a problem of scale, and may be solved by drawing an analogy with structured programming. By breaking templates into meaningful, reusable and isolated structures, we can solve larger problems. What would these *structured templates* look like? [Ma] and [Vö] provide some suggestions for analysing templates and other code artefacts in terms of ASTs and applying semantic constraints to them. From this work and also from Aspect Oriented Programming (AOP), we can take the notion of *join points*, and try to apply the idea to templates. While Jostraca templates are text based, there also exists a very convenient convention for converting them into XML [JX], provided by Sun Microsystems for JSP. With some minor variations, Jostraca templates can be represented using XML. This allows us to use XML nodes as join points for composing templates from smaller pieces, in a manner analogous to AST manipulations. It is however by no means clear whether an XML based approach can provide sufficient or suitable structure. As with structured programming, it is necessary to resolve scope issues for structured templates, such as protecting scripting language local variables.

Imposing a syntactic structure on templates would allow us to provide standard problem domain implementations, generated from a standard set of composable templates, but which are customizable in a well-defined manner. It would be possible to define mappings from feature sets [CzEi] to template structures. But note again the distinction between domain implementation variabilities (which such feature sets would describe) and domain representation variabilities (which are defined by the domain model semantics).

A second future direction is to address the issue of template ergonomics. Although, as stated, JSP syntax provides an initial stepping stone, the syntax itself remains quite ugly for defining source code templates. This effect

occurs because the variation points in source code are considerably more numerous than in HTML, for which the JSP syntax was originally designed. At present Jostraca attempts to solve this problem in a naïve fashion, by allowing the programmer to define simple text replacements on the template. For example, the template in figure 1 becomes the template in figure 3.

These replacements make the structure of the code much clearer and easier to work with in practice. The main problem with this approach is error detection: misspellings of the replacement terms are not identified and lead to mysterious errors. However, error detection in templates is a common problem in many template systems (C++ provides a good example), so this failing is hardly unique. The reference JSP implementations [Tomcat] provide some support for error detection using back references to line numbers in the original files (and Jostraca will support this in future), but the essential problem remains a weakness of text-based templates in general. However an advantageous consequence of this replacement technique is that the template in figure 3 can actually be compiled by Java. This does present a useful method for identifying syntax errors in the template, if not in the template script.

The code snippet in figure 3 can be interpreted as an example of the intent of the programmer. In effect, the template engine has been given an example of a generic structure at a lower level of abstraction than is commonly required for generic (or dynamic) programming structures. Using GP allows the programmer to add and modify fields of the value object very easily, without resorting to more complex dynamic data structures. It is commonly observed that examples are easier to grasp than abstract definitions and this psychological effect can be used to provide a new way of looking at templates: as *programming by example*.

```

/*
<% String[] fields = new String[] { "FirstName", "LastName" }; int i; %>

<% @replace "//-for-each-field-//" "$<o>for(i=0;i<fields.length;i++){<c>" %>
<% @replace "Field" "$<o>=fields[i]<c>" %>
<% @replace "//-end-//" "$<o> } <c>" %>
*/

public class Person {
//-for-each-field-//
    private String mField = "";

    public String getField() {
        return mField;
    }

    public void setField( String pField ) {
        mField = pField;
    }
//-end-//
}

```

Figure 3. User-friendly Template. Note: `<o>` and `<c>` are escapes for `<%` and `%>` respectively

The concept of programming by example is quite powerful. It allows us to design systems at one level of abstraction lower than usual, safe in the knowledge that the direct implementation can be used as an example (that is, as a template), to produce the full solution. We can explicitly fix constraints (say, the definition of fields in our database), which are in reality subject to change. To what extent this metaphor can be maintained remains to be seen. Certainly, it is impossible to remain unaware of the context which the template expects, or the subtle problems associated with data types (for example, in Java, the difference between `int` and `Integer`). However, by providing templates as close to the target language syntax as possible, we enable many more non-specialists to customize or otherwise work with templates.

4. Summary of Position

Jostraca does not attempt to solve the problems of the representation domain. It does however attempt to solve the problems of the implementation domain. In order to serve many domain models and languages, Jostraca maintains programming language independence. An attempt is also made to provide Jostraca as a component of a much larger GP infrastructure composed of modelling tools, DSLs, and XML domain models. Jostraca thus remains a plain text system, but with the future aim of providing a more abstract representation of templates themselves, so that templates can become structural units and can themselves be composed.

References

- [We01] Wegener, H., **Generative Programming and Incompleteness**, OOPSLA'2001 Workshop on Generative Programming, October 2001.
- [Js] Jostraca Code Generator, <http://www.jostraca.org>
- [Be] Bettin, J., **Practical Use of Generative Techniques in Software Development Projects: an Approach that Survives in Harsh Environments**, OOPSLA'2001 Workshop on Generative Programming, October 2001.
- [Cl] Cleaveland J., **Program Generators with XML and Java**, Prentice-Hall, 2001.
- [Ra] Rausch, A., **A Proposal for a Code Generator based on XML and Code Templates**, International Conference on Software Engineering, Toronto, May 2001.
- [Tomcat] The official implementation of JavaServer Pages, <http://jakarta.apache.org/tomcat>
- [PP] Hunt A., Thomas D., **The Pragmatic Programmer**, Addison-Wesley, 2000.
- [XPath] XML Path Language, <http://www.w3.org/TR/xpath>
- [XMI] XML Metadata Interchange,
<http://www.omg.org/technology/documents/formal/xmi.htm>
- [Ca] Catalyze, SteelTrace Ltd., <http://www.steeltrace.com>
- [EJB] Enterprise JavaBeans Specification, v1.1, Sun Microsystems Inc.,
<http://java.sun.com/products/ejb/docs.html>
- [Fr] Hakala M., Hautamäki J., et al, **Architecture-Oriented Programming Using FRED**, International Conference on Software Engineering, Toronto, May 2001.
- [Fj] van Emde Boas, G., **Architecture Based Code Generation from UML Models**, OOPSLA'2001 Workshop on Generative Programming, October 2001.
- [Vö] Völter M., **Jenerator – Generative Programming for Java**, OOPSLA'2001 Workshop on Generative Programming, October 2001.
- [Ma] Majkut M., **Syntactic Unit Trees for the Implementation of Software Product Lines**, ECOOP'2001 Workshop on Generative Programming, June 2001.
- [JX] JavaServer Pages Specification, v1.2, Sun Microsystems Inc.,
<http://java.sun.com/products/jsp/download.html>
- [CzEi] Czarnecki K., Eisenecker U., **Generative Programming, Methods, Tools, and Applications**, Addison-Wesley, 2000.