View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KITopen

Metaprogramming Applied to Web Component Deployment

Welf Löwe Markus L. Noga
University of Karlsruhe
Program Structures Group
Adenauerring 20a, D-76131 Karlsruhe, Germany
{loewe|noga}@ipd.info.uni-karlsruhe.de

October 31, 2001

Abstract

Metaprogramming is a generic approach described in many articles. Surprisingly, examples of successful applications are scarce. This paper gives such an example. With a metaprogram of less than 2500 lines, we deploy components on the web by adding specific XML-based communication facilities. This underlines the expressiveness of the metaprogramming approach.

1 Introduction

Metaprogramming interprets a source program as data the can be analyzed and transformed. Depending on the time of its application, we distinguish between static and dynamic metaprogramming. The former performs analysis and transformations at compile time, the latter at runtime.

The idea of metaprogramming is not new: it is a direct consequence of the von Neumann computer architecture. As early as in the 1950s, Lisp 1.5 [7] treated program and data terms uniformly. Both could be reflected and transformed at runtime. This instance of dynamic metaprogramming was implemented by simply interpreting all code.

Recently, interest in metaprogramming has been revived. One reason is the success of the Java programming language and its reflection interface that allows dynamic code analysis. Another is the need to re-engineer large legacy code bases. Last but not least, software engineering increasingly aims to reuse predefined components. As these components are defined for as broad a reuse as possible, deployment should make them fit into any concrete environment. This deployment is automatized with metaprogramming.

There are quite a few metaprogramming tools around. Most of them specialize in aspect weaving or refactoring. As such, their users cannot design new metaprograms. Exceptions to this rule are *Puma* [1], *Transmogrify* [2], the *Design & Maintenance System (DMS)* [3] and *Recoder* [6]. Apart from *Recoder*, the mentioned systems are restricted in the generality of their parsing and semantic analysis phases. *Recoder* provides a full compiler front end for Java and access to all syntactic and semantic information analyzed by it.

While the necessity of metaprogramming is undisputed and supporting tools have already been developed, the literature does not report many real applications. Publications address specific questions of metaprogramming at hand, or discuss the application side exclusively, leaving metaprograms to the intuition of reader. The present paper aims at bridging this gap.

As an sample application of metaprogramming, we discuss the deployment of components on the web by adding XML-based communication facilities. Although we deploy Java components only, and our metaprograms use *Recoder*, we refrain from exploiting Java specifics such as the language reflection interface. Therefore, our approach generalizes to other languages and tools.

The paper is organized as follows: We first discuss the domain of the case study in section 2. In section 3, we present a suitable software architecture for the domain. We then generate conforming implementations with metaprogramming in section 4. The final section, 5, summarizes our results and outlines directions for future work.

2 Case Study Domain

Web services are a hot topic. In essence, a web service is a class that supports remote method invocation. Invocation data are encoded in XML [13] and usually transmitted via HTTP [4]. The set of admissible invocation messages can be specified with a DTD or XML Schema [14, 15]. This foundation is attractive because it is independent of platform and language, based on open standards, intelligible to humans as well as scripting languages and thus easily amenable to adaptations. Secondary considerations, such as the capability to tunnel corporate firewalls, are still hotly debated and variously seen as a major benefit or a major security hazard.

Higher-level web service standards such as SOAP and WSDL [11, 12] define method invocation formats and interface specifications in more detail. Unfortunately, both SOAP and WSDL notations are quite cumbersome and require extensive print space. For the purposes of this article, we therefore limit ourselves to plain XML services whose interfaces are described by DTDs. We also disregard the question of addressing and use direct socket connections. None of these limitations are inherent to the metaprogramming approach. In production code, they may easily be eliminated.

How are web services to be implemented? Java [5] is today's educational language of choice. The metaprogramming system developed by our research group, *Recoder* [6], operates on Java programs, and is itself a Java program. Thus, we focus on deploying Java-based web services in this case study. To do so, we need a model of what exactly to deploy.

We distinguish component classes (or short components) from auxiliary classes: methods of component objects may be invoked remotely, while method invocations on auxiliary objects are always local, i.e. caller and callee run in the same address space. When invoking component methods, component objects are passed by reference, whereas auxiliary objects are passed by value.

Remote invocation must preserve polymorphism to guarantee location transparency. Moreover, both parameters passed to and return values of component methods may in general be complex object graphs, possibly containing cycles.

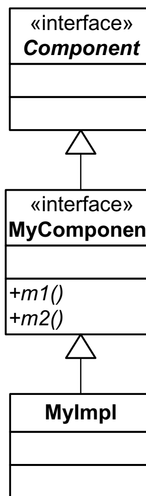

Figure 1: A component specification

To eliminate the problem of member variable access, which cannot be redirected in Java, we follow Java component tradition [10, 9] in specifying components in two parts: a component interface that specifies the permitted operations, and a component class that implements them. Interfaces must inherit a component marker interface to designate them as component interfaces (see Fig. 1).

Instead of using the language feature of interface inheritance, we could also specify component boundaries with an arbitrary metaprogram. This approach becomes appealing when componentifying legacy systems. In this presentation, we use the component marker interface for simplicity.

Now we can define the application problem precisely: Given a component specified as above, deploy it as a web service. This can be decomposed in two subtasks. First, add XML-based communication facilities that allow location-transparent access. Second, generate XML web service descriptions, i.e., DTDs. The next section will discuss a software architecture to handle XML-based communication.

3 Architecture

In the mould of classical component architectures like CORBA [8], we use the stub/skeleton pattern for remote component access. A stub implements the component interface. It runs in the client's address space and passes all method invocations over the wire, receiving the return value. A skeleton runs in the server address space. It decodes incoming messages and invokes the appropriate methods of the component class, passing return values over the wire. Specific stubs and skeletons are generated for each component. Fig. 2 shows the classes involved.

As passing return values is symmetric to passing arguments, we focus on passing arguments in this presentation. In detail, a stub is a proxy for the component class. It converts method calls to an XML element that encapsulates the method in question, component object references and auxiliary object values. Component object references can easily be encapsulated by a single element, but auxiliary object values may be very complex. However, given serializers that map auxiliary classes to elements, a method is simply mapped to an element containing a fixed sequence of argument elements.

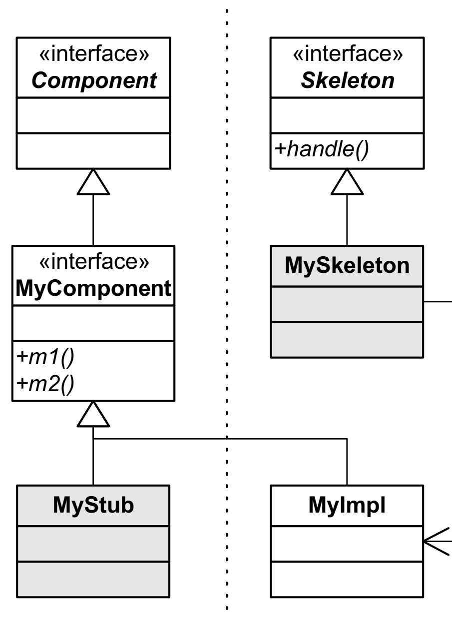

Figure 2: A deployed component. Classes to the left run on the client, those to the right on the server. Generated classes are shown in gray.

How do these serializers for auxiliary objects work, then? They map auxiliary objects to literal elements, which contain the sequence of all member variables. Here, primitive types are mapped directly to XML elements. Component objects are mapped to an element containing a URI reference. Auxiliary objects are recursively mapped to literals.

This results in a depth-first traversal of the accessible object graph, with component objects treated as leaves. For this traversal order, all cycles in the object graph manifest themselves as backward edges. They can be detected by maintaining a set of objects already serialized.

We employ sequential serialization order for reference purposes, facilitating single-pass deserialization. By universally allowing a backward reference element to stand for literal elements, cyclic object graphs can be accurately mapped to XML. In contrast, Microsoft .NET serialization does not know this concept. It can detect cycles, but is incapable of handling them.

Now, let us consider the software architecture of the serializers. A generated `XMLSerializer` class contains one static serializer method per auxiliary class, array and interface. These serializer methods target an invariant `XMLSerializerStream` object. Its class encapsulates the output stream and the set of already serialized objects. It contains convenience methods to serialize primitive types, backward references and nulls (represented as a backward reference to the zeroth object).

For access to private member variables, two serialization helper methods must be woven into the auxiliary classes. The static serializers can thus employ polymorphic method calls to resolve dynamic types.

Example 1 *Assume the following auxiliary class definitions:*

```
abstract class A                { int i; A a; }
interface B                     { }
class C extends A               { float f; B b; }
class X extends C implements B { X x; }
class Y extends X               { boolean[] b; }
class Z implements B            { X x; }
```

The following method is generated for `Y` *in* `XMLSerializer`*:*

```
public static void serializeClassY(XMLSerializerStream s, Y o) {
  if(s.serializeReference(o))
    return;
  o.serializeXML(s);    // o may be of type Y or any subtype thereof
}
```

The static serializer polymorphically invokes the data type serializer, which in turn monomorphically invokes the data layout serializer. Prior to serializing its own fields with the appropriate static serializers, the data layout serializer monomorphically invokes the superclass data layout serializer to serialize inherited member variables. This multi-stage dispatch has significant runtime flexibility. In contrast, Microsoft .NET serialization is restricted to statically specified data types. When adding classes, all methods using their superclasses need to be updated.

Example 2 *(Continues Example 1) The following methods are added to class* `Y`*:*

```
public void serializeXML(XMLSerializerStream s) {
    s.openingTag("<classY>");
    serializeBodyClassY(s);
    s.closingTag("</classY>");
}

protected final void serializeBodyClassY(XMLSerializerStream s) {
    serializeBodyClassX(s);                             // superclass data layout
    XMLSerializer.serializeArrayOfBoolean(s, this.b); // boolean[] b;
}
```

To illustrate the operation of the serializers, consider the object graph built from classes defined in Example 1 depicted in Fig. 3. Example 3 below shows the corresponding serialization of the instance of class `Z`.

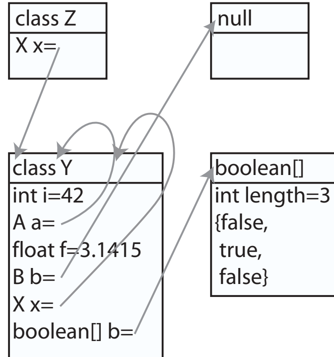

Figure 3: A sample object graph.

Example 3 *(Continues Examples 1,2) Annotated serialization of the* `Z` *instance in Fig. 3.*

```
<classZ>                          <!-- id 1                        -->
  <classY>                        <!-- id 2       content of x     -->
    <int>42</int>                 <!-- value      content of i     -->
    <ref>2</ref>                  <!-- reference  content of a     -->
    <float>3.1415</float>         <!-- value      content of f     -->
    <ref>0</ref>                  <!-- null       content of b     -->
    <ref>2</ref>                  <!-- reference  content of x     -->
    <arrayOfboolean length="3">   <!-- id 3       content of b     -->
      <boolean>false</boolean>    <!-- value      content of b[0] -->
      <boolean>true</boolean>     <!-- value      content of b[1] -->
      <boolean>false</boolean>    <!-- value      content of b[2] -->
    </arrayOfboolean>
  </classY>
</classZ>
```

Let us return to Fig. 2. So far, we omitted the details of skeleton operation. A skeleton is the inverse of a stub: it receives XML representations of method invocations from the transport channel, identifies the method in question and reconstructs the serialized argument sequence. Via a simple dispatcher, it invokes the appropriate component method. Just like a stub passes arguments, it then passes the return value over the wire.

Reconstructing the argument sequence requires deserializers for primitive types, component objects and auxiliary objects. As for serializers, the first two are rather simple constructs. For auxiliary objects, there are direct correspondences between `XMLSerializer` and `XMLDeserializer` as well as between `XMLSerializerStream` and `XMLDeserializerStream`. The main difference is that static deserializers cannot use polymorphic calls. They must explicitly dispatch over the known subtype names to call deserializing constructors woven into the auxiliary classes.

4 Implementation

To add the architectural elements discussed in the previous section to the components and auxiliary classes of section 2, we employ metaprogramming facilities provided by the *Recoder* system. Following

convention, the metaprogram can be decomposed into separate analysis and transformation phases. The former derives the required information from the existing sources, the latter performs the actual modifications required. In the subsections below, we discuss these phases individually.

4.1 Analysis

The analysis phase traverses the input program and derives the data necessary for the transformation phase. It mainly computes the set of components and the set of auxiliary types required for method calls to the components. It then determines member variable names and types for auxiliary types, as well as basic subtype relationships and its transitive closure. Component stub and skeleton generation additionally require information on available methods.

Although the actual implementation differs slightly, the analysis can be formulated as a fixed-point iteration. Its initialization looks like this:

- Identify all interfaces that inherit `Component`.
- For each such interface, identify all available methods.
- For each such method, add all types in the signature to the set of used types.
- Set the auxiliary types to the empty set.

An individual iteration step consists of these operations:

- For all used arrays, add their base types to the used types.
- Add all used types that do not inherit `Component` to the auxiliary types.
- For all auxiliary types, determine their superclasses and super-interfaces.
- Add them to the used types.
- For all auxiliary types, determine all subtypes.
- For all auxiliary types, identify their member variables.
- For all such member variables, add their types to the used types.

Obviously, a semantically rich model of the input program is required to perform these computations. While there is no need to visit method bodies, full semantic analysis of type relations, member signatures and member variables is a prerequisite to this fixed-point iteration.

4.2 Transformation

The transformation phase acts on the program model to actually incorporate the required modifications. In this case, there are no changes or deletions — only additions are performed.

Although it is plainly necessary to assemble various methods, e.g., static serializers and serialization helpers, this does not affect existing method bodies. In fact, the stubs, skeletons, static serializer and deserializer classes may be assembled off-line and added to the model as a whole. Similarly, all serialization and deserialization helpers can be assembled individually and added to the model on the method level.

4.3 Results

We implemented the analysis and transformation phases with *Recoder*. As the operations mentioned correspond very closely to *Recoder* methods, we do not present actual metaprogram code here. Consult the *Recoder* manual [6] for technical details.

Our working prototype supports the full Java type system, including all primitive types, arrays and arbitrary user-defined classes, abstract classes and interfaces. Serializers, deserializers, stubs and skeletons are correctly generated. Remote invocations are operable.

Of cause, a working prototype is not a time-tested product. We do not handle exceptions. `java.lang.Object` remains problematic, as it has both component and auxiliary subclasses. We cannot deal with binary classes — while simple in theory, a mapping to user-defined external serializers has not been implemented yet. In a similar vein, our network handling is very basic, lacking scalable component addressing schemes or security handling.

However, these domain issues are only mildly relevant to the metaprogramming example. What counts is this: although our prior exposure to *Recoder* was strictly theoretical, we were able to build a working prototype within four person-days. Metaprogram and runtime framework for the generated code together amount to 2500 LOC.

5 Conclusions

This case study leads to conclusions on three separate levels: domain, process and system architecture.

Let us first consider the problem domain. We have successfully realized web services. Our metaprogram invasively integrates existing components into a web invocation architecture. While we chose to implement a simple XML encoding for readability, full conformance to SOAP and WSDL is easily achievable with metaprogramming technology.

Apparently, metaprogramming works well for this domain. As we encountered in the constituent parts of our solution, the related domain of serialization, and therefore the larger issues of data bindings and persistence, are also well-suited to metaprogramming approaches.

The software development process for this example was a surprise to us. Although we had no prior experience using *Recoder*, the metaprogram and the associated runtime framework was implemented within four person-days. Together, they amount to a slim 2500 LOC. This is an impressive testimony to the expressive power of metaprogramming.

In the architectural sphere, we note that metaprogramming systems are a good layer of abstraction. Encapsulating some 82000 LOC behind a clean interface, *Recoder* provides a complete structure tree with full semantic analysis and support for general transformations. This is a far harder problem than most conceivable individual transformations. Metaprogramming systems make sophisticated compiler technology easily accessible. As such, they are a useful building block for larger systems.

Reading about large systems that manipulate software, one cannot fail to notice that the terminologies of metaprogramming, invasive adaptation and the field of aspect-oriented programming are overlapping ones. Based on our experience, we recommend a layered architecture for these systems (see figure 4).

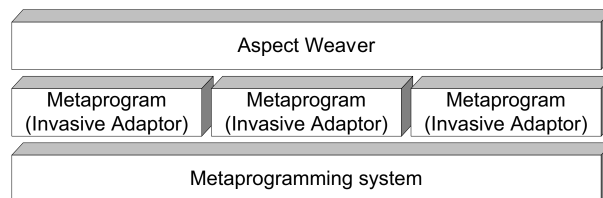

Figure 4: A high-level architecture for metaprogramming systems

A metaprogramming system like *Recoder* is the foundation. Individual invasive adaptations, like our study in XML web services, are metaprograms built upon that layer. Finally, an aspect weaver on top orchestrates the application of the various invasive adaptations.

What is on the road ahead? We plan to conduct further case studies. This increases the library of metaprograms for invasive adaptation in the architecture sketched above. In due course, this strategy should yield valuable insights about metaprogram design. It will pave the road to interoperable metaprograms that may one day be orchestrated by an aspect weaver, which remains a subject for future research.

References

[1] *Homepage of PUMA - The PURE Manipulator.* `http://ivs.cs.uni-magdeburg.de/~puma/home-eng.html`, 2001.

[2] *Transmogrify.* `http://transmogrify.sourceforge.net`, 2001.

[3] Ira D. Baxter. Transformation systems: Domain-oriented component and implementation knowledge reuse. In *Proc. 9th Workshops on Institutionalizing Software Reuse*, 1999.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* IETF RFC 2616, `http://www.ietf.org/rfc/rfc2616.txt`.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification.* Addison-Wesley, 2nd edition, 1996.

[6] Andreas Ludwig. *RECODER Homepage.* `http://recoder.sf.net`, 2001.

[7] John McCarthy. Lisp 1.5. *Communications of the ACM*, 3(4), 1960.

[8] *Corba 2.4.2 Specification.* OMG, `http://www.omg.org/technology/documents/formal/corbaiiop.htm`.

[9] *Enterprise JavaBeans 1.1 Specification.* SUN Microsystems, `http://java.sun.com/products/ejb/docs.html`.

[10] *JavaBea(tm) 1.0.1 Specification.* SUN Microsystems, `http://java.sun.com/products/javabeans/docs/spec.html`, 1997.

[11] *Simple Object Access Protocol (SOAP) 1.1.* W3C Note 08 May 2000, `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`, 2001.

[12] *Web Services Description Language (WSDL) 1.1.* W3C Note 15 March 2001, `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, 2001.

[13] *Extensible Markup Language (XML) 1.0.* W3C Recommandation, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[14] *XML Schema Part 1: Structures.* W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502`, 2001.

[15] *XML Schema Part 2: Datatypes.* W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-2-220010502`, 2001.