IEEE distributed systems Expert-authored articles and resources ONLINE
October 2006 (vol. 7, no. 11), art. no. 0611-oy001

Published by the IEEE Computer Society

Middleware
Spoon: Compile-time Annotation Processing for Middleware

Renaud Pawlak •INRIA and Rensselaer Polytechnic Institute

Spoon is a Java-based program analysis and transformation tool for compile-time annotation processing. It combines compile-time reflection with a pure Java template framework for well-typed and intuitive fine-grained metaprogramming, which is applied to the middleware context.

Applications running on middleware layers and infrastructures need a great deal of configuration and deployment information to parameterize the middleware services for specific uses. *Deployment descriptors* let programmers tune the applications in a declarative way and decouple the middleware's functional from its nonfunctional aspects. However, deployment descriptors induce redundant information for attaching the deployment data to the application's objects or components, so they aren't well suited to iterative and collaborative development. In such development, programmers must incrementally modify the program in one place to get the expected effect without using complex tools to maintain consistency between the different parts of the application.

Recently, the .NET platform and Java 5 introduced annotations (or metadata), which let programmers tag an application with deployment data. Annotations are an interesting alternative to deployment descriptors because they're natively supported in a typed and integrated way, making configuration more straightforward by limiting structural information redundancy. Annotations and metadata in general have proven to be extremely useful for ensuring better separation of concerns[1–3] and for optimization.[4] By defining the right annotations, you can raise the program's abstraction level and talk about intentions rather than having to use complex middleware-level APIs, making the program less coupled to a particular technology. However, you must process the annotations to modify the program's semantics and tune them with regard to a given execution context. In middleware, this tuning can lead to fine-grained adaptations, which can entail reorganizing the program's structure. For example, in highly distributed environments, objects can be split into several subobjects to enhance distribution. In lightweight environments, methods or attributes can be seamlessly removed or replaced by lighter implementations. Also, the processing can create new classes (such as remotable proxies) to deal with middleware-specific concerns.

In this context, middleware developers need intrusive, fine-grained program-processing techniques. Less intrusive techniques such as aspect-oriented programming (AOP) can't cover all the possible fine-grained adaptations the middleware requires because they adapt the program's behavior given a pre-existing structure. Reflective program-transformation techniques are one alternative.[5–7] However, reflective techniques lower the program generators' type safety and make it difficult to trace errors, which are most often detected on the generated program at deployment time or runtime. Finally, template-based metaprogramming techniques are a good candidate for processing annotations.[8–10] (See also the "Related Work in Java" sidebar). However, neither Java nor C# offers native support for templates, making these techniques harder for the typical programmer to use and integrate. In particular, the need to use a dedicated language to express template-based transformations makes validating the generators upfront harder because the template compiler should reimplement all the Java syntactic, typing, and semantic checks—a nontrivial task.

Spoon (http://spoon.gforge.inria.fr/) is a Java 5 program-transformation framework for intrusive fine-grained compile-time annotation processing.[3] It combines compile-time reflection and pure Java templates to ensure upfront compile-time syntactic, typing, and semantic validation without the need for an extra compiler or language.

Annotations

In Java 5, programmers can define a new annotation type similarly to a new class or interface. An annotation type's goal is to define some metadata that will be attached to some program elements (such as classes, fields, and methods). For instance, a programmer can define an annotation to limit the number of elements in a stack so that you need no subclassing or parameterization Java code to ensure that the used stack is bounded. To do so, the programmer can define the following annotation type:

```
public @interface Bound {
    int max() default 10;
}
```

After defining this annotation type, the programmer can instantiate it by annotating the stack class:

```
01 @Bound(max=5)
02 public class Stack<T> {
03   List<T> elements=new Vector<T>();
04   public void push(T element) {
05     elements.add(0,element);
06   }
07   public T pop() {
08     T element=elements.get(0);
09     elements.remove(0);
10     return element;
11 } }
```

In this program, an annotation of the `Bound` type is instantiated and attached to the `Stack` class (lines 1 and 2). The annotation values are initialized with the arguments given during the annotation's construction (here, `max` is the only argument).

Like deployment descriptors, well-defined annotations can raise a given program's abstraction level because the end user will only have to talk about abstract and declaratively expressed intentions. In the previous code example, the programmer only expresses the desire that the stack be bounded. How it's implemented and how the errors are reported don't matter. The final implementation can vary depending on the execution environment, ensuring better separation of concerns. Hence, annotations are particularly useful in the middleware context, where deployment can target different environments. In general, intentions should be ensured, while their actual implementations vary. Although annotations are new in Java, you can find many examples of using annotations as deployment information. In this article, I show two of the most recent and well-known annotation-based APIs: service component architecture and Enterprise JavaBeans 3. However, uses aren't restricted to these examples.

Service component architecture

SCA (http://www-128.ibm.com/developerworks/library/specification/ws-sca) is a set of specifications describing a model for building applications and systems using a service-oriented architecture (http://www-128.ibm.com/developerworks/webservices). SCA encourages an SOA organization of business application code based on components that implement business logic and that offer their capabilities and consume functions offered by other components through service-oriented interfaces, or *service references*. SCA is built on open standards, such as the Web Services Description Language for describing and exchanging service-oriented interfaces in a language-independent manner.

Implementation-level specifications of SCA are available in C++ and Java. In particular, the Java specification normalizes the use of annotations as an easy way to define service-oriented interfaces in pure Java and to link the implementation with the services' interfaces. For example, the following snippet shows the service interface and the implementation class of an SCA component implemented in Java:

```
1 public interface HelloService {
2   String hello(String message);
3 }
4
5 @Service(HelloService.class)
6 public class HelloServiceImpl
7     implements HelloService {
8   public String hello(String message) {...}
9 }
```

I use the `@Service` annotation to declare that a given Java class is an implementation of a given service-oriented interface, also defined in Java (line 5).

Enterprise JavaBeans 3

The EJB 3 specifications (http://java.sun.com/products/ejb/docs.html) aim to simplify the EJB standard, which many Java programmers consider hard to use. In particular, they encourage using annotations to replace complex, verbose XML deployment descriptors. You can define most EJBs as plain old Java objects (POJOs) with inline deployment information and reduce the need for implementing or using middleware-specific services. In addition to the core EJB component model, EJB 3 specifications provide an annotation-based persistence specification, which includes object/relational (O/R) mapping facilities for POJO Entity beans.

The code in figure 1 shows the implementation of two Entity beans (classes annotated with `@Entity` (lines 1 and 17) that are related through a one-to-many relationship. The two relationships' roles are defined by annotating the properties' getters with the `@ManyToOne` (line 4) and `@OneToMany` (line 19) annotations.

```
01 @Entity
02 public class Employee {
03   private Department department;
04   @ManyToOne
05   public Department getDepartment() {
06     return department;
07   }
08   public void setDepartment(
09     Department department) {
10     this.department = department;
11   }
12   private ssNum;
13   @Id
14   public String getSsNum() { ... }
15 ...
16 }
17 @Entity
18 public class Department {
19   @OneToMany(mappedBy="department")
20   public Collection<Employee>
getEmployees()...
21   @Id
22   public String getId() { ... }
23  ...
24 }
```

Figure 1. Implementation of two Entity beans related through a one-to-many relationship.

Spoon

Although you can use Java 5 annotations as deployment information for middleware, you must process annotations to implement their actual semantics (otherwise, they remain meaningless decorations). You can use the Spoon framework to perform this function.

Compile-time processing

One of Spoon's main goals is to let programmers specify pure Java program transformations, which they can parameterize with Java 5 annotations. Spoon is an open compiler built on the top of the Java programming language compiler (javac) and that uses compile-time reflection.[11] Compile-time reflection is a branch of metaprogramming that lets programmers access and modify the program's *abstract syntax tree* by writing metaprograms in the target language (Java in this case). To do this, Spoon provides users with a representation of the Java AST, or metamodel, which allows for both reading and writing. Each metamodel interface is a compile-time program element (`CtElement`), which represents an AST node.

As figure 2 shows, once Spoon has built a metamodel using javac's AST, a processing phase occurs on the metamodel. A *visitor pattern* implements the processing. The visitor scans each visited program element and can apply user-defined processing jobs, or *processors*. Processing can occur in several rounds until no more processing actions are to be applied. Although not mandatory (you can use Spoon only for program analysis), a last processor usually generates the processed Java program.

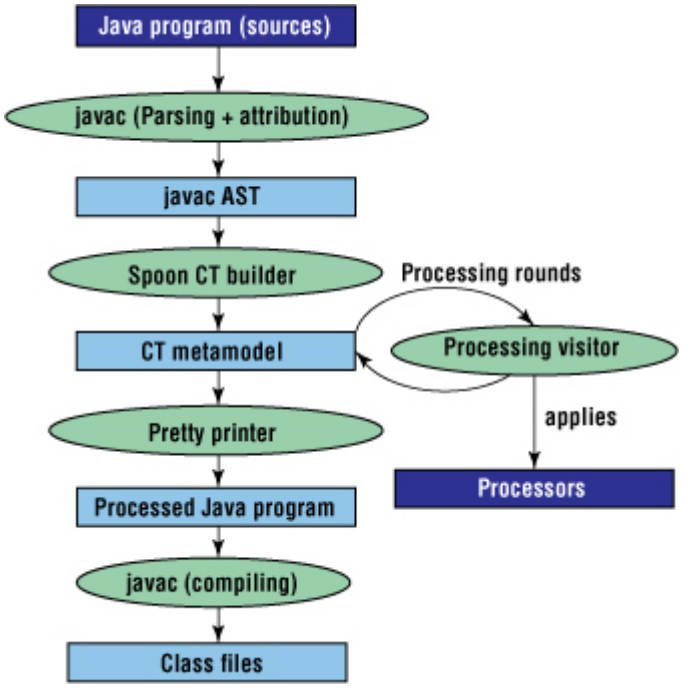

Figure 2. Spoon compile-time processing.

Compile-time reflection is a flexible, intrusive technique that gives the program access for reading and modification. Because Spoon provides a full reification, you can implement any kind of transformation. For instance, you can add or remove interfaces, methods, or fields, and manipulate method bodies, statements, and expressions. Also, through the factory, you can generate new classes or interfaces from scratch. This fine-grained API makes it particularly useful for implementing fine-grained analysis and transformations. For instance, with Spoon, users can determine if a method modifies or reads a given field or detect constant fields, parameters, and literals for optimizing or simplifying code.

User-defined processors

The Spoon framework defines an API for program processing in general. The main interface is the `Processor` interface:

```
public interface Processor<E extends CtElement> {
  void process(E element);
}
```

Spoon users can implement this interface to define a new code processor. As I explained earlier, at compile-time Spoon visits the program's AST and calls the `process` method of all the registered processors. The processors can then perform any program check or transformation using the currently visited element.

Spoon can also trigger transformations based on program annotations. To do so, programmers must implement the following interface:

```
01 public interface AnnotationProcessor
02     <A extends Annotation, E extends CtElement>
03     extends Processor<E> {
04   void process(A annotation,
05                E element);
06   Collection<Class<? extends Annotation>>
07     getProcessedAnnotationTypes();
08   Collection<Class<? extends Annotation>>
09     getConsumedAnnotationTypes();
10 }
```

An *annotation processor* processes annotated elements, taking the currently processed annotation as an additional parameter (line 4). It also defines a set of processed annotations that let Spoon know when to trigger the processor (lines 6–7). When annotation processing leads to annotation consuming, the processor should remove the annotations from the program. To do so automatically, the programmer should define the consumed annotations by implementing the `getConsumedAnnotationTypes` method (lines 8–9). For simplification, Spoon provides default abstract implementations of these interfaces.

Writing an annotation processor

Here, I implement a simple processor for the `Bound` annotation defined earlier. This processor transforms the annotated stack classes so that the `push` method implements a test to check that the `elements`' stack size is bounded by the `max` value. The processor in this example is specific to the stack, but you could generalize it by adding parameters to the `Bound` annotation. I define the processor for the example as follows:

```
1 public class BoundProcessor extends
2     AbstractAnnotationProcessor<Bound,CtClass> {
3   public void process(Bound b, CtClass<?> c) {
4     CtMethod<?> push = c.getMethod("push",
5       getFactory().Type().createReference("T"));
6     push.getBody().insertBegin(
7       Substitution.substituteMethodBody(c,
8         new TestTemplate(b.max()),"test"));
9 } }
```

This is a typical metaprogram that works at compile time. I get the compile-time representation of the `push` method as a `CtMethod` element (lines 4–5). I then use the `CtBlock.insertBegin` method to modify the `push` method's body so that the new body consists of the test expression followed by the old body expression (line 6). The test expression is a `CtBlock`, which the italicized code (lines 7–8) builds using a Spoon template representing the code "`if(element.size()>_max_) throw new RuntimeException("overflow")`."

When I apply the processor, Spoon automatically removes the `Bound` annotation from the processed program because `Bound` is defined as a consumed annotation. Thus, the final processed code for the stack is:

```
public class Stack<T> {
  public void push(T element) {
    if (elements.size()>=5) {
      throw new RuntimeException("overflow");
    }
    elements.add(0,element);
  }
 ...  // the rest remains unchanged
}
```

Spoon templates

As I mentioned earlier, compile-time reflection and metaprogramming let developers analyze or modify a program by manipulating its AST. However, metaprogramming can become complex when it involves executable code (method and constructor bodies). When this granularity of metaprogramming is required, programmers can use code templates to express patterns for generating code, such as in generative programming.[8]

Several languages already provide templates. Among the best-known are the C++ templates[10] and template Haskell.[12] Templates can also be provided as a preprocessing facility for Java.[9] However, Java doesn't provide a native template mechanism.

Spoon exploits Java 5's features to let programmers define code templates in pure Java using a template framework. The advantage of specifying templates in pure Java is that a regular compiler can ensure standard syntactic, typing, and semantic checks upfront. Consequently, programmers can write templates in their favorite Java IDE and exploit its advantages (incremental compilation, completion, syntax highlighting, contextual help, refactoring, wizards, and so on).

A Spoon pure-Java template is a regular Java class containing *template parameters* (or variation points). These template parameters are defined as fields annotated with `@Parameter`. Template parameters can represent primitive values (such as literal values, program element's names, and types) or actual program elements (`CtElement`). In the template code, programmers can substitute all references to template parameters by their actual values using Spoon's substitution engine API.

Although you can access primitive template parameters directly, nonprimitive template parameters must implement the `TemplateParameter<T>` interface, where `T` is the actual parameter type. Within a template, the programmer can reference a nonprimitive template parameter by invoking `TemplateParameter.S()` on the parameter. The return type of `S()` is `T`, which allows for the definition of well-typed template expressions. For more details on templates, please refer to the Spoon online resources (http://spoon.gforge.inria.fr/).

Referring to the annotation processor example, we can write the template code that checks for the max bound of the stack as follows:

```
01 public class TestTemplate extends Template {
02   @Parameter int _max_;
03   public TestTemplate(int max) { _max_=max; }
04   @Local Collection elements;
05   public void test() {
06     if (elements.size()>=_max_) {
07       throw new RuntimeException("overflow");
08     }
09   }
10 }
```

The only template parameter is `_max_` (line 2), initialized by the template's constructor. We can substitute this parameter with its actual value using the substitution engine (as I described in the previous section). The `elements` collection (line 4) is a local representation of the `elements` field defined by the stack. Another template parameter could replace this field to make it less specific.

Intrusive program processing with Spoon

As I stated in the introduction, the middleware context requires intrusive program processing. For instance, if I want to deploy a program in a portable device, intrusive program processing would generally allow for better optimizations. By combining compile-time reflection, which gives access to the full program AST, and pure Java templates, Spoon makes it possible to implement intrusive program processing, which can be guided by well-defined annotations. The stack example demonstrates intrusive program processing by introducing code that checks for the bound directly within the push method. Less intrusive techniques such as proxy-based or aspect-oriented methods would require using a delegation technique under the hood. This introduces infrastructural code using intermediate objects and leads to less efficient code (in regard to CPU and memory consumption).

With Spoon, you can implement optimizations that target sensible environments. For instance, to limit memory or network bandwidth consumption, you can define an annotation `RemoveIfUnused` to let the user specify whether a class or a method should be deployed when a given program isn't using them directly. A Spoon processor can then check whether references to the annotated elements exist and remove the elements if none are found. This example falls into the broader category of late deployment, a middleware service targeting distributed and heterogeneous environments. Spoon also lets you implement the program's compile-time analysis and pull forward some deployment decisions that are normally made at deployment time or runtime, thus limiting the risks of late errors in the development process.

Applications

To demonstrate Spoon applications for middleware, I use the SCA and EJB 3 examples I presented earlier. I also show a proxy example that illustrates the utility of well-typed Java processing.

Processing SCA annotations

As I showed earlier, programmers can implement service-oriented components in Java using the service annotation. The Java implementation corresponds to an XML-described component type that, according to the SCA specifications, should be automatically generated by some tool or by the service-oriented middleware itself. A simple way to generate this XML component type with the Spoon compile-time reflection API is as follows:

```
01 public class SCAServiceProcessor extends
02  AbstractAnnotationProcessor<Service,CtClass> {
03   public void process(Service s,CtClass<?> c) {
04     List<CtInterface<?>> itfs=getItfs(s);
05     for(CtInterface<?> itf:itfs)
06       writeInterface(itf);
```

```
07      writeImplementation(c);
08    }
09    // returns the interfaces
10    List<CtInterface<?>> getItfs(Service s) {...}
11    // write methods (typical XML output)
12    ...
13  }
```

For each processed `@Service` annotation, the processor looks up all the implemented interfaces (`getItfs` in line 10) and creates the corresponding component type descriptors (lines 5–6). It also creates a default SCA module (named after the package containing the implementation) that defines the component's implementation (line 7).

In addition to generation, an important feature is the validation of the annotated programs. Spoon processors can also ensure that annotations are used according to the specification. For instance, SCA specifications stipulate that using `@Service` without indicating an interface is meaningless; in this case, the middleware layer should simply ignore the `@Service` annotation. However, because it's a misuse of the annotation, warning the programmer that the annotation has no effect would be helpful. With Spoon, you can report a warning to the compile-time environment by adding the following statement in the `process` method:

```
if(itfs.size()==0) {
  getFactory().Environment().reportWarning(
     "annotation has no effect", impl, null,
     s.getPosition());
  return;
}
```

Processing EJB 3 annotations

Application servers that support EJB 3 provide a middleware layer that can process EJB 3 annotations and automatically integrate Java 2 Enterprise Edition (J2EE) services into the component implementations. Programmers can do this using several techniques, such as dynamic proxies and load-time bytecode instrumentation.[5,6] However, processing annotations at compile time is interesting for two main reasons:

- In a well-known execution context (that is, one with available libraries, available resources, component location, and so on), you can tune and optimize the processed code with statically known information.
- You can validate and check the deployment information several times during development, minimizing the risk of deployment-time and runtime errors.

For example, assume we need a light implementation of our Entity beans, which should be local and persistent. No transactions are needed, and they're mostly read and rarely written. In that case, we don't need a full-blown J2EE infrastructure and can easily imagine a simple implementation that would use, for instance, the native Java serialization. We can then implement the Entity `Employee` as follows:

```
01 public class Employee implements Serializable {
02   private Department department;
03   public Department getDepartment() {
04     return department;
05   }
06   public void setDepartment(
07     Department department) {
08     this.department = department;
09     Serialization.serialize(getSsNum(),this);
```

```
10   }
11   private ssNum;
12   public String getSsNum() { ... }
13   ...
14 }
```

This implementation is simple because it only requires the programmer to implement the Serializable interface (line 1) and to serialize the object after a setter has been executed (line 9). Figure 3 shows how Spoon implements this transformation.

```
public class SimpleEjbEntityProcessor extends
  AbstractAnnotationProcessor<Entity,CtClass> {
  public void process(Entity e,CtClass<?> c) {
    // add Serializable
    c.getInterfaces().add(getFactory().Type
     .createReference(Serializable.class));
    for(CtMethod<?> setter:getPropSetters(c)) {
      Template t=new SerTemplate(getIdName(c));
      setter.getBody().insertEnd(
         Substitution.substituteMethodBody(c,
            t,"serialize"));
    }
  ...
}
public class SerTemplate implements Template {
  @Parameter TemplateParameter<Object> _getId_;
  public SerTemplate(String id) {
    // initialize _getId_ to an invocation to
    // the id getter (using Spoon meta-model)
    ...
  }
  void serialize() {
    Serialization.serialize(_getId_.S(),this);
  }
}
```

Figure 3. Spoon's implementation of a Serializable based persistence.

Of course, because we're using the EJB 3 annotation-based standard specification, we can drop compile-annotation processing anytime to let a more generic middleware layer process the annotations (for instance, J2EE application servers).

Generating well-typed proxies

Instead of using reflective techniques to create remotable proxies for user-defined services, programmers can use pure Java templates, such as those that Spoon provides, to implement the proxies. Native templates' main advantage is that they enforce strong typing links with the APIs they use. Consequently, native compilers and IDEs will be able to detect any typing mistake without having to compile the generated code (as is often the case with generative techniques built on the top of Java, or any language that doesn't support well-typed templates).

The code snippet in figure 4 is a simple example of a Spoon Java template that allows for the generation of transactional delegators. You can easily extend it to implement any kind of proxy.

Redundancy of information is required to handle the methods returning a value (line 15) and `void` methods (line 25). This is because of the Java syntax's lack of uniformity and is part of the price for enabling pure Java templates without having to modify existing compilers. Also, the programmer needs to define the generic intermediate types `_ReturnType_` and `_DelegateType_` on compilation purpose (lines 35–41). Although programming templates with Spoon sometimes requires you to define intermediate and redundant information that a native template language wouldn't need, using typed templates in pure Java is preferable to using reflection, which easily leads to typing or even syntactic and semantic errors that can be extremely hard to debug. In particular, the template strongly references the transactional API (lines 13, 17, 20, 27, and 29). This ensures, for instance, that all of the passed parameters are well-typed and all of the typed exceptions are correctly handled or forwarded.

```
01 public class TxDelegator implements Template {
02   // variation points
03   @Parameter CtTypeReference _DelegateType_;
04   @Parameter CtTypeReference _ReturnType_;
05   @Parameter String _delegate_;
06   @Parameter String _voidDelegate_;
07   @Parameter List<CtParameter> _args_;
08   // template definition
09   _DelegateType_ delegate;
10   TxManager tm;
11   public Delegator(_DelegateType_ delegate) {
12     this.delegate = delegate;
13     tm=TxManager.get();
14   }
15   public _ReturnType_ _delegate_(
16       List<CtParameter> _args_) {
17     Tx tx=tm.start();
18    _ReturnType_ res=delegate._delegate_(
19         _args_);
20     try { tm.commit(tx); } catch(Exception e) {
21       tm.rollback(tx);
22     }
23     return res;
24   }
25   public void _voidDelegate_(
26       List<CtParameter> _args_) {
27     Tx tx=tm.start();
28     delegate._voidDelegate_(_args_);
29     try { tm.commit(tx); } catch(Exception e) {
30       tm.rollback(tx);
31     }
32   }
33 }
34 // intermediate types definitions
35 interface _ReturnType_ {}
36 interface _DelegateType_ {
37   _ReturnType_ _delegate_(
38      List<CtParameter> _args_);
39   void _voidDelegate_(
40      List<CtParameter> _args_);
41 }
```

Figure 4. Simple example of a Spoon Java template allowing for the generation of transactional delegators.

Evaluation

Spoon brings together two important techniques in Java: compile-time reflection and templates. However, it also has some limitations.

Integrated framework

None of the elements I present in this article are inherently new. For annotation processing, XDoclet and Sun's annotation processing tool (http://java.sun.com/j2se/1.5.0/docs/guide/apt) let programmers write annotation processors that are similar to Spoon's but don't work on a full Java AST. This creates important limitations (for example, the method bodies are unavailable for processing). For compile-time reflection, numerous tools and open compilers reify the AST.[5–7,13] Finally, some languages and tools support templates for generative programming.[9,10,12] However, Spoon brings together all of these techniques within an integrated, well-typed Java environment. Compile-time reflection combined with pure Java templates (which use Java generics for typing) allow for fine-grained, well-typed program analysis and transformation without requiring the use of a new language. These features, coupled with comprehensive, integrated support for Java 5 annotation processing, make Spoon an interesting candidate for projecting Java programs on middleware environments at compile time, thus minimizing the underlying infrastructural code.

Limitations

Spoon doesn't support automatic processor or template composition as advanced program-transformations tools do (see the sidebar), but leaves this task to the programmers. Nevertheless, it provides several processing strategies and a comprehensive API for ordering the processors and defining new strategies.

Using regular Java to express templates introduces some limitations. For example, it's hard, if not impossible, to program uniform templates to deal with the instrumentation of void methods and those returning a typed value. The same issue arises with static and nonstatic methods. Also, pure-Java templates' variation points are limited to specific places, making them less powerful than general template approaches. However, you can overcome these limits with compile-time reflection and annotations. These let the processors adapt the templates to a given use context, but make some aspects of Spoon metaprogramming more complex than a template-dedicated language, such as C++[10] and Haskell.[12] The templates are expressed in Java syntax, counterbalancing this relative complexity. Dedicated approaches, on the other hand, often require using constructions that can be far from the base language's programming style and make template-based programs hard for regular programmers to write.

Spoon metaprograms are as well-typed as Java programs when using templates and ensure consistent output. When using compile-time reflection, the typing is weak, although partially ensured by Java 5 generics. Ongoing work seeks to provide Spoon processors that will be applied to the Spoon reflection-based metaprograms to statically validate them and ensure that the output program is consistent and well-typed.

I'm currently working on several applications of Spoon for program validation and template-based AOP. I'm especially investigating how to integrate Spoon with a symbolic evaluator of the metamodel, which would help to implement more powerful static validations when needed.

References

1. J. Cachopo, "Separation of Concerns through Semantic Annotations," *Proc. Conf. Object-Oriented Programming Systems Languages and Applications* (OOPSLA), ACM Press, 2002, pp. 2–3.
2. G. Kiczales and M. Mezini, "Separation of Concerns with Procedures, Annotations, Advice, and Pointcuts," *Proc. European Conf. Object-Oriented Programming* (ECOOP), LNCS 3585, Springer, 2005, pp. 195–213.

3. R. Pawlak, "Spoon: Annotation-Driven Program Transformation—The AOP Case," *Proc. 1st Workshop Aspect-Orientation for Middleware Development*, ACM Int'l Conf. Proc. Series, vol. 118, article no. 6, ACM Press, 2005.
4. J. Hummel et al., "Annotating the Java Bytecodes in Support of Optimization," *Concurrency, Experience, and Practice*, vol. 9, no. 11, 1997, pp. 1003–1016
5. E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a Code Manipulation Tool to Implement Adaptable Systems," *Adaptable and Extensible Component Systems*, 2002, http://asm.objectweb.org/current/asm-eng.pdf (pdf).
6. S. Chiba, "Load-Time Structural Reflection in Java," *Proc. 14th European Conf. Object-Oriented Programming* (Ecoop), LNCS 1850, Springer, 2000, pp. 313–336.
7. M. Tatsubori et al., "OpenJava: A Class-Based Macro System for Java," *Proc. 1st Oopsla Workshop Reflection and Software Eng.* (OORaSE), LNCS 1826, Springer, 1999, pp. 117–133.
8. K. Czarnecki and U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
9. D. Nizhegorodov, "Jasper: Type-Safe Compile-Time Reflection Language Extensions and MOP-Based Templates for Java," *Proc. Workshop Reflection and Metalevel Architectures*, *14th European Conf. Object-Oriented Programming* (ECOOP), LNCS 1850, Springer, 2000.
10. D. Vandervoorde and N.M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2002.
11. S. Chiba, "A Study of Compile-Time Metaobject Protocol," PhD dissertation, Graduate School of Science, Univ. of Tokyo, 1996.
12. T. Sheard and S.P. Jones, "Template Meta-Programming for Haskell," *Haskell 02: Proc. 2002 ACM Sigplan Workshop on Haskell*, ACM Press, 2002, pp. 1–16.
13. S. Chiba, "A Metaobject Protocol for C++," *Proc. Conf. Object-Oriented Programming Systems Languages and Applications* (OOPSLA), ACM Press, 1995, pp. 285–299.

Renaud Pawlak is an assistant professor of computer science at Rensselaer Polytechnic Institute, Hartford campus. He's also a computer science researcher and leader of the Spoon project at INRIA JACQUARD project. His research focuses on the separation of concerns (SoC) in software engineering, reflection, open languages, distributed middleware, and aspect-oriented programming. He's the main author of the JAC Open Source framework, which is part of ObjectWeb. He cofounded the AOPSYS company with the primary goal of supporting the use of AOP and SoC-related industry techniques. Pawlak has a PhD in computer science from CNAM, Paris. Contact him at Rensselaer at Hartford, 275 Windsor St., Hartford, CT 06120; pawlar@rpi.edu or renaud.pawlak@inria.fr.

Sidebar: Related Work in Java

Research groups are doing important work in aspect-oriented programming. Several Java 2 Enterprise Edition frameworks, such as JBoss (http://www.jboss.org/products/aop) and Spring,[1] support AOP, which allows for parameterized integration of middleware-related concerns in end-user business-oriented Java programs. However, these frameworks aren't well-typed (dynamic AOP) and rely on a great deal of infrastructure, making them specific to a given use context and significantly impacting performance in lightweight environments.

AOP language approaches such as AspectJ[2] and LogicAJ[3] allow for better-typed AOP. The latest version of AspectJ, for instance, lets developers concisely integrate annotation-driven concerns but limits program transformations to introductions and incremental behavioral modifications. This isn't sufficient when implementing generic aspects needing more intrusive transformations,[4] as is often the case in dedicated middleware. LogicAJ, on the other hand, supports more fine-grained and intrusive program transformations. It's based on logic metaprogramming, so it's sound and powerful. However, the logic behind it is beyond what typical Java programmers know.

With the Spoon pragmatic approach, which uses pure Java, developers can't express metaprograms as concisely as with AOP—especially a logic metaprogramming-based approach—but Spoon offers direct

IDE support and is generally more intuitive to Java programmers. Also, AspectJ and LogicAJ both require some runtime libraries to run the woven program, which can be a problem when deploying the application in constrained environments. Spoon's generative approach doesn't require any specific libraries at runtime, other than those of the targeted environment. In general, generative techniques and AOP are complementary: you can use AOP for behavioral and incremental transformations with well-separated concerns, and you can use Spoon when AOP reaches its limits and when you need more intrusive processing.

Finally, Java offers other template-based program-transformation tools. Stratego enables logic rewriting of Java programs using the concrete object syntax.[5] MetaJ combines templates and reflection for Java metaprogramming.[6] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbuuse an XML-based approach to write templates and target J2EE middleware applications.[7]

Contrary to Spoon, these approaches require using non-Java elements for a preprocessing phase. This makes it hard to validate the metaprograms upfront and ensure correct use of the targeted middleware libraries. Consequently, the tools detect some errors in a second compilation phase, and tracing them back to the metaprograms can be difficult. This occurs less frequently with Spoon, which ensures the correct use of the targeted libraries upfront.

References

1. R. Johnson, "Spring AOP: Aspect Oriented Programming with Spring," *Spring Reference*, chap. 6, www.springframework.org/docs/reference/aop.html.
2. G. Kiczales et al., "An Overview of AspectJ," *Proc. 15th European Conf. Object-Oriented Programming* (Ecoop), LNCS 2072, Springer, 2001, pp. 327–353.
3. T. Rho, G. Kniesel, and M. Appeltauer, "Fine-Grained Generic Aspects," *Proc. Workshop Foundations of Aspect-Oriented Languages* (FOAL), *5th Int'l Conf. Aspect-Oriented Softwar*e Development (AOSD), 2006, www.cs.iastate.edu/~leavens/FOAL/papers-2006/proceedings.pdf (pdf).
4. G. Kniesel, T. Rho, and S. Hanenberg, "Evolvable Pattern Implementations Need Generic Aspects," *Proc. Workshop Reflection, AOP, and Meta-Data for Software Evolution*, *European Conf. Object-Oriented Programming* (ECOOP), 2004, http://roots.iai.uni-bonn.de/research/logicaj/downloads/papers/KnieselRhoHanenberg-RAM-SE04.pdf (pdf).
5. M. Bravenboer et al., "Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax," *Proc. Generative Programming and Component Eng.* (GPCE) (http://www.gpce.org/), Springer, 2005, pp. 157–172.
6. A. Alvarenga de Oliveira et al., "MetaJ: An Extensible Environment for Metaprogramming in Java," *J. Universal Computer Science*, vol. 10, no. 7, 2004, pp. 872–891.
7. E. Wohlstadter, S. Jackson, and P. Devanbu, "DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems," *Proc. Int'l Conf. Software Eng.* (ICSE), IEEE CS Press, 2003, pp. 174–186.

Related Links

- DS Online's Middleware Community
- "AVal: an Extensible Attribute-Oriented Programming Validator for Java," *6th Int'l Workshop Source Code Analysis and Manipulation* (SCAM 06)
- "Applying AspectJ to J2EE Application Development," *IEEE Software*

Cite this article:
Renaud Pawlak, "Spoon: Compile-time Annotation Processing for Middleware," *IEEE Distributed Systems Online*, vol. 7, no. 11, 2006, art. no. 0611-oy001.