Spoon: A Library for Implementing Analyses and Transformations of Java Source Code

Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, Lionel Seinturier

Spoon: A Library for Implementing Analyses and Transformations of Java Source Code

Renaud Pawlak Martin Monperrus Nicolas Petitprez
Carlos Noguera Lionel Seinturier

Abstract

This article presents Spoon, a library for the analysis and transformation of Java source code. Spoon enables Java developers to write a large range of domain-specific analyses and transformations in an easy and concise manner. Spoon analyses and transformations are written in plain Java. With Spoon, developers do not need to dive into parsing, to hack a compiler infrastructure, or to master a new formalism.

1 Introduction

Compilers and interpreters analyze source code. But source code analysis is used in many more places [6]: it is used to compute metrics [17], to detect bad smells [18], to detect code clones [20]. Companies and open-source projects set up their own metrics and coding conventions [16]. This motivates a library for source code analysis that is usable by the masses of developers and not dedicated to compiler hackers.

Beyond source code analysis, there is source code transformation. Source code transformation is a program transformation at the source code level, as opposed to program transformation done on binary code [8]. There are many usages of program transformation: profiling [41], security [11], optimization [28], refactoring [24]. As source code analysis, some source code transformations are written by normal Java developers. For instance, this happens when the transformation uses domain-specific knowledge [5].

This article presents Spoon, a library for the analysis and transformation of Java source code. Spoon enables Java developers to write a large range of domain-specific analyses and transformations in an easy and concise manner. Spoon analyses and transformations are written in plain Java. With Spoon, developers do not need diving to parse, to hack a compiler infrastructure, or to master a new formalism.

The main features of Spoon are:

1. a Java metamodel for representing Java abstract syntax trees (AST) which is both easy to understand and easy to manipulate;
2. a first-class intercession programming interface (intercession API) to modify and generate Java source code;
3. the use of generic typing for static checking of the analyses and transformations;

4. the native and seamless integration and processing of Java annotations;

5. a pure Java statically-checked templating engine.

Taking these features all together, SPOON is unique for the following reasons. Feature #1 and #2 are not the focus of compiler infrastructures. We are not aware of leveraging generics for AST manipulation (Feature #3). Java annotations have been extensively discussed [14] but generic annotation processors are scarce (Feature #4). While many templating engines exists (e.g. Apache Velocity[1]), none provide static checking as our plain Java templates provide. The related work section 5 deepens those points.

This paper supersedes INRIA technical report #5901 [36], which has been completely rewritten. It contains a better explanation of the concepts using appropriate illustrative examples, the evaluation contains a section of the correctness of SPOON as well as three new case studies, and the related work is thoroughly analyzed (including the most recent papers).

This article reads as follows. Section 2 discusses the foundations of source code analysis in SPOON (the metamodel and the queries). Section 3 presents our mechanisms for transforming source code. Section 4 exposes case studies of fruitful usages of SPOON. Section 5 discusses the related work. Section 6 concludes and discusses future work.

2 Source Code Analysis with Spoon

The first goal of SPOON is to enable standard developers to write their own domain-specific analyses on source code. This requires: first, an intuitive metamodel understandable by the mass of Java developers (presented in Section 2.2), second, mechanisms to analyze source code elements. The latter is embodied by queries (Section 2.3) and processors for traversing the program under analysis (Section 2.4). But let us first give an overview of the library before going into the details of the Java metamodel of SPOON.

SPOON is a meta-analysis tool, it provides software engineers with the primitives to write their own analyses. As such, SPOON does not any specific analysis such as dataflow analysis.

2.1 Overview of Spoon

Figure 1 gives the overview of our approach. A Java program is given as input. It is parsed with an off-the-shelf compiler in order to produce a first abstract syntax tree (AST). Then, SPOON simplifies the AST (deleting and creating nodes), in order to provide users with an intuitive and easy-to-manipulate model of their program. This compile-time (CT) model is an instance of the SPOON metamodel. The analysis and transformation of programs are written as "program processors" and "templates". A user-defined processor performs a specific action, such as a transformation on a kind of node, under a well-defined processing condition. The processing and templating engine takes them as input and applies them to the Java model as long as elements remain to be processed. Eventually, the SPOON model is translated back to source with a pretty-printer.

[1] http://velocity.apache.org

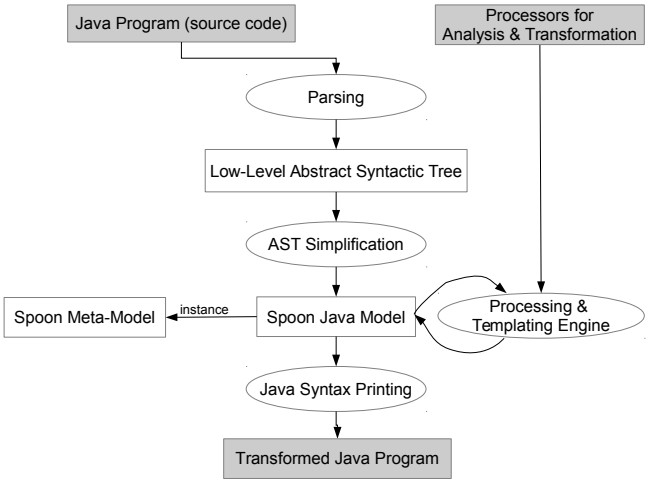

Figure 1: Overview of SPOON: Java Programs are transformed and analyzed as models of the SPOON Java metamodel..

This pretty printer preserves API comments and removed in-body comments (because they are not part of the metamodel).

2.2 The Spoon Metamodel of Java

A programming language can have different metamodels. An abstract syntax tree (AST) or model, is an instance of a metamodel. Each metamodel – and consequently each AST – is more or less appropriate depending on the task at hand. In this paper, we focus on Java and consequently on Java metamodels. For instance, the Java metamodel of Sun's compiler (javac) has been designed and optimized for compilation to bytecode, while, the main purpose of the Java metamodel of the Eclipse IDE (JDT) is to support different tasks of software development in an integrated manner (code completion, quick fix of compilation errors, debug, etc.).

Unlike a compiler-based AST (e.g. from javac), the SPOON metamodel of Java is designed to be easily understandable by normal Java developers, so that they can write their own program analyses and transformations. The SPOON metamodel is complete in the sense that it contains all the required information to derive compilable and executable Java programs (hence contains annotations, generics, and method bodies).

The SPOON metamodel can be split in three parts. The structural part (Figure 2) contains the declarations of the program elements, such as interface, class, variable, method, annotation, and enum declarations. The code part (Figure 3) contains the executable Java code, such as the one found in method bodies. The reference part models the references to program elements (for instance a reference to a type).

As shown in Figure 2, all elements inherit from CtElement which declares a parent element denoting the containment relation in the source file. For instance, the parent of a method node is a class node. All names are prefixed by "CT"

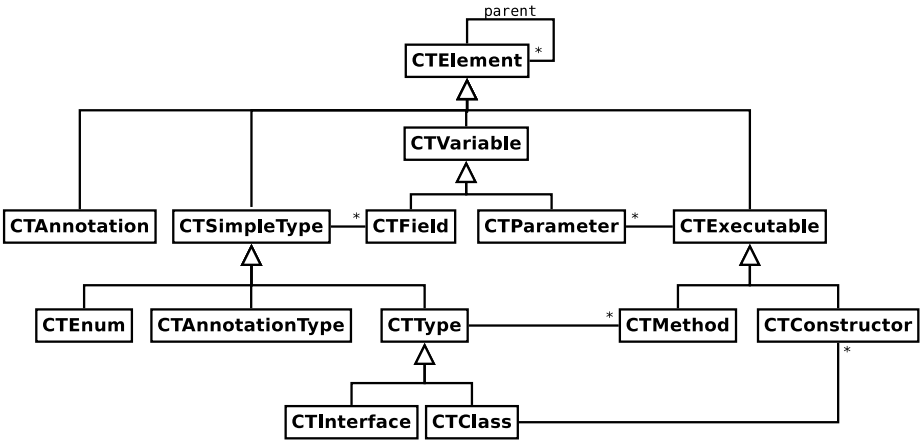

Figure 2: Excerpt of the structural part of the Spoon Java 5 metamodel.

which means "compile-time".

Figure 3 shows the metamodel for Java executable code. Because of the complexity of the Java language, the code metamodel figure contains only an excerpt of all classes. There are two main kinds of code elements. First, the statements (`CtStatement`) are untyped top-level instructions that can be used directly in a block of code. Second, the expressions (`CtExpression`) are used inside the statements (for sake of readability, this can not be seen on the figure). For instance, a `CtLoop` (which is a statement) points to a `CtExpression` which expresses its boolean condition. Some code elements such as invocations and assignments are both statements and expressions (multiple inheritance links). Concretely, this is translated as an interface `CtInvocation` inheriting from both interfaces `CtStatement` and `CtExpression`. The generic type of `CtExpression` is used to add static type-checking when transforming programs. This will be explained in details in Section 3.2.

The reference part of the metamodel expresses the fact that program references elements that are not necessarily reified into the metamodel (they may belong to third party libraries). For instance, an expression node returning a `String` is bound to a type reference to `String` and not to the compile-time model of String.java since the source code of `String` is (usually) not part of the application code under analysis. In other terms, references are used by metamodel elements to reference elements in a weak way. Weak references make it more flexible to construct and modify a program model without having to get strong references on all referred elements.

References are resolved when the model is built, the resolved references are those that point to classes for which the source code is available in the Spoon input path. Since the references are weak, the targets of references do not have to exist before one references them. The price to pay for this low coupling is that to navigate from one code element to another, one has to chain a navigation to the reference and then to the target. For instance, to navigate from a field to the type of the field, one writes `field.getType().getDeclaration()`.

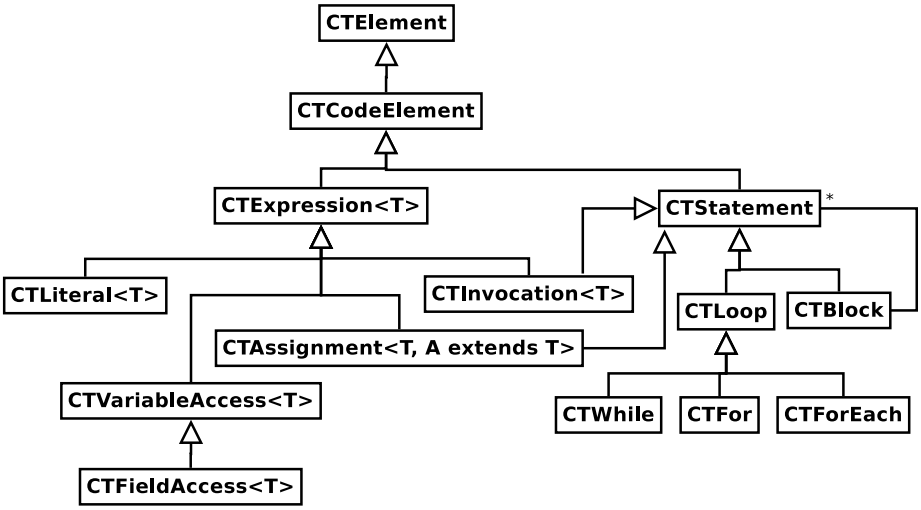

Figure 3: Excerpt of the code part of the Spoon Java 5 metamodel.

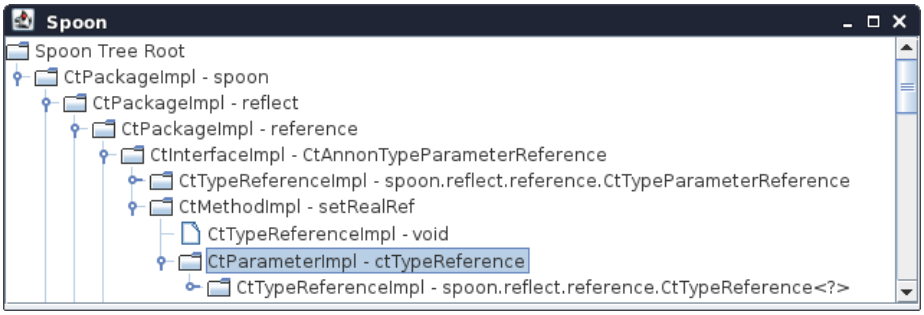

Figure 4: A GUI to understand and learn the Spoon metamodel

2.3 Quering Source Code Elements

Spoon aims at giving developers a way to query code elements in one single line of code in the normal cases. Classical research about code querying uses specific ad hoc languages [2]. On the contrary, code query is Spoon is done in plain Java, in the spirit of an embedded DSL. The information that can be queried is that of a well-formed typed AST. For this, we provide the query API, based on the notion of "Filter". A Filter defines a predicate of the form of a `matches` method that returns `true` if an element is part of the filter. A Filter is given as parameter to a depth-first search algorithm. During AST traversal, the elements satisfying the matching predicate are given to the developer for subsequent treatment. Table 1 gives an excerpt of built-in filters.

Listing 1 gives an example of client code. Three filters are used. The first returns all AST nodes of type "Assignment". The second one selects all deprecated classes. The last one is a user-defined filter that only matches public fields across all classes.

To guide the user in learning the metamodel, two pieces of information are

TypeFilter	returns all metamodel elements of a certain type (e.g. all assignment statements)
FieldAccessFilter	returns all accesses to a given field
AnnotationFilter	returns all elements annotated with a given annotation type
ReturnOrThrowFilter	returns all elements that ends the execution flow of a method

Table 1: Excerpt of Built-in Filters for Querying Source Code Elements.

```
// collecting all assignments of a method body                              1
list1 = methodBody.getElements(new TypeFilter(CtAssignment.class));         2
                                                                            3
// collecting all deprecated classes                                        4
list2 = rootPackage.getElements(new AnnotationFilter(Deprecated.class));    5
                                                                            6
// creating a custom filter to select all public fields                     7
list3 = rootPackage.getElements(                                            8
  new AbstractFilter<CtField>(CtField.class) {                              9
    @Override                                                               10
    public boolean matches(CtField field) {                                 11
      return field.getModifiers.contains(ModifierKind.PUBLIC);              12
    }                                                                       13
  }                                                                         14
);                                                                          15
```

Listing 1: Examples of Concise yet Non Trivial Queries to Collect Code Elements.

available. First, the user is given a graphical user interface, that provides a navigable view of the Spoon AST of the program under analysis. A screenshot of this GUI is given in Figure 4. Second, the metamodel is carefully designed so that the user can discover the metamodel through method calls. The object-orientation of the Spoon metamodel is appropriate for this. For instance, statement `((CtIf)method.getBody().getStatement(0)).getThenStatement()` navigates from a method object to the then statement of the first if.

2.4 Processing Code Elements

A program analysis is a combination of query and analysis code. In Spoon, this conceptual pair is reified in a "processor". A Spoon program processor is a class that focuses on the analysis of one kind of program elements. For instance, Listing 2 presents a processor that analyzes a program to find empty catch blocks.

The elements to be analyzed (here catch blocks), are given by generic typing: the programmer declares the AST node type under analysis as class generics. The processed element type is automatically inferred through runtime introspection of the processor class. There is also an optional overridable method for querying elements at a finer grain. The `process` method takes the requested element as input and does the analysis (here detecting empty catch blocks). Since a real world analysis combines multiple queries, multiple processors can be used at the same time. The launcher applies them in the order they have been declared.

Processors are implemented with a visitor design pattern [33] applied to

```
public class CatchProcessor extends AbstractProcessor<CtCatch> {
  @Override
  public void process(CtCatch element) {
    if (element.getBody().getStatements().size() == 0) {
      System.err.println("empty catch clause at "+element.getPosition());
    }
}}
```

Listing 2: Analysis Code to Detect Empty Catch Blocks. No More Code is Required to Run the Analysis on Millions of Lines of Code.

the Spoon Java model. Each node of the metamodel implements an `accept` method so that it can be visited by a visitor object, which can perform any kind of action, including modification.

Spoon *provides developers with an intuitive Java metamodel and concise abstractions to query and process AST elements.*

3 Source Code Transformation with Spoon

Spoon has been designed to facilitate source code transformation. For this, four mechanisms are provided: first-class intercession mechanisms (Section 3.1), the use of generics (Section 3.2), the notion of statically checked templates (Section 3.3), and the use of annotations (Section 3.4).

3.1 First Class Intercession Mechanisms

For transforming programs, Spoon provides first-class intercession mechanisms at different levels. At the structural level, Spoon enables one to add and remove types in packages, as well as fields and methods in types. At the behavioural level, Spoon enables one to modify any part of the code. For instance, one can add pre-conditions in methods, or add logging in catch blocks in an automated manner.

Table 2 provides an overview of the main intercession methods of Spoon. All intercession methods take as parameter one or several AST nodes. However, for adding code only, for sake of pragmatism, Spoon enables one to manipulate code strings encapsulated in a special object: a "code snippet". In order for code snippets to seamlessly integrate with the existing AST-level intercession method, in the metamodel, a code snippet is a subclass of an AST node.

Let us now discuss the concrete example of Listing 3. It shows the code of a processor that adds logging in order to detect null method parameters. The processor processes `CtParameter` (the metamodel type representing method parameters). It creates a snippet representing an "if/then" statement that checks the value of the parameter. It then adds this piece of code at the beginning of the method body corresponding to this parameter. Note that when several parameters are declared, all are processed and the checks are inserted in the same body in the order they are declared in the method signature (one can also write a sophisticated transformation to change this order). This is the only code to write. The command-line based launcher takes as input the processor name (in this case `NullLoggerProcessor`) as well as a folder containing the code to be transformed and then applies the transformation.

Scope	Name	Description
generic	replace(element)	replaces an element by another one.
generic	insertBefore(element)	inserts the current element ("this") before another element in a code block.
generic	insertAfter(element)	inserts the current element ("this") after another element in a code block.
block	insertBegin(element)	adds an element at the begin of a code block.
block	insertEnd(element)	appends an element at the end of a code block.
throw	setThrownExpression(expr)	sets the expression returning a throwable object.
assignment	setAssignment(expression)	sets the expression to be assigned in a variable.
if	setCondition(expression)	sets the conditional expression of an if.
if	setThenStatement(stmt)	sets the "then" statement of an if/then/else
if	setElseStatement(stmt)	sets the "else" statement of an if/then/else.

Table 2: Excerpt of AST Intercession Methods of SPOON.

SPOON *provides the developer with first-class intuitive intercession methods.*

3.2 Use of Generic Typing for Static Checking of Transformations

When manipulating an AST with intercession methods, well-formedness rules must be enforced so as to produce compilable code. For instance, a `throw` statement contains an expression that necessarily returns an exception object (in Java, an instance of `Throwable`). There are three ways of detecting violations of those well-formedness rules: statically when writing the code manipulation code, dynamically when applying the transformation, or by verifying that the generated code actually compiles. We believe that static checking is the best solution, since it gives instant feedback to the developer.

3.2.1 Static Checks of Intercession Methods.

In SPOON, we use Java generics to statically enforce the AST well-formedness rules. For instance, Listing 4 shows how we statically enforce that the expression of a `throw` statement is a valid exception object: the parameter of method `setThrownExpression` is typed by `CtExpression<? extends Throwable>`. More generally, as shown in Figure 3, an expression (`CtExpression`) has a type parameter `T`. This enables to statically enforce well-formedness rules at many places: the condition expression of an if statement or a loop must return a boolean;

```
/** Logs "null" passed as parameter */
public class NullLoggerProcessor extends
    AbstractProcessor<CtParameter<?>> {

  @Override
  public void process(CtParameter<?> element) {
    // we declare a new snippet of code to be inserted
     CtCodeSnippetStatement snippet = createCodeSnippetStatement();

    // this snippet contains an if check
    snippet.setValue("if("+ element.getSimpleName() +"␣==␣null)␣"
      + "Logger.log(\"null␣passed␣in␣"+ element.getSimpleName() +"\");");

    // we insert the snippet at the beginning of the method body
    element.getParent(CtMethod.class).getBody().insertBegin(snippet);
  }

}
```

Listing 3: A Code Transformation That Logs `null` Passed as Parameter. A Code Snippet is Built Iteratively and Eventually Injected in the Method Body.

```
public interface CtThrow extends CtCFlowBreak {
  // intercession method with static checking
  // of wellformedness rules
  void setThrownExpression(CtExpression<? extends Throwable> thrownExpr);
}
```

Listing 4: Using Generic Typing to Statically Enforce Correct AST Construction.

in an assignment, the generic type of the expression to be assigned must be compatible with the type of type variable, etc. The same technique is applied by the class `Class` of Java for reflection.

SPOON*'s use of Java Generics enables one to statically check the AST intercession code.*

3.2.2 Statically Verifiable Domain-Specific Transformations.

As shown previously, many intercession methods use generic typing to enforce well-formedness rules. The very same generic typing can be used to statically verify the composition of AST nodes referring to domain classes. For instance, let us assume that one manipulates classes in the `Book` domain (with domain classes such as `Book` and `Author`). Generic typing enables one to declare a variable as referring to an author instance, and to enforce that the expression initializing that variable indeed returns an author instance. This is illustrated in Listing 5, where the last line triggers a compilation error because one tries to set an expression that returns a book in a variable that expects an author.

3.3 Statically Type-Checked Code Templating for Java

We have seen so far two ways of writing code transformations. First, one can use the intercession API to manipulate AST objects. Second, one can generate code snippets using text fragments. Both have the same limitation: there is no way to be sure, before the actual transformation, that the transformed code will be compilable.

```
// declaring one variable representing a local variable typed by Author     1
CtLocalVariable<Author> varDecl = createLocalVariableDecl();               2
// declaring one expression returning an Author object                     3
CtExpression<Author> expression1 = createExpression();                     4
//declaring one expression returning a Book object                         5
CtExpression<Book> expression2 = createExpression();                       6
                                                                           7
// calling intercession method "setDefaultExpression"                      8
varDecl.setDefaultExpression(expression1);// compile                       9
varDecl.setDefaultExpression(expression2);// DOES NOT COMPILE              10
```

Listing 5: Using Generic Typing to Statically Enforce Correct AST Construction with respect to Domain Classes

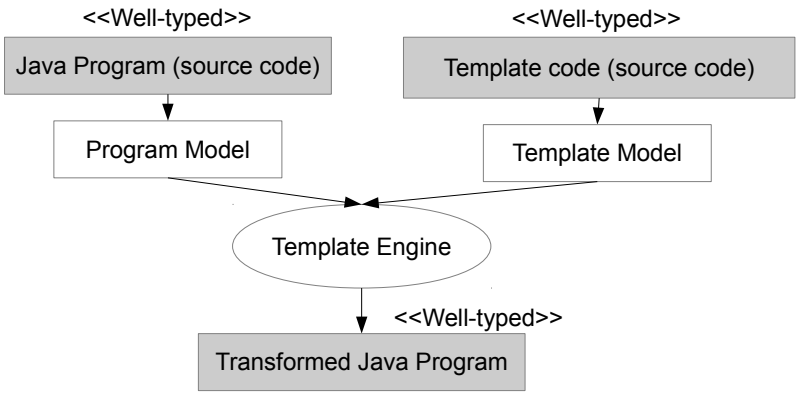

Figure 5: Overview of Spoon's Templating System.

Spoon provides developers with a third way of writing code transformations: code templates. Those templates are statically type-checked, in order to ensure statically that the generated code will be correct. Our key idea behind Spoon templates is that they are regular Java code. Hence, the type-checking is that of the Java compiler itself.

A Spoon template is a Java class that is type-checked by the Java compiler, then taken as input by the Spoon templating engine to perform a transformation. This is summarized in Figure 5. A Spoon template can be seen as a higher-order program, which takes program elements as arguments, and returns a transformed program. Like any function, a template can be used in different contexts and give different results, depending on its parameters.

3.3.1 Template Definition

Listing 6 defines a Spoon template. This template specifies a statement (in method `statement`) that is a precondition to check that a list is smaller than a certain size. This piece of code will be injected at the beginning of all methods dealing with size-bounded lists. This template has one single template parameter called `_col_`, typed by `TemplateParameter`. In this case, the template parameter is meant to be an expression (`CtExpression`) that returns a Collection (see constructor, line 3). All metamodel classes, incl. `CtExpression`, implement

```
public class CheckBoundTemplate extends StatementTemplate {
  TemplateParameter<Collection<?>> _col_;
  @Override
  public void statement() {
    if (_col_.S().size() > 10)
      throw new OutOfBoundException();
  }
}
```

Listing 6: An example of template in SPOON. If the template compiles, the transformed code compiles.

```
// creating a template instance
Template t = new CheckBoundTemplate();
t._col_ = createVariableAccess(method.getParameters().get(0));

// getting the final AST
CtStatement injectedCode = t.apply();

// adds the bound check at the beginning of a method
method.getBody().insertBegin(injectedCode);
```

Listing 7: Using a template to inject code at the beginning of a method.

interface `TemplateParameter`. A template parameter has a special method (named S, for **S**ubstitution) that is used as a marker to indicate the places where a template parameter substitution should occur. For a `CtExpression`, method `S()` returns the return type of the expression.

A method `S()`is never executed, its only goal is to get the template statically checked. Instead of being executed, the template source code is taken as input by the templating engine (see Figure 5) which is described above. Consequently, the template source is well-typed, compiles, but the binary code of the template is thrown away.

There are three kinds of templates: block templates, statement templates and expression templates. Their names denote the code grain they respectively address.

3.3.2 Template Instantiation

In order to be correctly substituted, the template parameters need to be bound to actual values. This is done during template instantiation.

Listing 7 shows how to use the check-bound of template of Listing 6. One first instantiates a template, then one sets the template parameters, and finally, one calls the template engine. In Listing 7, last line, the bound check is injected at the beginning of a method body.

Since the template is given the first method parameter which is in the scope of the insertion location, the generated code is guaranteed to compile. The Java compiler ensures that the template compiles with a given scope, the developer is responsible for checking that the scope where she uses template-generated code is consistent with the template scope.

```
public class TryCatchOutOfBoundTemplate                                    1
    extends BlockTemplate {                                                2
  TemplateParameter<Void> _body_; // the body to surround                  3
                                                                           4
  @Override                                                                5
  public void block() {                                                    6
    try {                                                                  7
      _body_.S();                                                          8
    } catch (OutOfBoundException e) {                                      9
      e.printStackTrace();                                                 10
       }                                                                   11
}}                                                                         12
```

Listing 8: A template to inject try/catch blocks

```
// with TemplateParameter
TemplateParameter<Integer> val;
...
val = Factory.createLiteral(5);
...
if (list.size()>val.S()) {...}
```

```
// with literal template parameter
@Parameter
int val;
...
val = 5;
...
if (list.size()>val) {...}
```

Figure 6: Two equivalent excerpts of templates. The left-hand side one use TemplateParameter, the right-hand side one is more concise thanks to @Parameter.

3.3.3 Template Substitution

The substitution engine uses the metamodel and querying API presented in Section 2.3 to look up all invocations of method `S()`. It then substitutes them by the template parameter instances. Here, if `_col_` represents the method parameter `x`, the substitution modifies the model to return the expression "`x.size()>10`" expression. If `_col_` stands for an expression that returns a collection (say `getCollection()`), the substitution result is "`getCollection ().size()>10`".

3.3.4 What can be Templated?

All metamodel elements can be templated. For instance, one can template a try/catch block as shown in Listing 8. This template type-checks, and can be used as input by the substitution engine to wrap a method body into a try/catch block. The substitution engine contains various methods that implement different substitution scenarios. For instance, method `insertAllMethods` inserts all the methods of a template in an existing class. It can be used for instance, to inject getters and setters.

3.3.5 Literal Template Parameters

We have already seen one kind of template parameter (`TemplateParameter<T>`). Sometimes, templates are parameterized literal values. This can be done with a template parameter set to a CtLiteral, for instance,

For convenience, SPOON provides developers with another kind of template parameters called *literal template parameters*. When the parameter is known to

Intercession API	Well-suited for fine-grain, surgical transformations. Cumbersome for inserting new code.
Code snippets	Well-suited for code injection. Fragile when complex conditionals are used.
Templates	Well-suited for large-scale transformations. Hard to learn and master.

Table 3: The Three Complementary Mechanisms for Code Transformation in Spoon

be a literal (primitive types, `String`, `Class` or a one-dimensional array of these types), a template parameter enables one to simplify the template code. To indicate to the substitution engine that a given field is a template parameter, it has to be annotated with a `@Parameter` annotation. Figure 6 illustrates this feature with two equivalent templates. By using a literal template parameter, it is not necessary to call the `S()` method for substitution: the templating engine looks up all usages of the field annotated with @Parameter. The listing above shows those differences.

3.3.6 Summary

To sum up, Spoon templates are regular Java code. Since they type-check successfully, they ensure that the transformed code actually compiles. It is complementary to the intercession API and the code snippets. Depending on the transformation and the amount of time given to write it, they all have pros and cons, as summed up in Table 3. The intercession API is well-suited for fine-grain, surgical transformations but cumbersome for inserting lots of new code. Code snippets are very handy for code injection, however, in complex transformations with many conditionals, they may result in incorrect code. The templates provides strong static checks with respect to the transformed code. However, the price to pay is that they have a longer learning curve required to master them. Again, they are complementary, one can use them in conjunction in the same transformation.

3.4 Annotation-Driven Program Processors

We now discuss how Spoon deals with the processing of annotations. Java annotations enable developers to embed metadata in their programs. Although by themselves annotations have no explicit semantics, they can be used by frameworks as markers for altering the behavior of the programs that they annotate. This interpretation of annotations can result, for example, on the configuration of services provided by a middleware platform or on the alteration of the program source code.

Annotation processing is the process by which a pre-processor modifies an annotated program as directed by its annotations during a pre-compilation phase. The Java compiler offers the possibility of compile-time processing of annotations via the API provided under the `javax.annotation.processing` package. Classes implementing the `javax.annotation.processing.Process` interface are used by the Java compiler to process annotations present in a client program. The client code is modeled by the classes of the `javax.lang.model`

```
@Target({ElementType.PARAMETER})                                    1
@Retention(RetentionPolicy.SOURCE)                                  2
public @interface NotNull{}                                         3
                                                                    4
class Person{                                                       5
  public void marry(@NotNull Person so){                           6
    if(!so.isMarried())                                             7
      //Marrying logic...                                           8
  }                                                                 9
}                                                                   10
```

Listing 9: Definition and Use of a Java Annotation at the Method Parameter Level.

package (although Java 8 has introduced finer-grained annotations, but not on any arbitrary code elements). It is partially modeled: only types, methods, fields and parameter declarations can carry annotations. Furthermore, the model does not allow the developer to modify the client code, it only allows adding new classes.

The Spoon annotation processor overcomes those two limitations: it can handle annotations on any arbitrary code elements (including within method bodies), and it supports the modification of the existing code.

3.4.1 Annotation Processing with Spoon.

Spoon provides developers with a way to specify the analyses and transformations associated with annotations. Annotations are metadata on code that start with `@` in Java. For example, let us consider the example of a design-by-contract annotation. The annotation `@NotNull`, when placed on arguments of a method, will ensure that the argument is not null when the method is executed. Listing 9 shows both the definition of the `@NotNull` annotation type, and an example of its use.

The `@NotNull` annotation type definition carries two meta-annotations (annotations on annotation definitions) stating which source code elements can be annotated (line 1), and that the annotation is intended for compile-time processing (line 2). The `@NotNull` annotation is used on the argument of the `marry` method of the class `Person`. Without annotation processing, if the method `marry` is invoked with a NULL reference, a `NullPointerException` would be thrown by the Java virtual machine when invoking the method `isMarried` in line 7.

The implementation of such an annotation would not be straightforward using Java's processing API since it would not allow us to just insert the NULL check in the body of the annotated method.

3.4.2 The Annotation Processor Interface.

In Spoon, the full code model (c.f. Section 2.2) can be used for compile-time annotation processing [35]. To this end, Spoon provides a special kind of processor called `AnnotationProcessor` whose interface is:

```
public interface AnnotationProcessor                                1
                  <A extends Annotation, E extends CtElement>       2
    extends Processor<E> {                                          3
  void process(A annotation, E element);                            4
```

```
class NotNullProcessor extends                                                    1
        AbstractAnnotationProcessor<NotNull,CtParameter>{                          2
                                                                                   3
  public NotNullProcessor(){                                                       4
    addConsumedAnnotationType(NotNull.class);                                      5
    addProcessedAnnotationType(NotNull.class);                                     6
  }                                                                                7
                                                                                   8
  @Override                                                                        9
  public void process(NotNull anno, CtParameter param){                            10
    CtMethod<?> method = param.getParent(CtMethod.class);                          11
    CtBlock<?> body = method.getBlock();                                           12
    CtAssert<?> assertion = constructAssertion(param.getSimpleName());             13
    body.insertBegin(assertion);                                                   14
  }                                                                                15
```

Listing 10: The Processor That Injects Assertions to Check the Validity of @NotNull Annotations.

```
  boolean inferConsumedAnnotationType();                                           5
  Set<Class<? extends A>> getProcessedAnnotationTypes();                           6
  Set<Class<? extends A>> getConsumedAnnotationTypes();                            7
}                                                                                  8
```

Annotation processors extend normal processors by stating the annotation type those elements must carry (type parameter `A`), in addition of stating the kind of source code element they process (type parameter `E`). The `process` method (line 4) receives as arguments both the CtElement and the annotation it carries. The remaining three methods (`getProcessedAnnotationTypes`, `getConsumedAnnotationTypes` and `inferConsumedAnnotationTypes`) configure the visiting of the AST during annotation processing. The SPOON annotation processing runtime is able to infer the type of annotation a processor handles from its type parameter `A`. This restricts each processor to handle a single annotation[2]. To avoid this restriction, a developer can override the `inferConsumedAnnotationType()` method to return `false`. When doing this, SPOON queries the `getProcessedAnnotationTypes()` method to find out which annotations are handled by the processor. Finally, the `getConsumedAnnotationTypes()` returns the set of processed annotations that are to be consumed by the annotation processor. Consumed annotations are not available in later processing rounds. Similar to standard processors, SPOON provides a default abstract implementation for annotation processors: `AbstractAnnotationProcessor`. It provides facilities for maintaining the list of consumed and processed annotation types, allowing the developer to concentrate on the implementation of the annotation processing logic.

Going back to our `@NotNull` example, we implement a SPOON annotation processor that processes and consumes `@NotNull` annotated method parameters, and modifies the source code of the method by inserting an `assert` statement that checks that the argument is not null.

The `NotNullProcessor` (Listing 10) leverages the default implementation provided by the `AbstractAnnotationProcessor` and binds the type variables representing the annotation to be processed and the annotated code elements to `NotNull` and `CtParameter` respectively. Then, in the class constructor, it

[2]Java does not allow annotations to extend other annotations.

configures the annotation types it is interested in by adding them to the lists provided by its super class (lines 5 and 6). The actual processing of the annotation is implemented in the `process(NotNull,CtParameter)` method (lines 10-13). Annotated code is transformed by navigating the AST up from the annotated parameter to the owner method, and then down to the method's body code block (lines 10 and 12). The construction of the `assert` statement is delegated to a helper method `constructAssertion(String)`, taking as argument the name of the parameter to check. This helper method constructs an instance of `CtAssert` (by either programmatically constructing the desired boolean expression, or employing the templating facilities explained in Section 3.3). Having obtained the desired assert statement, it is injected at the beginning of the body of the method.

More complex annotation processing scenarios can be tackled with Spoon. For example, when using the `@NotNull` annotation, the developer is still responsible for manually inspecting which method parameters to place the annotation on. A common processing pattern is then to use regular Spoon processors to *auto-annotate* the application's source code. Such a processor, in our running example, can traverse the body of a method, looking for expressions that send messages to a parameter. Each of these expressions has as hypothesis that the parameter's value is not null, and thus should result in the parameter being annotated with `@NotNull`.

With this processing pattern, the programmer can use an annotation processor in two ways: either by explicitly and manually annotating the base program, or by using a processor that analyzes and annotates the program for triggering annotation processors in an automatic and implicit way. This design decouples the program analysis from the program transformation logics, and leaves room for manual configuration.

A second complex use of annotation processing in Spoon is presented in the evaluation section (see Section 4.3.3).

4 Evaluation

We now present an evaluation of Spoon. The evaluation is composed of three parts: correctness (4.1), performance (4.2), and case studies (4.3).

4.1 Correctness

The core of an AST-based source code analysis tool as Spoon is made of two components: a model of the program and a pretty-printer. For real-world and rich programming languages, it is hard to achieve a model that provably covers all parts of the language. Similarly, it is very difficult to prove that all pretty-printed versions of all valid models would be valid programs that 1) correspond to the original one, and 2) can be compiled and executed.

To validate the correctness of those two components, we set up the following experiments: first, we collect a dataset of open-source programs; second, for each subject programs, we build the model from the original source code and we then pretty-print back as source code; third, we compile the pretty-printed

source code; fourth, we run the test suite of the pretty-printed program to check whether the program has still the same observable semantics.

This process hence has two stacked correctness oracles: the compiler, and the test suite. If all programs can be represented as a model whose pretty-printed version can be compiled and executed, this gives strong confidence in both the correctness of the model and of the pretty-printer. The test suite ensures that the model indeed corresponds to the original program.

We searched in software forges software packages that meet the following inclusion criteria: the program is open-source; the program is written in Java; the program contains a good test suite. Also, the better the program test suite is, the better the correctness oracle is. Hence, we take a special care of selecting programs that are well-tested. Note that the test suite is not pretty-printed in order to keep the exact same specification and avoid biases.

This process results in 13 software applications: Bukkit; Commons-codec; Commons-collections; Commons-imaging; Commons-lang; DiskLruCache; Java-writer; Joda time; Jsoup; JUnit; Mimecraft; Scribe Java; Spark. An archive containing the source code of those subjects is available as supplementary material.

Table 4 gives the key descriptive statistics for this dataset: the commit id (the digest of the commit in the Git version control system, necessary for future replication), the size in number of statements (a semantic Line-of-Code measure, that we also call LOC in this paper), and the number of tests (as defined by the JUnit test framework: the number of test methods). The smallest programs have less than 1000 LOC and the biggest program has 31k LOC. The total is 166 079 statements. The number of test cases ranges from 14 to 15 067.

> For all programs of Table 4, Spoon v2 successfully creates a model of the program as instance of the Spoon metamodel, and the pretty-printed version passes the two correctness oracles (compilation and application test suite success).

In addition, over the years, Spoon has been used in many different projects, both within academia and within industry, which further validates this formal experiment.

4.2 Performance

We now discuss the performance of Spoon. How long does it take to create a model of a program and to write it back on disk? For each program of Table 4, we measure the time required to parse the original program, build the model of the program as instance of the Spoon metamodel, and pretty print it to disk. All experiments are done on a MacBook with CPU Intel Core i7 (2GHz) and 8Gb RAM (1600 MHz DDR3) and a solid state disk with an HFS file system. The version of Spoon is 2.3.1 and the version of the Java virtual machine is OpenJDK 7. The results are shown in Table 5.

For 5 out of 13 subjects, the time to build a model of a program and write its pretty-printed version is lower than 1 second. For the remainder, it takes up to 6,8 seconds (for Commons-collections) to perform the same task. We see an expected correlation between the application size and the Spoon time.

Name	Commit Id	#stmt	#test
Bukkit	f210234e59275330f83b994e199c76f6abd41ee7	24176	906
Commons-codec	fe6dbee7524a5aa4160751155be2e8c975f66c68	6858	644
Commons-collections	a3503a1b569977dc6f8f136dcdcdabe49b8249c1	24125	15067
Commons-imaging	92440e4206a12ffd455def326c181058b53b6681	27875	94
Commons-lang	3482322192307199d4db1c66e921b7beb6e6dcb6	25551	2581
DiskLruCache	c2965c04a03d016c58323c41223d8e2808adf732	845	61
Javawriter	d39761f9ec25ca5bf3b7bf15d34fa2b831fed9c1	766	60
Joda time	8ddb3fb57078d57cfea00b1f588c9ffa8ffbec03	31552	4133
Jsoup	ab7feede4eedee351f320896114394235c2395ef	11746	430
JUnit	077f5c502a2494c3625941724ddcd5412a99ccc8	8509	867
Mimecraft	e7b3f9a764fc34cd473e6e2072cb5a995bec0bb2	331	14
Scribe Java	e47e494ce39f9b180352fc9b5fd73c8b3ccce7f8	1938	99
Spark	a19594a0e9d824c61996897cf8d1efcd9da43313	1807	54

Table 4: Descriptive statistics of our dataset of 13 Java applications

Name	Spoon Time	Compilation Time
Bukkit	5.2s	15.5s
Commons-codec	2.7s	31.1s
Commons-collections	6.8s	47.2s
Commons-imaging	4.6s	48.2s
Commons-lang	5.1s	55.0s
DiskLruCache	8.7s	10.3s
Javawriter	0.8s	9.7s
Joda time	5s	34.4s
Jsoup	1s	20.0s
JUnit	2.5s	31.0s
Mimecraft	0.7	15.0s
Scribe Java	0.5	28.7s
Spark	1s	13.1s

Table 5: The time in seconds to build a Spoon model for each program of the dataset. As comparison, the time to compile the program is given.

The correlation is not strict because not all applications use exactly the same features of the Java language, and thus do not exercise the exact same parts of Spoon.

If we compare those durations with the time required to compile the whole application (using javac), it is clearly less (between one third and one tenth). Consequently, this experiment shows that the compilation is a good upper bound of the price of source code analysis. Note that the Spoon time does not take into account the domain-specific analyses and transformations, it is only the basic cost to pay. According to the many different transformations we have done over the years, the transformation time is negligible compared to the model building and compilation time.

`List a;`	`[a] <= List`
`List b = new LinkedList();`	`[b] <= [new LinkedList()]` `E(b) = E(new LinkedList())`
`a=new ArrayList();`	`[a] <= [new ArrayList()]`
`a.add("shu");`	`["shu"] <= String` `E(a) = ["shu"]`
`b = a;`	`[b] <= [a]` `E(b) = E(a)`

Table 6: An Example of Extracted Constraints for Inferring Missing Java Generics (`[a]` stands for "the type of expression `a`", and `E(b)` stands for "the type of the `E` type variable in expression `b`"). The solution of the constraint equation is that both `a` and `b` are typed with "List<String>"

4.3 Case Studies

We now present four case studies to give concrete insights on how SPOON is relevant to analyze and transform source code.

4.3.1 Java 1.4 to Java 5 Translator

In the early years of Java 5, there was an important need for porting Java 1.4 code. In particular, the Java 5 compiler generates a great deal of type-safety warnings (in particular when using the collection framework). In order to avoid these warnings, one needs to include type parameters into the legacy code, which is a time-consuming, error-prone, and repetitive task.

Automated refactoring approaches are implemented in some IDEs (e.g. Eclipse, Idea). As an illustration of SPOON's power, let us now present how to implement this refactoring with SPOON. The refactoring handles two cases: the automatic addition of generic types, and the morphism of for loops into for-each loops.

Generics To introduce generic type parameters, one needs to statically determine the missing types. This analysis consists of checking variable usages for inferring their type parameter(s). To do so, we use the algorithm described by Fuhrer et al. [15], which uses a constraint-based approach.

Fuhrer's type-inference algorithm is composed of two main stages. In the first stage, a pass over the statements of the program is made. During this pass, type constraints on the expressions are derived and added to an equation system. In the second stage, once all constraints are derived, the equation system is solved, and a single type solution is associated with each expression. For example, in Table 6, a fragment of source code and the relevant derived constraints are shown. After having found a set of types that satisfies all the derived constraints, the program is then transformed into a generic-compliant Java 5 program.

Implementation. The refactoring is implemented by two sets of SPOON processors. The first set derives the type-constraints, while the second one eliminates cast operations made redundant by the inclusion of type parameters. The constraint system solution is found by means of a book-keeping algorithm as described in [15].

```
class LoopTemplate extends BlockTemplate {                                  1
  TemplateParameter<Collection<_I_>> _collection_;                          2
  TemplateParameter<Void> _body_;                                           3
  TemplateParameter<Iterable> _loopingVariable_;                            4
  @Parameter Class _I_;                                                     5
  @Override                                                                 6
  public void block() throws Throwable {                                    7
    for(_I_ _loopingVariable_ : _collection_.S()) {                         8
      _body_.S();                                                           9
} } }                                                                       10
```

Listing 11: The Template Used to Produce Enhanced For-loops

First, constraints are derived using five analysis processors; each of them takes care of a single kind of statement: assignments, class declarations, method invocations, method returns, and field and local variable definitions. Each processor implements an inference rule that produces a given set of constraints for the statement or expression it processes.

Second, the constraint equation is solved and the solution is translated back into code using a transformation processor that manipulates the type declaration of field declarations, method parameters, and local variable declarations.

Finally, redundant casts are eliminated by a last processor which, during a second processing round of the program's model, visits all invocations and checks if they are casted. If the cast is a type that is assignable from the return type of the method, the cast is removed.

Loop Transformations Java 5 introduced a new language construct called "enhanced for loop" (a.k.a. *foreach*). This *foreach* construct is syntactic sugar that avoids the explicit use of iterators. Let us now describe a refactoring that uses SPOON to translate regular *for* iterator-based loops into their *foreach* equivalent.

To be able to translate a traditional *for* loop into a *foreach* loop, it is necessary to be able to identify the source code pattern that denotes a translatable *loop*, and replace the *for* loop with its equivalent. We have implemented a SPOON processor that performs this task for this particular kind of *for* loops.

The `ForeachProcessor` is implemented as follows. When a for loop is found (`CtFor` elements), it is tested whether it matches the desired structure (i.e. is translatable). For instance, we check that the loop body does not contain further calls to `next()`. We use the reflection API to check that the initialization, looping expression, as well as the first statement of the block are of the expected form. To replace the initial *for* loop statement, we use the template `LoopTemplate` shown in Listing 11, where `_collection_` represents the iterable collection, `_body_` represents the loop body without the first statement of the initial loop, `_I_` represents the type of the collection's contents (see Section 3.3.5 for type substitution), and `_loopingVariable_` represents the name of the identifier used to denote the currently iterated element. `ForeachProcessor` then instantiates this template by feeding the parameters with the appropriate values in order to get the piece of code used to replace the original loop.

4.3.2 Meta-Object Protocol For Method Calls

A meta-object protocol [21] enables developers to access the internals of a programming language's semantics. There are different meta-object protocols and different ways for implementing them. For instance, a meta-object protocol can modify the object instantiation or the read and write accesses of object fields. It can be implemented within the runtime environment (such as in Smalltalk) or with code transformation. In this section, we present an implementation of a meta-object protocol for method calls (called meta-call protocol or MCP) implemented as a code transformation with Spoon.

A meta-object protocol for method calls enables one to intercept all method calls of a software application [43]. The interception consists of many pieces of information starting with the receiver object, the method to be called and the parameter values. AspectJ [22] provides one kind of meta-call protocol using aspects. The interception can have several different goals: logging some information, replacing the method by a more efficient alternative, checking preconditions, changing the arguments, etc.

Our meta-call protocol intercepts the following information: the receiver, the actual parameters, the statically declared parameter types, the called method, the class containing the called method, the location of the call, the code being replaced. To illustrate this, Listing 12 shows a method call before and after transformation, where all the required information is passed to the method call protocol handler (class `MCP`). MCP takes as input one object that specifies the method call (an instance of class `MethodCall`). This object is set once by chaining setters and is immutable.

Within class `MCP`, several strategies are implemented. The default strategy simply calls the method using reflection. It is semantically equivalent to the original code. We have implemented other strategies, such as logging or advanced exception handling. Let us now concentrate on the transformation code.

Listing 13 shows a simplified view of the code used to replace all method invocations by a proxy to the meta-call protocol handler. It is a processor (see Section 2.4) that processes all AST nodes representing a method invocation (instances of `CtInvocation`). The processor incrementally builds a code snippet. For each required information, the AST is traversed in a semantic manner (e.g. `getExecutable().getDeclaration().getParent()` to know the class containing the method to be called).

Spoon proved to be appropriate for implementing a meta-method call protocol for Java. In this use case, three features can be considered as essential: first, the ability to specify the elements to be processed in a declarative manner; second, the methods to navigate the abstract syntax tree (e.g. from the invocation to the called method to its enclosing class); third, the intercession API which enables a one liner to specify the replacement.

4.3.3 Annotation Validation of Java Webservices

Since their introduction in version 5 of the Java programming language, annotations have been adopted by a number of frameworks as a means to embed metadata in a program's source code. Annotation types are typically used as an alternative to XML configuration files (e.g. J2EE [30]), or as an augmentation

```
// before code transformation
z = mt.addition(2 ,3)

// after code transformation
z = (Integer) MCP.callMethod(new MethodCall()
  // the called method
  .setCalledClass(MathTest.class)
  .setCalledMethodName("addition")
  .setParameterTypes(int.class,int.class)
  // the receiver and parameters
  .setReceiver(mt).setActualParameters(2,3)
  // the location
  .setEnclosingFileName("MainMath.java").setLineNumber(23)
      // the initial expression for sake of debug
  .setCode("mt.addition(2␣,3)")
);
```

Listing 12: The Result of the Code Transformation that Sets up a Meta-call Protocol

```
class MCPProcessor extends AbstractProcessor<CtInvocation<?>> {
  // we process all method invocations
  @Override
  void process(CtInvocation<?> astNode)
  {

    String code="new␣MethodCall()";

    // adding the called class
    CtClass calledClass
           = astNode.getExecutable().getDeclaration().getParent();
    code += ".setCalledClass("+calledClass.simpleName()+")";

    // adding the called method
    CtMethod calledMethod = astNode.getExecutable().getDeclaration();
    code += ".setCalledMethod("+calledMethod.simpleName()+")";

    // adding the actual parameters
    String passedParameters = "";
    for (int i = 0; i < astNode.getArguments().size(); i++) {
      passedParameters += astNode.getArguments().get(i)+",";
      ...
    }
    code += ".setActualParameters("+passedParameters+")";

    ...

    // replacing the call AST note by the new
    finalCode = "MCP.callMethod("+code+")"
    astNode.replace(getFactory().CodeSnippet().setValue(finalCode));
  }
}
```

Listing 13: The Simplified Code to Replace all Method Invocations by the Meta Method-call Protocol Handler

of the semantics of the base language (e.g. AspectJ5 [1]).

Users of these frameworks annotate their code in order to access services or declare special semantics of their program. The manner in which annotations are placed on program elements must respect the rules specified by the framework being used, and violations of these rules result in compilation or runtime errors.

In this section, we describe a SPOON-based framework for the specification and checking of the usage rules of annotations. Our annotation validation framework is called *AVal*. The key idea is that when annotations are defined, they also specify the way in which they should be validated. The validation rules are specified by *meta-annotations* – i.e., annotations on the annotations.

The meta-annotations declare constraints with respect to the program element that it annotates (its *target*) as well as with respect to other annotations in the system (interplay between annotations).

Constraints in AVal belong to one of these four kinds: local, scoped, attribute or target. Local and scoped constraints deal with the presence of annotations on the AST with respect to each other. Attribute constraints define valid values for the attributes of annotations used in the system, while target constraints deal with the characteristics of the AST nodes that can carry each annotation. Examples of AVal annotation for each kind follow:

Local Constraints restrict the annotations allowed on a particular AST node. For example, `@Requires` takes as parameter another annotation and states that the constrained annotation is only valid if the target AST node already carries the parameter given to the `@Requires` annotation.

Scoped Constraints dictate the annotations allowed or not on sibling nodes on the AST. For example, to constrain the presence of an `@Id` annotation to only one of the fields of a class, the `@Id` annotation definition must be constrained with `@UniqueInside(CtClass.class)`

Attribute Constraints restrict the values allowed on the parameters of annotations. When placed on an attribute defined on an annotation type, the `@URLValue` annotation checks that the values for the attribute are valid URL strings.

Target constraints describe the characteristics of the AST nodes allowed to carry an annotation. For example, the annotation `@Type(Customer.class)` would restrict an annotation to be placed on classes or fields that extend the Customer type.

An in-depth description of the semantics of these constraints can be found in [32].

AVal has a four layer architecture: At the bottom lies the client code that must be validated. The client code carries domain annotations defined by the domain annotation library. The annotation types defined in the library are themselves annotated by constraining annotations defined by AVal (e.g., `@Requires`). On the top layer, the validation of each constraint is implemented by a SPOON annotation processor.

Annotation Validation with AVal – Web-Services with JSR 181 We will now use the three annotations defined in the JSR181 for web-services (`@WebService`, `@WebMethod` and `@OneWay`) to show how, using AVal, their use can be validated by special SPOON processors.

```
@AValTarget(CtClass.class)                                    1
@Modifier(                                                    2
  "!final␣&␣!abstract␣&␣public")                              3
@HasDefaultConstructor                                        4
public @interface WebService {                                5
  ...                                                         6
}                                                             7
```

Listing 14: Restricting the @WebService annotation to public classes which are not abstract or final

```
@AValTarget{CtMethod.class}                                   1
@Inside(WebService.class)                                     2
@Modifier("public")                                           3
public @interface WebMethod {                                 4
 ...                                                          5
};                                                            6
```

Listing 15: @WebMethod annotations are only allowed on public methods that are defined in a class annotated with @WebService

The JSR181 [47] is a specification for the description of web services using pure Java objects. The JSR defines a set of annotations and their mapping to the XML-Based Web Service Description Language. In Section 2.5.1 of the specification, it is stated that implementations of the JSR must provide a validation mechanism that performs the semantic checks on the Java Bean web service definition. Below we describe three of those annotations, and how AVal is used to provide the semantic checks required by the specification.

`@WebService` marks a class as representing a web service. It specifies the following constraints: the class must be public and must not be `final` nor `abstract`, it must also define a default public constructor, the wsdlLocation parameter must be a valid URL pointing to the definition of the WSDL file for this web-service. The meta-annotated code for the `@WebService` annotation is show in Listing 14. The `@AValTarget` annotation is used to constrain `@WebService` to classes. `@Modifier` checks that the class carrying the annotation is public and neither final nor abstract.

`@WebMethod` marks a method as being a web operation for the web service. Web methods can only be declared on web service classes. This is expressed by the AVal constraint `@Inside` on line 2 of Listing 15. Also, the method must be `public` (line 3).

Asynchronous web methods are declared with the `@OneWay` annotation. This means that it is an error to annotate a method as being `@OneWay` without it being a `@WebMethod` (line 1, Listing 16). Also, being a one way method, no return value is allowed. This is specified on line 2 by the `@Type` constraining the return type to `void`.

Domain Specific Meta-annotations When AVal annotations are not enough, the Spoon API enables developers to write domain-specific AVal meta-annotations and their corresponding custom checkers. In the presentation of `WebService`, a domain-specific meta-annotation `@HasDefaultConstructor` is used. It checks for the presence of a no arguments, public constructor in the annotated class. This is achieved by annotating the annotation type definition for the `@HasDe-`

```
@Requires(WebMethod.class)                                    1
@Type(void.class)                                             2
public @interface OneWay {                                    3
};                                                            4
```

Listing 16: Methods annotated with OneWay must already be annotated with @WebMethod and must not return a value

```
@Implementation(DefaultConstructorValidator.class)            1
public @Interface HasDefaultConstructor{}                     2
```

Listing 17: Annotation Type for custom AVal annotation

`faultConstructor` with an AVal-defined annotation `@Implementation`, which makes the link between the AVal annotation and the SPOON processor that validates the rule it defines.

The source code for the SPOON processor that checks this constraint is shown in Listing 18. This processor validates annotations that carry the `@HasDefaultConstructor` meta-annotation, as specified by its implementation of the `Validator <HasDefaultConstructor>` annotation. For each occurrence of such an annotation, the `check` method is called with as parameter an object that contains contextual information (such as the program element AST node on which the annotation was found). The validator reports violations on the *has default constructor* constraint (see line 16 in Listing 18).

We have used AVal as a means to describe and enforce the usage rules of annotations defined by three large frameworks [32]: Hibernate's Java Persistence API implementation [30], Fractal's component model implementation for Java Fraclet [37] and the JSR181 for Web Services. We were able to use AVal to specify 23 annotations from these frameworks.

Implementing AVal on top of SPOON provides benefits at two levels: On the implementation side, SPOON's abstractions allow AVal to rely on meta-annotations to specify annotation constraints in a modular manner.

It is to be noted that SPOON is built on top of the existing Java annotation

```
public class DefaultConstructorValidator implements                    1
     Processor<CtElement> {                                            2
                                                                       3
  public void check(ValidationPoint<HasDefaultConstructor> vp){        4
    if(vp.getProgramElement() instanceof CtClass){                     5
      CtClass<?> clazz = (CtClass<?>) vp.getProgramElement();          6
      boolean foundDefCons = false;                                    7
      for(ctConstructor cons : clazz.getConstructors()){               8
        boolean isPublic = cons.hasModifier(ModifierKind.PUBLIC);      9
        boolean noArgs = cons.getParameters().size == 0;               10
        if(isPublic && noArgs){                                        11
          foundDefCons = true;                                         12
          break;}}                                                     13
      if(!foundDefCons)                                                14
        ValidationPoint.report(Severity.ERROR, clazz,                  15
          "No default, public, no arguments constructor");             16
      }}}                                                              17
```

Listing 18: Implementation of a custom annotation checker

```
class Bird {}                                                    1
class Flying { void fly(){} }                                    2
class Swimming { void swim(){} }                                 3
                                                                 4
@Mixin({Flying.class, Swimming.class})                           5
class Duck extends Bird {}                                       6
```

Listing 19: Annotation-based Mixins in Java

system, but the processing of annotations is completely different from the default one. This enables annotations to be put on any AST elements and subsequently processed. The default annotation processing mechanism is only limited to specific elements (e.g. classes and fields).

4.3.4 Mixin Support

Over the years, a number of extensions have been proposed for the Java language, for instance to provide new features not supported by the language, or to overcome some of its limitations. One of the perhaps most mentioned features is the limited reuse power related to the absence of multiple inheritance. Several mechanisms have been proposed to circumvent the issue. Among them, the notion of "mixins", as proposed in Bracha and Cook [7], and implemented for example in JAM [3], is one of the most popular ones and has some similarities with the trait mechanism [38] proposed by languages such as Scala.

Mixin is a specialization mechanism where the fields and methods from some classes are composed and mixed into a target class. For example, consider the code sample provided in Listing 19 that defines the `Bird` class, a class for `Flying` ones, and a class for `Swimming` ones. Obviously ducks can both fly and swim. However, single inheritance cannot appropriately capture on its own such a specialization scheme since it would be cumbersome to have a single line of specialization where `Flying` and `Swimming` would be artificially related to each other. Mixins can help to deal with such an issue. We illustrate in the remainder of this section how a simple annotation-based language for Java mixins can be defined with Spoon. We are aware that defining a full-fledged mixin language would require many more features than the ones we introduce. This is why, for the sake of clarity, we limit ourselves to the very first steps of such a project. Our purpose is to illustrate the capabilities of Spoon for defining simple annotation-based domain specific languages.

Listing 19 introduces the annotation-based concept of mixin to define the `Duck` class (line 5). `Duck` extends `Bird` and results from the composition of the `Flying` and `Swimming` classes.

The `@Mixin` annotation specifies the list of classes to be composed with the target annotated class. In simple cases, when the methods and fields defined in mixed-in classes are different and interact, neither with each other, nor with those of the target class, the composition mechanism is simple and additive: fields and methods from the mixed-in classes are added to the target class.

Yet, in more complex cases, it may happen that some methods in some mixed-in classes override those of the target class or of other mixed-in classes. To illustrate this, Listing 20 revisits the previous example and introduces the `print` method.

```
class Bird { void print(){ System.out.println("Bird"); } }                1
abstract class Flying {                                                   2
  void fly(){}                                                            3
  void print() { System.out.println("Flying"); _super_print(); }          4
  abstract void _super_print();                                           5
}                                                                         6
abstract class Swimming {                                                 7
  void swim(){}                                                           8
  void print() { System.out.println("Swimming"); _super_print(); }        9
  abstract void _super_print();                                           10
}                                                                         11
                                                                          12
@Mixin({Swimming.class, Flying.class})                                    13
class Duck extends Bird {                                                 14
  void print(){                                                           15
    System.out.println("Duck");                                           16
    super.print();                                                        17
} }                                                                       18
```

Listing 20: Mixin Classes With Method Overriding

The `print` method in `Duck` invokes the inherited `print` with the usual Java syntax: `super.print`. In the mixed-in classes, we want to express the same idea: once the method has displayed its message, we want that the execution flows to the next inherited level. We cannot use the `super` keyword in such a case: Flying and Swimming do not inherit from a class. To deal with these cases, we use a naming convention: the `_super_` prefix is used to invoke an overridden method in the mixin hierarchy (see Lines 4 and 9 of Listing 20). To keep the classes error-free from the point of view of the Java compiler, we define the `_super_print` method to be abstract, and consequently the `Flying` and `Swimming` classes too.

The actual mixin is achieved by merging the mixed-in classes and the inherited ones. In the example, the specialization hierarchy will be, starting from the most specialized class: `Duck > Swimming > Flying > Bird`.

Two processors are put at work to perform the code rewriting that is needed to support this mixin mechanism. A first processor (`ClassMergingProcessor`), for `Mixin` annotated classes, merges classes by (1) copying methods and fields, and (2) appending the `$$`*`index`* suffix to the inserted methods that override existing ones. A second processor (`InvocationInliningProcessor`), for method invocations, redirects invocations from the method of the original classes to the merged methods.

As an example, the result of the rewriting performed by these two processors for the `Duck` class is shown in Listing 21. We have used this Spoon-based mixin generator with a library of 65 mixin classes for implementing the core component framework of the FraSCAti middleware platform [40].

4.4 Discussion

To sum up, we have shown that Spoon can be used in a number of different analysis and transformation scenarios. As the examples show, the Spoon API enables the developer to write a transformation that is intuitive to write and easy to maintain.

Spoon is deeply founded on AST analysis and consequently, all comments and layout are discarded when transforming code. In the case of large scale

```
class Duck extends Bird {                                                    1
  void print(){                                                              2
    System.out.println("Bird");                                              3
    print$$0();                                                              4
  }                                                                          5
  void swim(){}                                                              6
  void print$$0() { System.out.println("Swimming"); print$$1(); }            7
  void fly(){}                                                               8
  void print$$1() { System.out.println("Flying"); print$$2(); }              9
  void print$$2() { super.print(); }                                          10
}                                                                            11
```

Listing 21: Generated Code With Mixins

refactorings or mass maintenance scenarios, this is an important limitation.

In this paper, we have not discussed the usage of Spoon for standard generic static analyses such as building a control-flow graph, a program dependence graph, a single static assignment form, etc. This is on purpose: as the title suggests Spoon is meant to be used "by the masses" for writing their own domain-specific analysis (such as the one related to web services). However, based on our experience, Spoon can also be used for such tasks, for instance to perform a dead code elimination in a Java project.

5 Related Work

For early papers on source code analysis and transformation, we refer the reader to the classical surveys of Partsch et al. [34] and Feather [13]. With respect to analysis only (and not transformation), there is the survey by Binkely [6]. An analysis on the different characteristics of program transformation systems has been done by Nadera and colleagues [31]. We now present the most related pieces of research. Note that in the literature, there are different terms related with transformation: instrumentation to refer to adding monitoring points; meta-programming to refer to create pieces of programs for more abstract specifications; reflection and intercession to refer to changing a program's behavior at runtime. They all somehow refer to the same concept and are discussed below.

Ichisugy and Roudier [19] devised a preprocessor for Java. A preprocessor enables one to only write transformations specified with intrusive specific preprocessor directives. On the contrary, Spoon allows one to transform any Java program in a non-intrusive manner.

Chiba introduced the notion of compile-time meta-architecture [9]. Spoon has the same kind of architecture for the Java language. Also, it uses Java generics for well-typing program analyses and transformations. Tatsubori et al. described a macro system for Java called OpenJava [44]. OpenJava contains introspection and intercession methods that are similar to Spoon. The Javadoc system [25] enables to write "doclets", pieces of code that analyze the structure of a Java programs (classes, field, methods). Beyond the structural part, Spoon enables to analyze every single program element. Spoon allows full intercession up to the statements and expressions of the language.

Compile-time meta-architecture and aspects à la AspectJ [22] are closely related concepts. Spoon can be seen a a compile-time aspect weaver, where the AST node type and `isToBeProcessed` method act as the pointcut and

the `process` method as the advice. The main difference between AspectJ and Spoon is that Spoon enables any kind of pointcut and advice to be written: any arbitrary code patterns can be encoded in `isToBeProcessed` as well as any transformation up to code elements. AspectJ is simpler to use but to the price of having a more limited scope of aspects.

Van Deursen and Visser [45] introduced the idea of combining AST visitors to perform a wide range of analysis tasks. The main difference with Spoon is that Spoon provides not only visiting mechanisms but also a powerful transformation API. However, Van Deursen and Visser's visitor combinators might be plug on top of Spoon transformations to create higher-order transformations.

There are many bytecode analysis and transformation libraries (e.g. [10, 8]). Spoon does not work on bytecode but on source code. This enables developers to reason and transform language structures and constructs that disappear during the compilation process (e.g. try/catch blocks, anonymous classes, etc.). Also, the transformations are more intuitive to write since they refer to the syntactic programming concepts.

Klint et al. presented Rascal [23]. Rascal is a domain-specific language for source code analysis and manipulation. It enables one to implement refactoring algorithms that are independent of the one particular programming language. While a domain-specific language allows conciseness and expressiveness, it has a steep learning curve. On the contrary, any good Java developer is able to write a Spoon transformation of Java programs within a relatively short time.

Visser invented Stratego, a language for program transformations using rewriting rules [46]. Schordan and Quinlan described a source-to-source transformation framework dedicated to optimization [39]. MetaJ is a metaprogramming environment for Java [12]. It provides a template mechanism that is similar to ours described in Section 3.3. Cordy presented the TXL source transformation language. The TXL language is based on rules to specify the transformations. Balland et al. summarized their research efforts on Tom, a term-rewriting platform that is able to deal with Java code [4]. For all those approaches, the analysis and transformation developers are required to master a new abstract formalism. On the contrary, using Spoon, only the concept of abstract syntax tree is required to start writing productive transformations in plain Java.

Ludwig and Heuzeroth designed the COMPOST system [29] and some program transformations. Our goal and architecture are similar, but the Spoon mechanisms for transforming programs are more powerful. For instance, Compost has no way for specifying templates.

Strein et al. [42] proposed an extensible metamodel for program analysis. They only focus on program analysis while Spoon enables users to specify analyses and transformations using the same framework. Kuipers and Visser [26] presented a meta-approach to analysis: given a grammar definition, JJForester generates the classes implementing the meta-model and the traversal helper classes. Contrary to Spoon, this is one meta-level further.

Cetus [27] is a compiler extension that provides an intermediate representation and an API to manipulate C/C++ programs. In Spoon, what corresponds to Cetus' intermediate representation is the Spoon metamodel. While their approach is for low-level languages with a focus on automatic parallelization, ours is in Java and aims at being generic, as shown by our different case studies.

6 Conclusion

We have presented Spoon, a library for analyzing and transforming Java source code. Spoon provides a unique blend of features: a Java metamodel dedicated to analysis and transformation; a powerful intercession API; static checking of transformations using templates; fine-grained annotation processing.

Spoon has been developed since 2006 and has been used in a number of research and industrial projects. We have shown with four case studies how it is a key component in different scenarios. Future work will consist in porting the concepts of Spoon for analyzing and transforming newly popular programming languages, in particular JavaScript. Also, our current research heavily uses Spoon to insert runtime hooks, used for runtime verification and runtime failure recovery.

Acknowledgments

We would like thank all contributors of Spoon over the years: Olivier Barais; David Bernard; Benoit Cornu; Didier Donsez; Favio DeMarco; Christophe Dufour; Sebastian Lamelas Marcote; Matias Martinez; Nicolas Pessemier; Phillip Schichtel; Stefan Wolf. Gerard Paligot did the heavy lifting of Section 4.1 and 4.2.

References

[1] The aspectj 5 development kit developer's notebook. http://eclipse.org/aspectj/doc/released/adk15notebook, 2004.

[2] T. L. Alves, J. Hage, and P. Rademaker. A comparative study of code query technologies. In *2011 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 145–154, 2011.

[3] D. Ancona, G. Lagorio, and E. Zucca. A smooth extension of Java with mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00)*, pages 154–178, Sophia Antipolis and Cannes, France, June 2000.

[4] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In *Term Rewriting and Applications*, pages 36–47. Springer, 2007.

[5] D. R. Barstow. Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, (11):1321–1336, 1985.

[6] D. Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, pages 104–119. IEEE Computer Society, 2007.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (ECOOP/OOPSLA'90)*, volume 25 of *SIGPLAN Notices*, pages 303–311. ACM Press, Oct. 1990.

[8] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[9] S. Chiba. A metaobject protocol for c++. In *ACM Sigplan Notices*, volume 30, pages 285–299. ACM, 1995.

[10] S. Chiba. Javassist—a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, page 174, 1998.

[11] C. Dahn and S. Mancoridis. Using program transformation to secure c programs against buffer overflows. In *Proceedings of WCRE*, volume 3, page 323, 2003.

[12] A. A. De Oliveira, T. H. Braga, M. de Almeida Maia, and R. da Silva Bigonha. Metaj: An extensible environment for metaprogramming in Java. *J. UCS*, 10(7):872–891, 2004.

[13] M. S. Feather. A survey and classification of some program transformation approaches and techniques. In *The Working Conference on Program Specification and Transformation*, pages 165–195, 1987.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM Sigplan Notices*, volume 37, pages 234–245. ACM, 2002.

[15] R. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 71–96, 2005.

[16] Y. Guo, C. Seaman, N. Zazworka, and F. Shull. Domain-specific tailoring of code smells: An empirical study. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (Workshops)*, pages 167–170. ACM, 2010.

[17] T. J. Harmer and F. G. Wilkie. An extensible metrics extraction environment for object-oriented programming languages. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, pages 26–35. IEEE, 2002.

[18] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12), 2004.

[19] Y. Ichisugi and Y. Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. In *Scientific Computing in Object-Oriented Parallel Environments*, pages 153–160. Springer, 1997.

[20] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[21] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, July 1991.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[23] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[24] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1):9–51, 2004.

[25] D. Kramer. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153. ACM, 1999.

[26] T. Kuipers and J. Visser. Object-oriented tree traversal with jjforester. *Science of Computer Programming*, 47(1):59–87, 2003.

[27] S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2004.

[28] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM (JACM)*, 24(1):121–145, 1977.

[29] A. Ludwig and D. Heuzeroth. Metaprogramming in the large. In *Generative and Component-Based Software Engineering*, pages 179–188. Springer, 2001.

[30] L. D. Michel and M. Keith. *Enterprise JavaBeans, Version 3.0.* Sun Microsystems, May 2006. JSR-220.

[31] B. S. Nadera, D. Chitraprasad, and V. S. S. Chandra. The varying faces of a program transformation systems. *ACM Inroads*, 3(1):49–55, 2012.

[32] C. Noguera. *A Model-Driven Toolset for the Development and Validation of Annotation Frameworks.* PhD thesis, Université de Lille 1, 2008.

[33] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 9–15. IEEE, 1998.

[34] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys (CSUR)*, 15(3):199–236, 1983.

[35] R. Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11), 2006.

[36] R. Pawlak, C. Noguera, and N. Petitprez. Spoon: Program analysis and transformation in Java. Rapport de recherche RR-5901, INRIA, 2006.

[37] R. Rouvoy and P. Merle. Leveraging component-based software engineering with fraclet. *Annales des Télécommunications*, 64(1-2), Jan. 2009.

[38] N. Scharli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer-Verlag, July 2003.

[39] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proceedings of JMLC'2003*. Springer, 2003.

[40] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practise and Experience (SPE)*, 42(5):559–583, May 2012.

[41] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *ACM SIGPLAN Notices*, volume 36, pages 104–113. ACM, 2001.

[42] D. Strein, R. Lincke, J. Lundberg, and W. Lowe. An extensible meta-model for program analysis. *IEEE Transactions on Software Engineering*, 33(9):592–607, 2007.

[43] É. Tanter, N. M. Bouraqadi-Saâdani, and J. Noyé. Reflex—towards an open reflective extension of Java. In *MetaLevel Architectures and Separation of Crosscutting Concerns*, pages 25–43. Springer, 2001.

[44] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. Openjava: a class-based macro system for Java. In *Reflection and Software Engineering*, pages 117–133. Springer, 2000.

[45] A. Van Deursen and J. Visser. Source model analysis using the JJTraveler visitor combinator framework. *Software: Practice and Experience*, 34(14):1345–1379, 2004.

[46] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *Rewriting techniques and applications*, pages 357–361. Springer, 2001.

[47] B. Zotter. *Web Services Metadata for the Java Platform, Version 2.0*. BEA Systems, June 2005. JSR-181.