Spoon: Program Analysis and Transformation in Java

Renaud Pawlak, Carlos Noguera, Nicolas Petitprez

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Spoon: Program Analysis and Transformation in Java

Renaud Pawlak
Carlos Noguera
Nicolas Petitprez

N° 5901

Mai 2006

Thème COM

ISSN 0249-6399

Spoon: Program Analysis and Transformation in Java

Renaud Pawlak[1], Carlos Noguera[2], Nicola Petitprez[3]

Thème COM – Systèmes communicants
Projet JACQUARD

Rapport de recherche n° **5 9 0 1** – Mai 2006 - 26 pages

Abstract: In this research report, we present Spoon, a framework for program transformation and static analysis in Java. More precisely, Spoon is an open and extensible Java compiler, written in pure Java by using Compile-time reflection techniques. We take advantage of the new features added by Java 5, and particularly of annotations and generics. Using annotations within the Spoon framework allows the programmer to extend the Java language without defining new syntactic elements, and in such a way that it is naturally supported by IDEs for Java 5 and greater. Generics, as a priceless complement, allow for the well-typing of Spoon programs that implement the programmers' language extensions. Enforcing typing naturally provides better IDE support (such as static checks, completion, documentation, and navigation), and also allows us to define a pure Java template mechanism, which we use as a tool to define well-typed and straightforward program transformations. In addition to its basic transformation capabilities, Spoon comes with a partial evaluation engine that is used to calculate the control flow of the program and to simplify the results of template-based transformations for correctness, optimization, and readability. In order to demonstrate the usability and usefulness of our framework, we present three applications, which have been chosen to cover most of Spoon's features: a translator from Java 1.4 programs into well-typed Java 5 programs, an efficient template-based AOP extension, and an automatic implementation and validation of the visitor pattern.

Keywords: Program Analysis and Transformation, Open Compilers, Compile-time Reflection, Annotations, Generics, Templates, Partial Evaluation, AOP.

[1] INRIA Futurs, Projet Jacquard, LIFL – renaud.pawlak@inria.fr
[2] INRIA Futurs, Projet Jacquard, LIFL – noguera@lifl.fr
[3] INRIA Futurs, Projet Jacquard, LIFL – nicolas.petitprez@lifl.fr

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des vignes
4 rue Jacques Monod - Bât G -91893 Orsay Cedex (France)
Téléphone : +33 1 72 92 59 00— Télécopie : +33 1 60 19 66 08

Spoon : Analyse et Transformation de Programmes en Java

Résumé: Dans ce rapport de recherche, nous présentons Spoon, un framework pour la transformation et l'analyse statique de programmes Java. Plus précisément, Spoon est un compilateur Java extensible et ouvert, lui-même écrit en Java, et qui utilise les techniques de réflexivité à la compilation. Nous tirons parti des nouvelles fonctionnalités introduites par Java 5, et particulièrement des *annotations* et des *generics*. L'utilisation d'annotations dans le framework Spoon permet aux programmeurs d'étendre le langage Java sans avoir à définir de nouveaux éléments syntaxiques, et de telle manière que les extensions sont naturellement supportées par les environnements de développement disponibles sur le marché. Les *generics*, quant à eux, permettent le typage des programmes qui implantent les extensions de langage. Ce typage fort nous permet un meilleur support des environnements de développement (vérifications statiques, complétion, documentation, navigation, etc.), et un support d'un mécanisme de templates en Java standard, que nous utilisons comme un outil de programmation des transformations de programme. En plus de ses capacités basiques de transformations, Spoon comprend aussi un moteur d'évaluation partielle qui est utilisé pour calculer le flot d'exécution du programme et pour simplifier le résultat des transformations exprimées à l'aide de templates, principalement à des fins d'optimisation, de validation sémantique, et de lisibilité. Pour démontrer l'utilisation de notre framework, nous présentons trois applications, qui ont été sélectionnées pour couvrir la plupart des fonctionnalités offertes par Spoon: un transformateur de programmes Java 1.4 vers des programmes Java 5 bien typés, une extension AOP efficace, et une implémentation automatisée du pattern visiteur.

Keywords: Analyse et transformation de programmes, Compilateurs Ouverts, Réflexivité à la Compilation, Annotations, Generics, Templates, Evaluation Partielle, AOP.

1 Introduction

In 2004, SUN released its new version of the Java platform: Java 5 (Tiger). With this new version, came many improvements that have been pushed forward by the Java Community Process and expert groups through JSRs (Java Specification Requests). Amongst these features, *generics* significantly improve the Java typing system and allow generalized type-safe programming. They make the Java 5 programs much more robust than before and allow industrials to rely on Java to produce better quality applications. With this come enhanced security and safety properties at runtime, since more checks can be statically achieved by the compiler. The second important feature of Java 5 is the *annotation* mechanism. Annotations allow the Java 5 programmers to define and attach metadata to the program elements. These metadata can be used for declarative configuration of the programs, which comes as an interesting complement to typical XML configuration, because it is more integrated into Java, type-safe, and avoids information redundancy.

In the domains of program validation, static analysis, and transformation, these two new features have numerous potential applications. In particular, they can guide program checkers and optimizers to perform transformations in a more flexible way through annotations, and in a safer way through generics.

Despite their new applications, Java 5 generics and annotations are still not quite operational yet. Although Java 5 is backward compatible from a runtime perspective, the compilation of legacy code written in older Java versions generates a great deal of type safety warnings, which is a problem for industrials who wish to use Java 5 for projects that rely on a consequent base of legacy code. For annotations, SUN has provided an API called Annotation Processing Tool [13] (APT), which can process the annotations of Java 5 programs. At the present time, JSR 269 is defining the final API for this tool, which is to be added to Java 6. However, the uses of APT and JSR 269 are limited to partial annotation-based validation and generation, because they do not allow fine-grained modification of Java programs. Some other tools can implement program transformation through the use of the `java.lang.reflect` API. This API is, however, considerably inefficient and is not well adapted for fine-grained program validation/transformation support.

As a consequence, one of the greater challenges for the next few years will be to provide an efficient and fine-grained program manipulation tool that will totally support and take advantage of generics and annotations. Spoon, which was partially presented in [11], is a significant step toward this goal. It is an open compiler that uses Compile-time (CT) reflection [4]. Thanks to generics and annotations, it is easier for the programmer to define meta-programs that validate or transform Java programs. The novel and unpublished part of our approach is well-typed CT reflection through generics, which allows us to provide a typed template mechanism in pure Java.

In Section 2, we give an overview of Spoon and its architecture. Section 3 presents Spoon's well-typed CT reflection API. Section 4 explains program processing. Section 5 presents our pure Java template mechanism and shows how to use it for the definition of well-typed program transformations. Section 6 depicts three representative applications that we have implemented as proofs of concept. Finally, in Section 7, we evaluate Spoon and compare it to related work.

2 Overview

Figure 1 gives the overall architecture of our framework. Like many open compilers, Spoon reifies the program in a meta-model, which allows for direct access and modification of its

structure at compile-time (CT). This general technique is called CT reflection [4]. In fact, the meta-model can be seen as an abstract syntax tree (AST) that filters useful information needed by the end-users, rather than keeping all the syntactic details of the program.

The heart of the program processing framework is a *visitor* pattern applied to the meta-model. Each node of the meta-model implements an *accept* method so that it can be visited by a visitor object, which can perform any kind of action, including modification. Program processing is then naturally implemented by a specific visitor, called a *processing visitor*. The role of the processing visitor is to apply user-defined *processors*, which are declared on start-up. A user-defined processor usually performs a specific action, such as a transformation on a kind of node, under a well-defined processing condition. During one or several processing rounds, the processing visitor applies the user-defined processors until no processing condition is verified anymore, so that no processors remain to be applied.

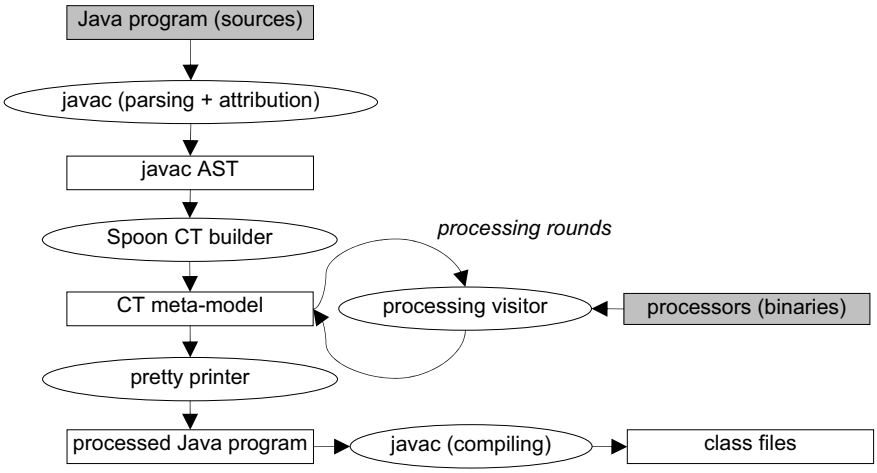

Fig. 1. Spoon architecture.

Spoon does not implement the Java parsing and bytecode generation phases. Instead, it delegates these tasks to SUN's Java compiler (javac). Therefore, to build the meta-model, it uses the javac's AST coming from the parsing and attribution phases, which implies that we reuse SUN's type-inference implementation. For the compilation phase, Spoon uses a pretty-printer visitor that generates an intermediate source-level representation of the program, which can then be compiled by javac. This technical choice has the advantage of avoiding re-implementing typical compiler jobs, and focuses on user-oriented processing. It is also easier for the end-user to have access to the processed program this way, since it is available in source code format. Spoon can also process binaries by using a decompiler (Jode [5]).

3 Well-typed Compile-time Reflection

The CT meta-model of Spoon allows the full reification and intercession (modification) of a Java program, including annotations, generics, and method bodies. It is geared toward a comprehensive, type-safe, and easy-to-use API. For instance, javac syntactic details, such as *top-level*, *import*, *apply*, and *exec* nodes have been ignored or abstracted to more intuitive notions. Spoon meta-model can be split in three parts.

- The structural part (Fig. 2), which represents the declarations of the program, such as interface, class, variable, method, annotation, and enum declarations.

- The code part (Fig. 3), which represents the executable Java code, such as the one found in method bodies and initializers.
- The reference part (Fig.4), which models the references to program elements (for instance a type reference). The referenced elements are not necessarily reified into the meta-model and may belong to third party libraries accessible from the class path, but not processed by Spoon.

3.1 Spoon's Java 5 Meta-model

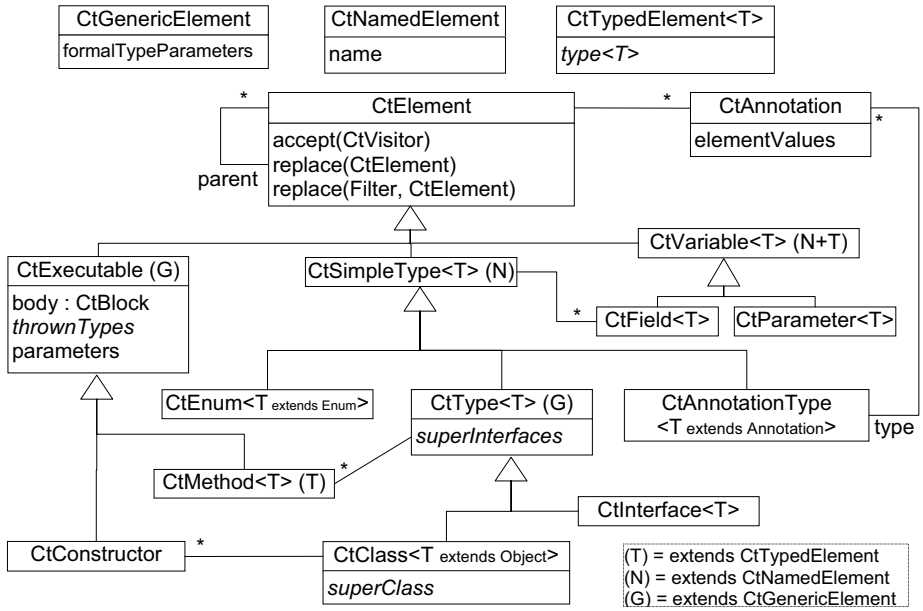

Fig. 2. Structural part of the Spoon Java 5 meta-model.

As shown in Figure 2, our Java 5 interface-based meta-model reifies the structural elements encountered in Java 5 programs. For simplification sake, we show only properties (for instance the `name` property in the `CtNamedElement`). Each property is actually implemented with getter and setter methods that follow usual naming conventions. Associations' roles are implemented in the same way and we precise their names only when different from the associated elements. Also, we do not show all the inheritance links, but we add (T), (N), or (G) when the element extends `CtTypedElement`, `CtNamedElement`, or `CtGenericElement`, which is an element that can take type parameters (a.k.a. *generics*). Finally, properties in italics are weak references that use the reference part shown in Figure 4.

The root interface is `CtElement`, which is visitable by `CtVisitor` and defines the program hierarchy through the `parent` role. Any element can be annotated, even though it is not necessarily the case in the Java syntax. Finally, typed elements are parameterized by a type parameter named `T`, which allows type-safe operations on the program's model. For the types, the `T` parameter has the same role as the type parameter in `java.lang.reflect.Class<T>`: it represents the runtime type of the CT type. For typed elements, generics ensure that this type is the same as the `type` property defined in `CtTypedElement`.

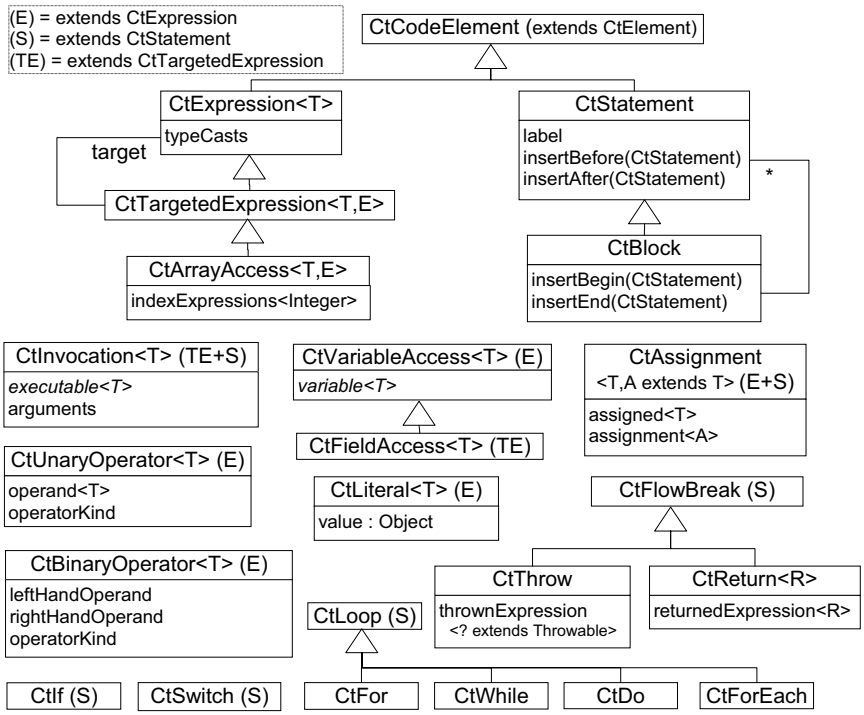

Fig. 3. Code part of the Spoon Java 5 meta-model (partial).

Figure 3 shows the meta-model for Java executable code[4]. There are two main code element kinds: the statements (`CtStatement`), which are un-typed top-level instructions that can be used directly in a block of code, and the expressions (`CtExpression`), which are typed (type parameter `T`) and used inside the statements in well-defined contexts. However, some code elements such as invocations and assignments can be used as statements or as expressions depending on the context. In the javac AST, these are enclosed in *exec* nodes when used as statements. In our meta-model, they simply inherit from both `CtStatement` and `CtExpression`.

The use of the `T` type parameter is of primary importance in our code meta-model. Thanks to it, expressions can be well-typed in several contexts. To illustrate this, we have added a type parameter to the element properties, which represents the type of the corresponding expression. For instance, in an array access, we can precise that the expressions used for the indexes are necessarily of the `Integer` type, that the expression in a `throw` statement necessarily extends `Throwable`, that the assignment expression must extend the assigned expression, etc.

The reference part of the meta-model is shown in Figure 4. References are used by the other meta-model parts 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 elements that may not exist at the time they are referenced. Besides, the referencing system holds useful information such as the actual type arguments when referencing a generic element (`CtGenericElement`), or the array types (`CtArrayTypeReference`). `CtReference` is the root class for all the references. Like the other meta-model elements, it is visitable by `CtVisi-`

[4] Because of the complexity of the Java language, the code meta-model presented here does not show all the features covered by our implementation (which is fully Java compliant). For more details on the meta-model, please refer to its implementation available at http://spoon.gforge.inria.fr/.

tor, which allows for implementing global tasks such as refactoring on the references. Also, the `getDeclaration` method of `CtReference` returns the referenced element. It is not shown in the diagram, but this method is overridden to return a more specific type when needed. For example, it returns `CtField<T>` for `CtFieldReference<T>`.

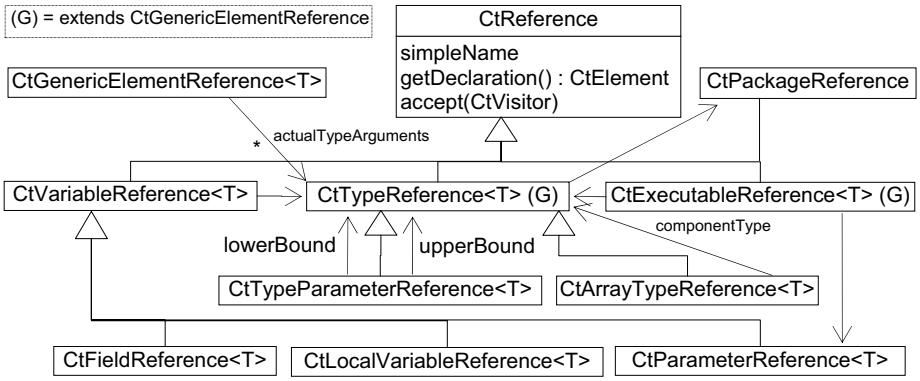

Fig. 4. Reference part of the Spoon Java 5 meta-model.

3.2 Queries

As said in the previous section, the meta-model's visitor interface is `CtVisitor`. Spoon provides an implementation called `CtScanner`, whose default behavior is to scan all the elements of the meta-model tree using a deep-search algorithm. This scanner can be sub-classed and some of its methods overloaded in order to perform specific tasks on particular elements. Based on `CtScanner`, the query API of Spoon allows the programmers to search for elements in the meta-model. To define a query, the programmer must implement the `Filter` interface defined as follows.

```
public interface Filter<T extends CtElement> {
    boolean matches(T element);
    Class<T> getType();
}
```

The `matches` method returns `true` if the currently scanned element is part of the filter. The `getType` method returns the super type of the element types that should be tested for a match. All the other element types are not tested and are not considered as matching. The query API is one of the core tools for static analysis and for extracting semantic information. For instance, the code below uses a query to select all the invocations of a given method or any method that overloads it within an `m` method.

```
CtMethod m = ...; final CtMethodReference ref = ...;
List<CtInvocation> invocations=Query.getElements(m.getBody(),
  new AbstractFilter<CtInvocation>(CtInvocation.class) {
    public void matches(CtInvocation invocation) {
      return invocation.getExecutable().isOverloading(ref);
  } });
```

3.2.1 Meta-model Modifications and Factory

To create and add new elements from scratch, it is recommended to use the factory. Spoon provides `Factory`, which contains several sub-factories for all the types of elements that need to be created. The following code shows the creation of a getter method for a field, which is of the `F` type. The use of generics ensures that the return type of the getter is the same as the field's

type[5].

```
Factory f =...; CtClass<C> c =...;
CtField<F> field = c.getField("fName");
CtMethod<F> getter = f.Method().create(
  c, f.modifiers(CtModifier.PUBLIC), field.getType(),
  "getFName", new ArrayList<CtParameter<?>>(),
  new TreeSet<CtTypeReference<?>>(), f.Code().createBlock(
  f.Code().createReturn(f.Code.createFieldAccess(
    f.Code().createFieldReference(field)))));
```

4 Program Processing

As stated in Section 3, program processing is implemented as a processing visitor, which scans the meta-model and triggers the application of user-defined processors when needed.

4.1 Processors

A user-defined processor must implement the following interface.

```
public interface Processor<E extends CtElement> {
  Factory getFactory();
  void setFactory(Factory f);
  boolean isToBeProcessed(E candidate);
  void process(E element) throws ProcessingException;
  Set<Class<? extends CtElement>> getProcessedElementTypes();
  void processingDone();
}
```

The factory is set by Spoon when constructing the declared processors. It should be retrieved by the processors with `getFactory` so that a given processing phase uses a unique factory. The `isToBeProcessed` method's implementation defines the precondition on which the `process` method is applied. The `getProcessedElementTypes` must return the element types that are to be processed by this processor. Finally, `processingDone` is up-called by the processing visitor when an entire scanning of the meta-model proceeded without impacting the meta-model. We provide an abstract implementation of this interface so that only `isToBeProcessed` and `process` have to be implemented. In this default implementation, the processed element type is automatically inferred to the `E` type parameter through runtime introspection of the processor's class.

On start-up, Spoon takes and instantiates a list of user-defined processor types. During the processing phase, on each scanned program element, Spoon tries to apply the processors in the order they have been declared. When a processor modifies the program, the meta-model is set to dirty and Spoon restarts a round of processing until no processor performs a modification. Besides, when a processor has been detected to perform no modifications during a round, Spoon up-calls the `processingDone` method. This method can be used to implement different processing strategies, such as N-phase processing. For instance, to trigger another set of rounds implying another set of processors.

4.2 Annotation Processors

A particularly useful type of processor is for processing annotations. An *annotation processor* extends a processor and is triggered when the scanned program element is annotated by one of its processed annotations. If the processed annotation is also defined as a consumed annotation, the processing visitor automatically removes the annotation when the processing is over.

```
public interface AnnotationProcessor
        <A extends Annotation,E extends CtElement>
```

[5] The `create` method's prototype is: "`<T> CtMethod<T> create(CtClass<?> owningClass, Set<CtModifiers> modifiers, CtTypeReference<T> returnType, String name, List<CtParameter<?>> parameters, Set<CtTypeReference<?>> thrownTypes, CtBlock body)`".

```
        extends Processor<E> {
  void process(A annotation, E element)
               throws ProcessingException;
  Set<Class<? extends A>> getProcessedAnnotationTypes();
  Set<Class<? extends A>> getConsumedAnnotationTypes();
}
```

In most cases, a convenient way of using Spoon is to create annotation processors that consume one annotation type at a time. To do this, `AbstractAnnotationProcessor` should be sub-classed with the consumed annotation type for the `A` type parameter. When doing so, only the `process` method actually needs to be implemented. We suggest that regular processors are only used to annotate the program, so that they can be chained with annotation processors that consume the produced annotations and perform the actual program transformations. With this design, 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 analyses 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.

4.3 Example

To illustrate the use of processors and annotation processors, we use a bounded stack example, which we will reuse throughout the report. Let us assume that we need to modify the following stack so that the number of elements it contains cannot exceed a max value.

```
public class Stack<T> {
  List<T> elements=new Vector<T>();
  public void push(T element) {
    elements.add(0,element);
  }
  public T pop() { ... }
}
```

One of our requirements is that we want to be able to plug/unplug the code that checks for the bound, so that we can easily reuse this stack in different contexts. To do so, we can define a processor that performs the appropriate code transformation.

```
public class BoundProcessor extends
       AbstractProcessor<CtClass<Stack>> {
  public Boolean isToBeProcessed(CtClass<Stack> s) {
    return s.getActualClass()==Stack.class;
  }
  public void process(CtClass<Stack> s) {
    TypeFactory f= getFactory().Type();
    // resolve the push method
    CtMethod<Void> push=s.getMethod(
      "push",f.createTypeParameterReference("T"));
    // add the thrown exception
    push.getThrownTypes().add(
      f.createReference(OutOfBoundException.class));
    // insert the test code at the beginning of push
    push.insertBegin(getTestCode());
} }
```

We will show later how to define the test code (`getTestCode` method). For the moment, let us assume that it corresponds to an `if` statement that throws an exception when the number of elements is greater than 5. When applied to our stack, the processor will transform the `push` method as follows.

```
public void push(T element) throws OutOfBoundException {
  if(elements.size() > 5) throw new OutOfBoundException();
  elements.add(0,element);
}
```

However, it would be nice to be able to be able to parameterize the `max` bound without having to modify the processor. That is where we can use annotations. With Java 5, programmers can define the following annotation.

```
@Target(ElementType.TYPE)
public @interface Bound { int max() default 10; }
```

Programmers can then annotate the stack, for instance by adding `@Bound(max=5)` before the stack declaration. This annotation is consumed by the following annotation processor parameterized with the `max` value of the `Bound` annotation.

```
public class BoundProcessor extends
       AbstractAnnotationProcessor<Bound,CtClass<Stack>> {
  public void process(Bound b,CtClass<Stack> s) {
    TypeFactory f= getFactory().Type();
    CtMethod<Void> push=s.getMethod(
      "push",f.createReference("T"));
    push.getThrownTypes().add(
      f.createReference(OutOfBoundException.class));
    push.insertBegin(getTestCode(b.max()));
} }
```

This time, the `getTestCode` method is parameterized with the max value. In the next section, we show how to define parameterized code by using the template feature of Spoon.

5 Template-Based Code-Level Transformations

As we have seen in the previous section, Spoon allows for the writing of processors that can introspect or transform the program with the use of CT reflection. Writing meta-programs on the structural part of the program (the definitions) is relatively simple. However, the task becomes more complex when dealing with the executable code (method bodies and initializers). Using CT reflection in this context leads to hardly understandable and maintainable programs. As a consequence, different techniques are needed for this kind of processing.

Here, we present a template mechanism that we have defined, which is a tool for specifying well-typed code-level transformations directly in Java, without requiring the use of CT reflection. In Section 5.1, we introduce pure Java templates; in Section 5.2, we present our parameter substitution mechanism; in Section 5.3, we explain template instantiation; and in Section 5.4, we focus on primitive template parameters. Finally, Section 5.5 presents our partial evaluation facility, and particularly discusses its use in the context of template-based transformations.

5.1 Pure Java Templates

A template is a piece of code that is not meant to be executed as is, but to be used as a base for generating an executable piece of code. As such, a template can be seen as a higher-order function (or program), which takes program elements as arguments, and returns a program. Like any function, a template can be used in different contexts and give different results, depending on its parameters. Because of those properties, templates have been used in many works for implementing program analysis and transformations, for instance in Generative Programming [3]. However, using templates often requires extra language constructs.

Thanks to generics and to the new auto-boxing/unboxing capabilities of Java, it is possible to specify code-level templates using pure Java. The simple idea behind it consists of defining a special method (named `S`) that is never executed, but that is used as a marker to indicate the places where a template parameter substitution should occur. Next, we explain our templates and their implementation in details. We also show how to use templates to implement intuitive and type-safe program transformations.

5.2 Template Parameters and Substitution

In Spoon, the following interface defines a template parameter.

```
public interface TemplateParameter<T> {
    T S();
    CtCodeElement getSubstitution(CtSimpleType targetType);
}
```

With this interface, it is possible to specify well-typed template code, by defining new template parameters (fields typed as `TemplateParameter<T>` or `TemplateParameter<T>[]`) and asking for their substitution by calling `S`. If we take again the stack example of Section 4.3, we can define the test on a max bound as follows.

```
TemplateParameter<Integer> _max_;
Collection elements; // shadow of stack elements
public void aTest() throws OutOfBoundException {
  if(elements.size()>_max_.S())
    throw new OutOfBoundException();
}
```

Since `_max_` is typed as an integer, and `S` is generically defined as returning this same type, the expression "`elements.size()>_max_.S()`" is well-typed and the compiler can check for type soundness. Note that this works because of the auto-boxing/unboxing feature, which makes the primitive types / boxing types mapping when needed. Here, the `Integer` type of `_max_.S()` is automatically translated into an `int` type, as expected by the `<` operator.

When using this template to generate new code, the programmer must use a substitution engine provided by Spoon. This engine uses the meta-model and querying API presented in Section 3.2 to look up all the `S` invocations. It then substitutes them by the result of the `getSubstitution` method, which is invoked on the template parameter instance. Here, if `_max_` represents the literal `5`, the substitution modifies the meta-model to return the "`elements.size()>5`" expression. As another example, if `_max_` stands for an expression that is an invocation of the `size` method on a list `l`, the substitution result is "`elements.size()>l.size()`".

Meta-model Element Template Parameters. Because our code meta-model (Fig. 3) is well-typed, it is very easy to use a program element as a template parameter. Indeed, `CtStatement` and `CtExpression` simply inherit from the `TemplateParameter` interface, so that any kinds of statements and expressions can be used as template parameters. In the case of an expression, the `T` type of the template parameter corresponds to the `T` type of the expression, while `T` is set to `Void` for a statement. Also, `getSubstitution` directly returns the statement or the expression.

Java-defined Template Parameters. It is not always convenient to use meta-model elements as template parameters. In particular, when defining an expression or a statement as a template parameter, creating those by using CT reflection limits the usability of templates. Hence, Spoon provides two special forms of template parameters: `BlockTemplateParameter` and `ExpressionTemplateParameter<T>`. These two respectively require the implementation of a `void block()` method and of a `T expression()` method. When substituted, a block template parameter returns the automatically constructed representation of the block method's body (a `CtBlock`), and an expression template parameter substitutes with the returned expression of the return statement. For instance, the following expression template parameter corresponds to the expression "`l.size()`".

```
class SizeCall extends ExpressionTemplateParameter<Integer> {
    List l;  // dummy field for compilation
    public Integer expression() { return l.size(); }
}
```

The next section shows how to instantiate templates for use in code-level transformations.

5.3 Template Instantiation

In order to be correctly substituted, the template parameters need to be bound to actual values. This is done during template instantiation. Each class containing template methods should define a constructor that completely binds its parameters. Besides, for clarity and typing, a template class must implement a marker interface called `Template`. Once a template has been instantiated, Spoon provides a substitution engine (`Substitution`) that implements the substitution of the parameters by their values. The most generic substitution method can transform any program element as well as all the reachable sub-elements recursively.

```
public static <C extends CtElement> C substitute(
  CtSimpleType<?> targetType, // for updating the references
  Template template, // holds the template parameter values
  C element) // to be recursively substituted
```

Let us take again our stack example (Section 4.3). The following template class, defined as a block template parameter, specifies the binding for the bound test code.

```
public class TestBoundTemplate
             extends BlockTemplateParameter
             implements Template {
  TemplateParameter<Integer> _max_;
  List elements; // shadow for the elements of the stack
  // parameter binding
  public TestBoundTemplate(CtFactory f,int max) {
    _max_= f.Code.createLiteral(max);
  }
  public void block() {
    if(elements.size() > _max_.S())
      throw new OutOfBoundException();
} }
```

Once the binding is defined, the template's result can be inserted at the beginning of the `push` method. This code replaces the `getTestCode` method used in the processor of Section 4.3.

```
CtMethod push= ...;
push.getBody().insertBegin(new MyTemplate(f,bound.max())
     .getSubstitution(push.getParent()));
```

Finally, we can also use templates and meta-model introspection together to perform in-depth transformations of the program. For instance, although our stack processor works, it might lead to inconsistent code because it adds a checked exception throw. All the code that uses the `push` method should hence be transformed in order to explicitly deal with this new exception. For this, we can define a processor that processes all the methods of the program and checks if they use the `push` method (we use the query API presented in Section 3.2). If so, it will instantiates the following template and replaces the body with the result of the substitution. The effect is to surround the body with a try/catch that prints out the stack trace.

```
public class TryCatchOutOfBoundTemplate extends
       BlockTemplateParameter implements Template {
  TemplateParameter<Void> _body_; // the body to surround
  public TryCatchOutOfBoundTemplate(CtBlock body) {
    _body_=body;
  }
  public void block() {
    try { _body_.S(); }
    catch(OutOfBoundException e) { e.printStackTrace(); }
} }
```

5.4 Primitive Template Parameters

In the previous sections, we have seen template parameters that implement the `Template-Parameter<T>` interface. In particular, we have seen that they can represent statements and

expressions that can be substituted in a template by using the `S` method. In this section, we show other kind of template parameters called *primitive template parameters*, which are used to simplify the use of template parameters and to allow the substitution of other kinds of program elements.

When the parameter is known to be a literal (primitive types, boxing types, and `String`), a `Class`, or a one-dimension array of these types, it is not necessary to use a `TemplateParameter<T>`. In place of it, we can directly use the actual type of the parameter. However, to indicate to the substitution engine that a given field is a template parameter, it has to be annotated with a `@Parameter` annotation. By using this feature, it is not necessary to call the `S` method for a substitution. For instance, we can simplify our stack example as follows.

```
@Parameter int _max_;
public void block() {
  if(elements.size()>_max_) throw new OutOfBoundException();
}
```

When a primitive parameter is of a `Class` type, it can be used in two ways. First, it can be used to access the runtime class's instance, which is actually an access to the `class` static field in Java. Also, it can be used to substitute a declared type for a typed element (parameter, method, field, and local variable declarations). The following code shows the substitution of a template method that uses a `_C_` type as a template parameter. Note that this `_C_` type can be declared as a type parameter (generics), so that it avoids the declaration of intermediate types, and allows for typing through the use of upper bounds.

```
public class ClassParameterExample<_C_> implements Template {
 @Parameter Class _C_;
 public String m(_C_ p) {
  return _C_.toString()+p;
 }
}
```

substitution result with _C_=String.class:

```
public String m(String p) {
 return String.class.toString()+p;
}
```

Finally, in some cases, identifier names need to be parameterized. To do this, it is possible to use a `String` parameter with a carefully chosen name. Any occurrence of this name in any identifier (method names, variable names, type names) will be replaced by the parameter value. This kind of substitution should be carefully used, as it can easily produce inconsistent results. The following code shows an example.

```
public class IdentParameterExample implements Template {
 @Parameter String _s_;
 public String m() {
  String _s_String="hello";
  return _s_String;
 }
}
```

substitution result with _s_="p":

```
public String m() {
 String pString="hello";
 return pString;
}
```

5.5 Partial Evaluation

Spoon provides a meta-model partial evaluation facility implemented as a visitor, which can be applied to any code element of the meta-model. The partial evaluator returns a transformed model where the code is simplified when possible, that is to say, when the evaluated expressions contain several constant values, such as literals or final static field accesses (including the `class` access). For instance, "`if(-1 < 0) throw ...`" will be simplified as "`throw ...`". Another important feature of our partial evaluator is that it calculates the control flow and removes all the unreachable statements. Typically, to simplify a target method body:

```
target.setBody(
  new PartialEvaluator().evaluate(target,target.getBody()));
```

This partial evaluation facility is particularly useful when programming template-based transformations. Indeed, it is a common scenario to bind the template parameters with constant val-

ues. In these cases, the substituted code can sometimes be significantly simplified. For instance, using our partial evaluator allows us to implement an optimization for our stack example. With the following template code, we express that when the `_max_` value is equal to 0, we directly throw the exception rather than performing the test on the elements' size. Also, we implement as a convention that a negative value implies that the stack is not bounded.

```
public void block() {
  if(_max_==0) throw new OutOfBoundException();
  else if(_max_>0 && elements.size()>_max_)
      throw new OutOfBoundException();
}
```

If we apply the partial evaluator on the `push` method's body at the end of the processing, the `push` method is transformed as follows when `_max_` equals to 5.

```
public void push(T element) throws OutOfBoundException {
  if(elements.size() > 5) throw new OutOfBoundException();
  elements.add(0,element);
}
```

The tests have been simplified by the partial evaluator since "`_max_==0`" statically evaluates to `false` and "`_max_>0`" to `true`. Similarly, if the bound is set to `0`, the resulting `push` method is simplified to the only exception throw, since the partial evaluator removes any unreachable statement by default. Finally, for a negative value, no code is generated and there is no runtime overhead for the transformation. As we can see, templates and partial evaluation used jointly produce meaningful and efficient code. It allows for the writing of templates that depend on the static context in order to select the right transformation to be applied.

6 Applications

In this section, we present three useful applications of the Spoon framework. These applications are not meant to be complete but have been developed as proofs of usability for solving complex program analysis and transformation problems. In Section 6.1, we present a Java 1.4 to Java 5 program translator. Section 6.2 implements an efficient AOP engine using templates and partial evaluation. Finally, Section 6.3 discusses an automatic implementation of the visitor design pattern.

6.1 Java 1.4 to Java 5 Translator

Java 5 is backwards compatible. However using a Java 1.4 legacy code with Java 5 source code or third-party libraries generates a great deal of type-safety warnings. This is a common scenario given that the Java 5 API, in particular the collection framework, is updated to use generics. In order to avoid these warnings, it is necessary to update the legacy code to include type parameters, which is a time-consuming, error-prone, and repetitive task. Therefore, it is desirable to make this task automatic by means of refactoring. Such a refactoring has been proposed before [5][6] and is currently implemented in some IDEs (Eclipse, Idea). As a proof-of-concept, we implemented it in an IDE-independent manner with Spoon.
Also, relying on generics, Java 5 introduced a new language construct called *Enhanced For Loop* (a.k.a. *foreach*). This *foreach* construct is syntactic sugar that does away with the explicit use of iterators. Here, we propose a refactoring that uses Spoon processors to translate regular *for* iterator-based loops into their *foreach* equivalent.

6.1.1 Generics

Including type parameters automatically requires static type analysis. This analysis consists of checking variables uses for inferring their type parameter(s). To do so, we chose the algorithm described by Fuhrer et al. [6], which uses a constraint-based approach, instead of a context based one [5].

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 (in)equation system whose solution will yield the expected type parameters. In the second stage, once all constraints are derived, the equation system is solved, and a single type solution is associated to each expression. For example, in the following table, a fragment of source code and the relevant derived constraints are shown[6].

`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)`

After having found a set of types that satisfy all the derived constraints, the program is then transformed into a generics-compliant Java 5 program.

6.1.2 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 [6].

Constraints are derived using 5 processors: `AssigmentProcessor`, `ClassDefinitionProcessor`, `InvocationProcessor`, `ReturnProcessor` and, `VariableDefinitionProcessor`. Each processor 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.

Finally, redundant casts are eliminated by `CastInvocationRemoveProcessor`. This processor, 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.

6.1.3 Template-based Loop Transformations

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 (left part of Fig. 5), and replace the *for* loop with its equivalent (right part of Fig. 5). We have implemented a Spoon processor that performs this task for this particular kind of *for* loops. Extending it to other kinds of *for* loops, or even to other looping statements, is a matter of creating new processors and new matching policies.

```
for(Iterator it = b.iterator();
                  it.hasNext();){
   String s = (String)it.next();
   System.out.println(s);
}
```

```
for(String s:b) {
  System.out.println(s);
}
```

Fig. 5. A translatable *for* loop (left) and its translated *foreach* version (right).

Loops are transformed by `ForeachProcessor`. When a *for* loop is found (`CtFor` elements), it

[6] `[a]` stands for "the type of expression `a`", and `E(b)` stands for "the type of the `E` type variable in expression `b`"

is tested to see if it matches the desired structure. We use CT reflection 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 `TemplateFinal` shown below, 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 5.4 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.

```
class TemplateFinal<_I_ extends Collection<?>>
         extends BlockTemplateParameter implements Template {
  TemplateParameter<Collection<_I_>> _collection_;
  TemplateParameter<Void> _body_;
  @Parameter Class _I_;
  @Parameter String _loopingVariable_;
  public void block() throws Throwable {
    for(_I_ _loopingVariable_ :  _collection_.S()) {
      _body_.S();
} } }
```

We have evaluated our approach by comparing it to the available Eclipse refactoring. In several scenarios, our approach gives better results in terms of removed warnings. Also, Eclipse can produce un-compilable code on tricky cases, which is not the case with our implementation.

6.2 Template-Based AOP

In this section, we present a template-based implementation of the advising mechanism, which is one of the core ones implemented by AOP compilers (*weavers*).

6.2.1 Before and After Advice as Templates.

In order to define before and after advising in Spoon, we have provided the **`BeforeAdvice`** and **`AfterAdvice<T>`** interface, which respectively define the "**`void before()`**" method and the "**`T after()`**" method, where **`T`** is the return type of the advised executable (method or constructor). These interfaces must be implemented by template classes in order to provide the code that should be inserted before or after a given target executable. Like regular template code, the before and after code can use template parameters that will be substituted when the insertion occurs. In order to bind the template parameters with the right values, the before or after template class can define a constructor that takes the target's executable as an argument. During the template instantiation, the advice can introspect the target in order to extract useful static information. For instance, the following advice template traces the execution of a method.

```
public class Trace implements BeforeAdvice,Template {
  @Parameter String _name_;
  public Trace(CtExecutable e) { _name_=e.getSimpleName(); }
  public void before() {
    System.out.println("calling "+_name_);
} }
```

For template-based AOP, we provide an annotation processor called `AdviceProcessor`, which processes the `Advice` annotation. This annotation allows the specification of the advice list to be applied to an executable. `AdviceProcessor` instantiates these advice templates and inserts the result of the `getSubstitution` method at the right place, depending on the advice type. Here is the result for the `Trace` advice:

```
@Advice(Trace.class)          public m() {
public m() {                    System.out.println("calling "+"m");
  {body}                  ▶     {body}
}                             }
```

The transformation for before advice is relatively simple. It consists of inserting the before code

at the beginning of the body. The only exception is the case of a constructor where the code has to be inserted after the `super` or `this` call if it exists. For after advising, things are more complex. `AdviceProcessor` has to search for all the return statements and insert the after code before them (actually, the semantics of our after advice is more like a *before return* advice). If the method returns a result, the returned expressions are stored in a temporary local variable so that the after code can access the returned value. For instance:

```
int m(int i) {                    int m(int i) {
  return i * i;                     int _RESULT_=i * i;
}                                   {inserted after code}
                                    return _RESULT_;
                                  }
```

Since the advice code is inlined, is can directly access the target executable's context such as fields (including `this`), parameters, and even local variables. This has several advantages but can also induce name clashes. In order to avoid those, `AdviceProcessor` renames all the advice's local variables by prefixing them with the name of the advice class and the advice kind (before or after).

6.2.2 Accessible Static Context and Partial Evaluation

To avoid the need to re-implement all the typical static context accesses, we have defined an abstract template named **`ExecutableJoinPointAccessor`**, which defines the bindings for frequently encountered template parameters, and that can be sub-classed by the advice programmer. In addition to manually bound parameters, this template automatically binds the following parameters:

- `CtBlock _body_`: the target's body,
- `String _executableName_`: the name of the target executable,
- `int _argumentsCount_`: the number of arguments,
- `Class<?>[] _argumentTypes_`: the types of the arguments,
- `CtVariableAccess<?>[] _argumentValues_`: the accesses to the parameters,
- `_returnType_ Class<T>`: the return type,
- `CtVariableAccess<T> _returnValue_`: the access to the variable where the result is stored.

It is important to add that `AdviceProcessor` runs the partial evaluator on the result of the transformation (see Section 5.5). Except `_argumentValues_`, `_returnValue_` (which are variable accesses) and the body, all the other template parameters are constants and can trigger important code optimizations during partial evaluation. As an example, take the following advice code.

```
public void before() {
  if(_argumentsCount_>0) {something} else {something else}
}
```

Because of partial evaluation, this advice code is simplified to keep only the useful parts and skip the test. In the next paragraph, we compare our template-based AOP approach with classical AOP approaches.

6.2.3 Template-Based AOP vs. Classical AOP

Classical approaches for AOP do not use templates and do not inline the advice code. This has the advantage of simplifying the weaver's task by separating the target's context from the advice's context. The only transformation that needs to be done is the generation of a stub method that up-calls the before advice, delegates to the original method, and up-calls the after advice. Because of this implementation strategy, weavers cannot optimize the generated code depending on statically known contextual information. When programming advice, it is, however, use-

ful to access this context in order to implement more generic and reusable advice. As said before, Spoon allows efficient compilation of this kind of generic advice by using its partial evaluator, and is thus far more efficient than most of the existing AOP implementations.

In order to demonstrate this, we have compared the execution speed of similar dummy programs written in Spoon and in AspectJ [8].

```
public class Empty implements BeforeAdvice, Template{
  public void before() {}
}

public class Contextual extends ExecutableJoinPointAccessor
                        implements BeforeAdvice, Template {
  public Contextual(CtExecutable e) { super(e); }
  public void before() {
    if(_argumentsCount_>0) {} else {}
}}
```

Fig. 6. Spoon version (template-based AOP).

```
public aspect Empty {
    before() : execution(* ToBeAdvised.*(..)) {}
}

public aspect Contextual {
  before() : execution(* ToBeAdvised.*(..)) {
    if(thisJoinPoint.getArgs().length>0) {} else {}
}}
```

Fig. 7. AspectJ version (classical AOP).

We have measured the execution speed of 20,000,000 invocations of two advised methods on the `ToBeAdvised` class. As shown in Table 1, Spoon has no overhead in both cases, which is normal because the contextual test is removed by the partial evaluator. On the other hand, AspectJ has an overhead (about x2) for the empty advice and a significant overhead (> x100) for the empty advice performing a contextual test.

Table 1. Spoon vs. AspectJ performance comparison. Reference time (no advice): 83 ms.

	Spoon (Fig. 6)	AspectJ (Fig. 7)
Empty advice	83 ms	193 ms
Contextual advice	83 ms	8570 ms

6.3 Visitor Design Pattern

In this section, we present another application of Spoon, which consists of automatically implementing and validating the visitor pattern by using a Spoon processor and the template mechanism.

6.3.1 Context

The visitor design pattern aims to modularize a global task that implies several local sub-tasks on a set of classes that are part of a class hierarchy with a common super-class. The idea is to use a visitor object (a.k.a. a switch object), which implements the needed tasks for each node of the hierarchy. In the example of Figure 8, we show a class hierarchy that models expressions, and its corresponding visitor. Typically, this visitor can be sub-classed in order to implement specific tasks on the expressions such as printing or evaluation.

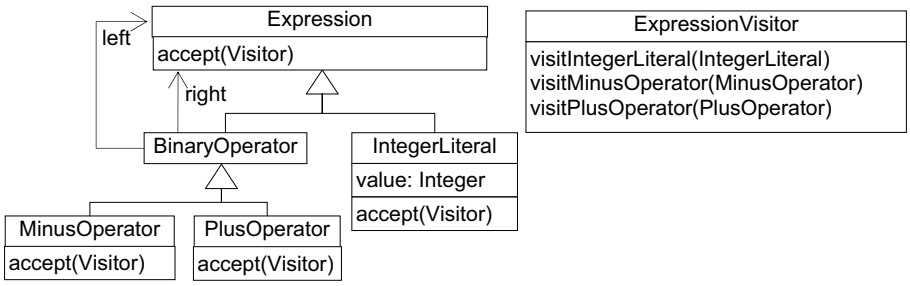

Fig. 8. A visitor design pattern example.

One of the main problems with visitor patterns is that they introduce a great deal of crosscutting within the class hierarchy with the `accept` methods. Because of this crosscutting effect, the maintenance and the evolution of visitor-based architectures are harder and lead to repetitive operations. For instance, when introducing a new node that extends an existing node, it is easy to forget to overload the `accept` method since the compiler will not detect any mistake. Besides, when refactoring the hierarchy, for instance by renaming a class, it is easy to forget to update the names of the visitation methods and the `accept` implementations, which can lead to name inconsistency in the best cases, or to runtime errors in the worst scenarios.
For these reasons, it is interesting to maintain better cohesion for the visited hierarchy and the visitors. Here, we are going to implement a Spoon processor that ensures better cohesion, and that facilitates the programmers' task at the same time, by automatically generating the `accept` implementations.

6.3.2 The Visitor Processor and accept Method Template

In order to ensure better cohesion for the visitor pattern, we propose to use a processor that automatically introduces the **`accept`** methods into the visited classes. This way, the hierarchy is totally oblivious to the visitor design pattern. When the processed code is compiled, it will report errors if the visitor is not consistent with the hierarchy, hence ensuring better cohesion.

The difficulty in implementing this processing with Spoon is the transformation that introduces the `accept` method. Indeed, its code depends on the context of application. In particular, the visitation method that must be called depends on the target class to which `accept` is added. This could be done using CT reflection. However, the use of a template leads to better typed and more understandable code. The following code shows the definition of the template for introducing the `accept` method in a generic way.

```
public interface _Visitor_{void visit_target_(Object o) {};}
public class VisitorTemplate implements Template {
  @Parameter String _target_;
  @Parameter Class _Visitor_;
  public VisitorTemplate(String target, Class visitorType) {
    _target_ = target;
    _Visitor_ = visitorType;
  }
  public void accept(_Visitor_ visitor) {
    visitor.visit_target_(this);
} }
```

We use an intermediate `_Visitor_` type, which represents the actual visitor type. We have to define this type since it allows for the definition of a `visit_target_` method that stands for the visitation method to be called. We then have to define a `Class` template parameter named `_Visitor_` that holds the actual visitor type, and a `String` template parameter named `_target_` that represents the target class' simple name. By doing so, all the references to the

`_Visitor_` type will be substituted by the actual visitor type, and all the identifiers that contain `"_target_"` in their names will be renamed by replacing the `"_target_"` substring with the value of the `_target_` template parameter (see Section 5.4 for type and identifier substitution). For instance, when `_Visitor_` is set to the `ExpressionVisitor` class, and that `_target_` is `"IntegerLiteral"`, the resulting `accept` method is:

```
public void accept(ExpressionVisitor visitor) {
  visitor.visitIntegerLiteral(this);
}
```

The only limitation that we have found in this approach is that it forces the definition of a `Visitable` interface that defines the `accept` method and that is automatically introduced by our processor. Then, when calling `accept` on a visited element, the programmer needs to cast it to `Visitable`, which makes the visitation methods' implementations more complicated.

7 Evaluation and Related Works

7.1 Performance

Table 2 gives the performance figures of the main tasks achieved while processing a program containing 84 top-level Java classes and a total of 2037 lines of code[7]. The applied processor does nothing, so that the processing figure shows the raw performance of meta-model scanning by the processing visitor. Of course, the actual figure varies depending on the number of applied processors and on their complexity. Finally, just for model building, processing, and pretty printing, the Spoon compilation chain is about 2.7 times slower than regular javac compilation. However, the performance remains satisfying (less than one second for about 2000 lines of code), and could be enhanced by implementing our own parsing, attribution, and bytecode generation engines.

Table 2. Spoon performance.

Task	parsing + attribution	building	processing	pretty-printing	compiling	total
Time (ms)	193.62	255.33	11.11	65.96	296.59	822,61

In order to evaluate the performance of an actual processing, we have measured the overhead of the processing phase when running the applications presented in Section 6. We give these overheads as a percentage that represents the ratio between Spoon running on the program with an empty processor (which is the reference figure given in Table 2) and the processing time of a given application.

Table 3. Spoon processing overhead for our applications.

Application	Generics (Section 6.1)	Advising (Section 6.2)	Visitor (Section 6.3)
Context	*38 classes, 1221 lines, 8 translatable loops*	*weaving of the trace advice on all the methods of the program (1 class, 40 methods)*	*insertion of "accept" into a hierarchy of 25 visited classes*
Overhead	type inference: + 152% analysis+transformation:	+52% (429 ms)	+23% (190 ms)

[7] Average execution time for 200 runs, under the following environment:
model name : Intel(R) Pentium(R) 4 CPU 3.00GHz HT
cpu MHz : 2993.991 cache size : 1024 KB
Linux version 2.6.12-10-686-smp
java version "1.5.0_05"
Java(TM) 2 Runtime Environment, Standard Edition version (build 1.5.0_05-b05)
Java HotSpot(TM) Client VM (build 1.5.0_05-b05, mixed mode, sharing)

	+17%

Table 3 summarizes the obtained results. The overhead depends on the processing complexity. For instance, the overhead for generics is mainly caused by type inference (type equation system resolution). The overhead of Spoon analysis and transformation is small compared to it. The advising and visitor applications give a good estimate for template instantiation and substitution time, which appears to be around 10 ms for our applications.

7.2 Related Work

OpenJava [14] is a macro system that uses *Meta-Objects* to represent the source code structure of a program. It uses type-driven translation, assigning a meta-object to each class in the program. Although OpenJava supports a kind of CT reflection similar to that of Spoon, it lacks an easy way of manipulating the model for the construction of source code structures; in contrast with Spoon's template mechanism.

Template Haskell [12] is an extension to Haskell that allows type-safe compile-time meta-programming by manipulating a reification of Haskell expressions with Haskell itself. To ensure the type safety of templates in Template Haskell, a special compiler with complex type-checking states is necessary. This comes as a disadvantage when compared to Spoon, which does not require a special compiler to provide the same kind of type safety.

Stratego [2] is a language for program transformation based on rewriting strategies that supports a variety of languages, including Java. Transformations are composed of conditional rewrite rules that match to a given pattern. Stratego's rewrite rules can be compared to Spoon processors. However, contrary to Stratego, Spoon transformations can be specified in pure Java, without the use of a specific language.

In [1], d'Amorim et al. introduce a tool named Coder, which is based on templates for the refactoring and transformation of Java programs. Coder's templates can be used as "source templates" to bind template parameters, and "target templates", which generate code using the parameters bound in the source templates. The later type of templates is similar to Spoon's; however, they do not offer any kind of type checking.

LogicAJ [9] is implemented by means of Conditional Transformations. These transformations are composed of a name, a precondition, and a transformation, which operates on a Prolog fact database containing the representation of a Java program. The Conditional Transformations are similar in spirit to those of Spoon, which also follow the schema of precondition - transformation. However, they require the use of a new language (LogicAJ) and they do not ensure any kind of type safety.

8 Conclusion

In this research report, we have presented the Spoon framework and how it can be used for well-typed program analysis and transformation. Spoon is an open compiler built using CT reflection, which allows for the writing of pure Java meta-programs within processors and annotation processors. In addition, Spoon's main breakthrough is to offer a pure Java and type-checked template facility that can be combined with CT reflection in order to specify intuitive and type-safe program transformations. We believe that it is a convenient programming model to support annotation-driven development and much more. Spoon shows that annotations and generics coupled with auto-boxing make a significant difference for implementing program manipulation frameworks. Not only do they make the program more type safe, but they also considerably improve the usability of these frameworks for the end users.

The current version of Spoon [10] covers all the Java language, which makes it usable from now on for experimentations and more serious developments. Besides the three applications presented in this report, we envision many uses of Spoon for the future. In particular, we think of the possibility of applying Spoon to itself in order to improve the quality of its code and detect design errors through static analysis. For instance, we could use our visitor processor to ensure better coupling between our meta-model and all its visitors. Also, we plan to extend the query API presented in Section 3.2 in order to support template matching based queries (similarly to Coder [1]). Indeed, using templates to look up fragments of code in the program is a promising technique to enhance the static analysis power of our approach.

Finally, all the concepts presented in Spoon for Java could be easily applicable to the .Net environment, since .Net also supports generics and annotations (called *attributes*). It would be interesting to investigate how Spoon could be applied to itself in order to self-translate into Spoon-C#, for instance.

9 References

[1] d'Amorimm M., Nogueira, C., Santos, G., Souza,A., Borba, P.: Integrating code generation and refactoring. In: Workshop on Generative Programming, 16th European Conference on Object-Oriented Programming, ECOOP'2002, Malaga, Spain (2002).

[2] Bravenboer, M. and Visser, E.: Concrete Syntax for Objects. Domain-Specific Language Embedding and Assimilation without Restrictions. In OOPSLA, pages 365--383, Vancouver, Canada, ACM Press (2004).

[3] Butler, G., Czarnecki, K., Batory D., and Eisenecker, U.: Generative techniques for product lines. In proceedings of ICSE, pages 760-761 (2001).

[4] Chiba, S.: A Metaobject Protocol for C++. In OOPLSA (1995).

[5] Donovan, A., Kiezun, A., Tschantz, M.S., Ernst, M.D.: Converting Java Programs to use generic libraries. In proceedings of OOPSLA, pages 15-34 (2004).

[6] Fuhrer, R., Tip, F., Kiezun, A., Dolby, J., Keller, M.: Efficient refactoring Java applica-tions to use generic libraries. In: European Conference Object-Oriented Programming (ECOOP 2005), Glasfow, Scotland (2005).

[7] Hoeniche, J.: Java Optimize and Decompile Environment (Jode). ver. 1.1.1. (2001). http://jode.sourceforge.net/.

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G.: An Overview of AspectJ, In proceedings of ECOOP (2001).

[9] Kniesel, G., Rho, T., Hanenberg, S.: Evolvable Pattern Implementations Need Generic Aspects. In: Workshop on Reflection, AOP and Meta-Data for Software Evolution, European Conference on Object-Oriented Programming, ECOOP'2004.

[10] Noguera, C., Pawlak, R., Petitprez, N.: Spoon home page : http://spoon.gforge.inria.fr/.

[11] Pawlak, R.: Spoon: Annotation-Driven Program Transformation — the AOP Case. In the first workshop on Aspect-Orientation for Middleware Development, Middleware (2005).

[12] Sheard, T., Jones, S. P.: Template meta-programming for Haskell. In: Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell (2002).

[13] SUN: Annotation Processing Tool. http://java.sun.com/j2se/1.5.0/docs/guide/apt/

[14] Tatsubori, M., Chiba, S., Killijian, M., Itano K.: OpenJava: A Class-Based Macro System for Java. In: Reflection and Software Engineering, Papers from OORaSE 1999, 1st OOPSLA Workshop on Reflection and Software Engineering (2000).

10 Table of Contents

Spoon: Program Analysis and Transformation in Java 1
1 Introduction 3
2 Overview 3
3 Well-typed Compile-time Reflection 4
3.1 Spoon's Java 5 Meta-model 5
3.2 Queries 7
3.2.1 Meta-model Modifications and Factory 7
4 Program Processing 8
4.1 Processors 8
4.2 Annotation Processors 8
4.3 Example 9
5 Template-Based Code-Level Transformations 10
5.1 Pure Java Templates 10
5.2 Template Parameters and Substitution 11
5.3 Template Instantiation 12
5.4 Primitive Template Parameters 13
5.5 Partial Evaluation 13
6 Applications 14
6.1 Java 1.4 to Java 5 Translator 14
6.1.1 Generics 15
6.1.2 Implementation 15
6.1.3 Template-based Loop Transformations 15
6.2 Template-Based AOP 16
6.2.1 Before and After Advice as Templates. 16
6.2.2 Accessible Static Context and Partial Evaluation 17
6.2.3 Template-Based AOP vs. Classical AOP 18
6.3 Visitor Design Pattern 19
6.3.1 Context 19
6.3.2 The Visitor Processor and accept Method Template 19
7 Evaluation and Related Works 20
7.1 Performance 20
7.2 Related Work 21
8 Conclusion 22
9 References 22
10 Table of Contents 24