# Spoon User Manual

Martin Monperrus Gerard Paligot Simon Urli Nicolas Harrand
Pavel Vojtechovsky Gérard Paligot Thomas Durieux Egor Bredikhin
Martin Witt Alexander Shopov Benjamin Danglot et. al.

version of 2020-07-19, commit beed357cc3b3e776efb5e283cf394dfdeac6afdc

## Contents

# Introduction

This PDF is generated from the markdown files that are in https://github.com/INRIA/spoon/tree/master/doc

If you notice an error, it would be great if you can do a pull-request on the corresponding files.

An update of this PDF is done regularly[1].

If you need professional support (training, consultancy) on Spoon, post a comment on https://github.com/INRIA/spoon/issues/3251

# Getting started

## Core concepts

### Abstract Syntax Tree

An Abstract Syntax Tree, also known as ASTs, is a a tree-based representation of source code. Spoon is a library to build and manipulates ASTs of Java source code.

[1] with `create-spoon-book.sh` at https://github.com/monperrus/spoon/blob/create-spoon-book.sh/doc/create-spoon-book.sh

With Spoon, you can visualize the AST of a Java file, as follows:

```
$ java -cp spoon-core-{{site.spoon_release}}-jar-with-dependencies.jar spoon.Launcher \
   -i MyClass.java --gui
```

If you have Java 11 with Java FX, there is a new GUI, see https://github.com/INRIA/spoon/tree/master/spoon-visualisation

### Code Analysis

Spoon is a tool for doing static code analysis at the source code level.

In Spoon, the core concept for code analysis is a processor. A processor analyses all AST elements of a given type, one per one.

For a first processor, we'll analyze all catch blocks of a `try {...} catch {...}` element to know how many empty catch blocks we have in a project. This kind of empty catch can be considered bad practice. That could be a great information to know how many and where are these catches in a project to fill them with some code, e.g. throws a runtime exception or logs the exception.

```
// file processors/CatchProcessor.java
package processors;

import org.apache.log4j.Level;
import spoon.processing.AbstractProcessor;
import spoon.reflect.code.CtCatch;

/**
 * Reports warnings when empty catch blocks are found.
 */
public class CatchProcessor extends AbstractProcessor<CtCatch> {
    public void process(CtCatch element) {
        // we get all statements and if there isn't statement, it means the block catch is
            empty!
        if (element.getBody().getStatements().size() == 0) {
            getFactory().getEnvironment().report(this, Level.WARN, element, "empty catch
                clause");
        }
    }
}
```

This processor extends `AbstractProcessor` (javadoc). This super class takes a generic type parameter to know what type you want inspect in a AST. For this tutorial, we inspect a catch, a `CtCatch` (javadoc).

When the class `AbstractProcessor` (javadoc) is extended, we implement the method `void process(E element)` where `E` is a generic type for any elements of the AST (all classes in the Spoon meta model which extends `CtElement` (javadoc)). It is in this method that you can access all information you want of the the current `CtCatch` (javadoc).

Next, you can compile your processor. You can use javac in command line to generate the `CatchProcessor.class` file. Then we execute Spoon as follows to analyze all catch blocks in Java files that are in `/path/to/src/of/your/project`:

```
$ java -classpath /path/to/processor.jar:spoon-core-{{site.spoon_release}}-jar-with-
    dependencies.jar \
    spoon.Launcher -i /path/to/src/of/your/project -p processors.CatchProcessor
```

{{site.data.alerts.important}} 1. Specify all dependencies in your classpath. If your processor has dependencies, don't forget to package your processor.jar with all dependencies! 2. Specify your processors in fully qualified name (here `processors.CatchProcessor`). {{site.data.alerts.end}}

There are many more examples of source code analysis in https://github.com/SpoonLabs/spoon-examples.

### Code Transformation

An unique feature of Spoon is its ability to automatically transform source code, for instance for instrumentation or refactoring, see section "Transformation" below.

## Using Spoon for Architecture Enforcement

In software, an architectural rule (aka architectural constraint) specifies a design decision on the application. Architectural rules cannot usually be expressed in the programming language itself. Architectural rules can be written as AST analysis, which makes Spoon very appropriate to express and check them. Since architectural rules must be automatically checked as often as possible, it's good to have them part of continuous integration.

To write an architectural rule in Spoon that is checked in CI, the idea is to write a standard Junit test case that loads the application code, express the rule and check it. Doing this only requires to depend on Spoon at testing time, ie `<scope>test</scope>` in Maven.

### Example rule: never use the TreeSet constructor

For instance, let's imagine that you want to forbid the usage of `TreeSet`'s constructor, in your code base, you would simply write a test case as follows:

```
@Test
void noTreeSet() throws Exception {
    SpoonAPI spoon = new Launcher();
    spoon.addInputResource("src/main/java/");
    spoon.buildModel();

    assertEquals(0, spoon.getFactory().Package().getRootPackage().getElements(new
        AbstractFilter<CtConstructorCall>() {
        @Override
        public boolean matches(CtConstructorCall element) {
            return element.getType().getActualClass().equals(TreeSet.class);
        };
    }).size());
}
```

That's it! Every time you run the tests, incl. on your continuous integration server, the architectural rules are enforced.

For instance, you can check that you never return null, or always use an appropriate factory, or that all classes implementing an interface are in the same package.

### Example rule: all test classes must start or end with "Test"

A common mistake is to forget to follow a naming convention. For instance, if you use Maven, all test classes must be named `Test*` or `*Test` in order to be run by Maven's standard test plugin `surefire` (see doc). This rule simply reads:

```
@Test
public void testGoodTestClassNames() throws Exception {
    SpoonAPI spoon = new Launcher();
    spoon.addInputResource("src/test/java/");
    spoon.buildModel();

    for (CtMethod<?> meth : spoon.getModel().getRootPackage().getElements(new TypeFilter<
        CtMethod>(CtMethod.class) {
            @Override
            public boolean matches(CtMethod element) {
                    return super.matches(element) && element.getAnnotation(Test.class) != null
                        ;
            }
    })) {
```

```
            assertTrue("naming contract violated for "+meth.getParent(CtClass.class).
                getSimpleName(), meth.getParent(CtClass.class).getSimpleName().startsWith("
                Test") || meth.getParent(CtClass.class).getSimpleName().endsWith("Test"));
    }
}
```

#### Example rule: all public methods must be documented

How to check that all public methods of the API contain proper Javadoc? One can also use Spoon to check documentation rules like this one.

```
@Test
public void testDocumentation() throws Exception {
    SpoonAPI spoon = new Launcher();
    spoon.addInputResource("src/main/java/");
    spoon.buildModel();
    List<String> notDocumented = new ArrayList<>();
    for (CtMethod method : spoon.getModel().getElements(new TypeFilter<>(CtMethod.class))) {
            // now we see whether this should be documented
            if (method.hasModifier(ModifierKind.PUBLIC)
                && method.getTopDefinitions().size() == 0 // optional: only the top
                    declarations should be documented (not the overriding methods which are
                    lower in the hierarchy)
            )) {
                    // is it really well documented?
                    if (method.getDocComment().length() < 20) { // at least 20 characters
                            notDocumented.add(method.getParent(CtType.class).getQualifiedName
                                () + "#" + method.getSignature());
                    }
            }
    }
    if (notDocumented.size() > 0) {
            fail(notDocumented.size()+" public methods should be documented with proper API
                documentation: \n"+StringUtils.join(notDocumented, "\n"));
    }
}
```

#### Related work in architecture enforcement

- Architecture enforcement with Checkstyle
- Sonar architecture rule engine
- Archunit: specify and assert architecture rules in plain Java (Maven integration)
- jqassistant does architectural checking on top of Neo4J.

## Using Spoon as a Better Runtime Reflection API

Spoon can be used as an alternative to the standard Java reflection API. Instead of manipulating a `java.lang.Class` object, you manipulate `CtClass` objects (also "shadow classes" of the normal binary classes). The advantage is that you can share code for analyzing both binary and source code.

To do this, use `TypeFactory` as follows:

```
CtType s = new TypeFactory().get(String.class);
System.out.println(s.getSimpleName());
```

Spoon also provides you with reflection over the Spoon metamodel itself, in class `spoon.Metamodel`:

```
Set<CtType> list = Metamodel.getAllMetamodelInterfaces();
for(CtType t : l) {
    System.out.println(t.getMethods());
}
```

If you need to analyze the method bodies of binary code, have a look at spoon-decompiler.

## More about Spoon Processors

A program analysis is a combination of query and analysis code. In Spoon, this conceptual pair is reified in a `processor`. A Spoon processor is a class that focuses on the analysis of one kind of program elements. For instance, the processor at the end of this page 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). At any time, you can interrupt the processing of the model with a call to `interrupt()` (this stops all processors, and proceeds with the next step which is usually pretty-printing the code to disk).

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 applied to 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.

{{site.data.alerts.tip}} Spoon provides developers with an intuitive Java metamodel and concise abstractions to query and process AST elements. {{site.data.alerts.end}}

```
package fr.inria.gforge.spoon.processors;

import org.apache.log4j.Level;
import spoon.processing.AbstractProcessor;
import spoon.reflect.code.CtCatch;

/**
 * Reports warnings when empty catch blocks are found.
 */
public class CatchProcessor extends AbstractProcessor<CtCatch> {
    public void process(CtCatch element) {
        if (element.getBody().getStatements().size() == 0) {
            getFactory().getEnvironment().report(this, Level.WARN, element, "empty catch
                clause");
        }
    }
}
```

### Examples of Analysis Processors

The HelloWorldProcessor example prints hello world with compile-time reflection.

The CatchProcessor example detects empty catch blocks.

The ReferenceProcessor example detects circular references between packages.

This Factory example example detects wrong uses of the factory pattern.

### Parallel Processor

Lets assume you want to use multiple cores for your processor. Spoon provides a simple high-level API for this task. Using the CatchProcessor from before create a `AbstractParallelProcessor`.

```
            Processor<CtCatch> parallelProcessor = new AbstractParallelProcessor<CtCatch>(
                Arrays.asList(new CatchProcessor(), new CatchProcessor())) {};
```

Now you have the same processor behavior as before, but 2 parallel running processor. You can upscale this pretty high, but keep in mind to not use more parallel processors than available cores for maximum speedup. For more information about parallel processor and the API have a look in the documentation.

## FAQ

### Where is the Javadoc?

The javadoc is at http://spoon.gforge.inria.fr/mvnsites/spoon-core/apidocs

### Are there snapshots versions deployed somewhere?

```
<dependencies>
    <dependency>
        <groupId>fr.inria.gforge.spoon</groupId>
        <artifactId>spoon-core</artifactId>
        <version>{{site.spoon_snapshot}}</version>
    </dependency>
</dependencies>
<repositories>
    <repository>
      <id>maven.inria.fr-snapshot</id>
      <name>Maven Repository for Spoon Snapshots</name>
      <url>https://maven.irisa.fr/artifactory/spoon-public-snapshot</url>
    </repository>
</repositories>
```

### How to access Spoon's source code repository?

Spoon is developed on GitHub at https://github.com/INRIA/spoon/. You can browse the Spoon source using code intelligence (Go-to-definition, Find References, and Hover tooltips) at https://sourcegraph.com/github.com/INRIA/spoon.

### What is the meaning of each digit in the version X.Y.Z of spoon?

- X is the digit for the major version (major new features or major incompatible API changes).
- Y is the digit for the minor version (bug fixes or minor API changes).
- Z is the digit for the critical bug fixes of the current major or minor version.

### Where is the Spoon metamodel?

The Spoon metamodel consists of all interfaces that are in packages `spoon.reflect.declaration` (structural part: classes, methods, etc.) and `spoon.reflect.code` (behavioral part: if, loops, etc.).

### How to prevent Annotation processors from consuming the annotations that they process?

By default, whenever an Annotation Processor processes a CtElement it will consume (delete) the processed annotation from it. If you want the annotation to be kept, override the init() method from the `AbstractAnnotationProcessor` class, and call the protected method `clearConsumedAnnotationTypes` like so:

```
@Override
public void init() {
    super.init();
    clearConsumedAnnotationTypes();
}
```

### How to compare and create type references in a type-safe way?

Use actual classes instead of strings.

```
CtTypeReference t=...
if(t.getActualClass()==int.class) { ... }
Factory f=...
t=f.Type().createReference(int.class);
```

#### How to set the JDT compiler arguments?

`SpoonModelBuilder` exposes a method named `build(JDTBuilder)`. This method compiles the target source code with data specified in the JDTBuilder parameter.

```
final String[] builder = new JDTBuilderImpl() //
        .classpathOptions(new ClasspathOptions().classpath(TEST_CLASSPATH).bootclasspath(
            TEST_CLASSPATH).binaries(".").encoding("UTF-8")) //
        .complianceOptions(new ComplianceOptions().compliance(8)) //
        .annotationProcessingOptions(new AnnotationProcessingOptions().compileProcessors()) //
        .advancedOptions(new AdvancedOptions().continueExecution().enableJavadoc().
            preserveUnusedVars()) //
        .sources(new SourceOptions().sources(".")) //
        .build();
```

# Querying source code elements

## Navigation and Query

### Getters

All elements provide a set of appropriate getters to get the children of an element.

```
methods = ctClass.getMethods();
```

In addition, there exists a generic getter based on the role played by an element with respect to its parent. See CtRole for a complete list of roles.

```
methods = ctClass.getValueByRole(CtRole.METHOD);
```

While not recommended, it is also possible to get all direct children of an element

```
allDescendants = ctElement.getDirectChildren();
```

### Filters

A Filter defines a predicate of the form of a `matches` method that returns `true` if an element has to be selected in the filtering operation. A Filter is given as parameter to `CtElement#getElement(Filter)` ()or `CtQueryable#filterChildren(Filter)`) which implements a depth-first search algorithm. During AST traversal, the elements satisfying the matching predicate are selected by the filter.

Here are code examples about the usage of filters. The first example returns all AST nodes of type `CtAssignment`.

```
// collecting all assignments of a method body
list1 = methodBody.getElements(new TypeFilter(CtAssignment.class));
```

The second example selects all deprecated classes.

```
// collecting all deprecated classes
list2 = rootPackage.getElements(new AnnotationFilter(Deprecated.class));
```

Now let's consider a user-defined filter that only matches public fields across all classes.

```
// creating a custom filter to select all public fields
list3 = rootPackage.filterChildren(
  new AbstractFilter<CtField>(CtField.class) {
    @Override
    public boolean matches(CtField field) {
      return field.getModifiers.contains(ModifierKind.PUBLIC);
    }
  }
).list();
```

### Scanners

`CtScanner` provides a simple way to visit a node and its children.

```
//Scanner counting the number of CtFieldWrite
class CounterScanner extends CtScanner {
    private int visited = 0;
    @Override
    public <T> void visitCtFieldWrite(CtFieldWrite<T> fieldWrite) {
        visited++;
    }
}

CounterScanner scanner = new CounterScanner();

//Run the scanner on an element, here the CtClass representing FieldAccessRes
launcher.getFactory().Class().get("FieldAccessRes").accept(scanner);

//scanner.visited now contains the number of children of type CtFieldWrite
assertEquals(1, scanner.visited);
```

`EarlyTerminatingScanner` is a specialized Class implementing `CtScanner` that stops once `terminate()` has been called.

See also `CtVisitor`.

### Iterator

`CtIterator` provides an iterator on all transitive children of a node in depth first order.

```
CtIterator iterator = new CtIterator(root);
while (iterator.hasNext()) {
    CtElement el = iterator.next();
    //do something on each child of root
}
```

`CtBFSIterator` is similar to CtIterator but in Breadth first order.

### Queries

The Query, introduced in Spoon 5.5 by Pavel Vojtechovsky, is an improved filter mechanism:

- queries can be done with a Java 8 lambda
- queries can be chained
- queries can be reused on multiple input elements

**Queries with Java8 lambdas**: `CtQueryable#map(CtFunction)`enables you to give Java 8 lambda as query.

```
// returns a list of String
list = package.map((CtClass c) -> c.getSimpleName()).list();
```

**Compatibility with existing filters** `CtQueryable#filterChildren(Filter)` is a filtering query that can be chained:

```
// collecting all methods of deprecated classes
list2 = rootPackage
    .filterChildren(new AnnotationFilter(Deprecated.class)).list()
```

A boolean return value of the lambda tells whether the elements are selected for inclusion or not.

```
// creating a custom filter to select all public fields using java 8 lambda
list3 = rootPackage.filterChildren((CtField field)->field.getModifiers.contains(ModifierKind.
    PUBLIC)).list();
```

**Chaining** If the CtFunction returns an object, this object is given as result to the next step of the query. All results of the last query step are results of the query. The results of intermediate steps are not results of the query.:

```
// a query which processes all not deprecated methods of all deprecated classes
rootPackage
  .filterChildren((CtClass clazz)->clazz.getAnnotation(Deprecated.class)!=null)
  .map((CtClass clazz)->clazz.getMethods())
  .map((CtMethod<?> method)->method.getAnnotation(Deprecated.class)==null)
  .list()
;
```

Finally, if the CtFunction returns and Iterable or an Array then each item of the collection/array is sent to next query step or result.

**Query reuse**. Method `setInput` allows you to reuse the same query over multiple inputs. In such case it makes sense to create unbound query using `Factory#createQuery()`.

```
// here is the query
CtQuery q = factory.createQuery().map((CtClass c) -> c.getSimpleName());
// using it on a first input
String s1 = q.setInput(cls).list().get(0);
// using it on a second input
String s2 = q.setInput(cls2).list().get(0);
```

**Query evaluation**. Each example above use `CtQuery#list()` to evaluate the query. The `list` method evaluates the query and returns the `List`, which contains all the results of the query.

But it is not the only way how to evaluate query. There is `CtQuery#forEach(CtConsumer)`, which sends each query result to the `CtConsumer#accept` function. It is more efficient in cases when query results can be immediately processed.

```
//prints each deprecated element
rootPackage
    .filterChildren(new AnnotationFilter(Deprecated.class)).forEach((CtElement ele)->System.
        out.println(ele));
```

Finally there is `CtQuery#first()`, which evaluates the query until first query result is found. Then the evaluation is terminated and first result is returned. It is faster then `query.list().get(0)`, because query engine does not collect other results.

```
// returns first deprecated element
CtElement firstDeprecated = rootPackage.filterChildren(new AnnotationFilter(Deprecated.class))
    .first();
```

## Path

`CtPath` (javadoc) defines the path to a `CtElement` (javadoc) in a model, similarly to XPath for XML. For example, `.spoon.test.path.testclasses.Foo.foo#body#statement[index=0]` represents the first statement of the body of method foo. A `CtPath`is based on: names of elements (eg `foo`), and roles of elements with respect to their parent (eg `body`). A role is a relation between two AST nodes. For instance, a "then" branch in a if/then/else is a role (and not an node). All roles can be found in `CtRole`. In addition, each getter or setter in the metamodel is annotated with its role.

### Evaluating AST paths

Paths are used to find code elements, from a given root elements.

```
path = new CtPathStringBuilder().fromString(".spoon.test.path.testclasses.Foo.foo#body#
    statement[index=0]");
List<CtElement> l = path.evaluateOn(root)
```

### Creating AST paths

### From an existing element

Method `getPath` in `CtElement` returns a path

```
CtPath path = anElement.getPath();
```

### From a string

`CtPathStringBuilder` (javadoc), creates a path object from a string according to the following syntax:

- `.<name>` which denotes a child element with name `name`, eg `.fr.inria.Spoon` (the fully qualified name)
- `#<role>` which denotes all children on `CtRole role` .statements, `#body#statement[index=2]#else` is the else branch of the second statement of a method body
- `name=<somename>` - filter which accepts only elements with `somename`. E.g. `#field[name=abc]`
- `signature=<somesignature>` - filter which accepts only methods and constructors with signature `somesignature`.
  - Example of method signature: `#method[signature=compare(java.lang.String,java.lang.String)]`
  - Example of constructor signature: `#constructor[signature=(int)]`
- `index=<idx>` - fitler which accepts only idx-th element of the List. The first element has index 0. the fifth type memeber in a class `#typeMember[index=4]`

### From the API

The low-level `CtPathBuilder` (javadoc) defines a fluent api to build your path:

- `name(String, String[])` (javadoc) adds a name matcher to the current path.
- `type(Class, String[])` (javadoc) matches on element of a given type.
- `role(CtPathRole, String[])` (javadoc) matches on elements by their role (where `CtPathRole` gives all constants supported).
- `wildcard()` (javadoc) matches only on elements child of current one.
- `recursiveWildcard()` (javadoc) matches on any child and sub-children.

For instance, if we want all elements named by "toto" and with a default value in a project. Use `CtPathBuilder` like the example below.

```
CtPath p1 = new CtPathBuilder().recursiveWildcard().name("toto").role(CtPathRole.DEFAULT_VALUE
    ).build();
// equivalent to
CtPath p2 = new CtPathStringBuilder().fromString(".**.toto#default_value").build();

// takes all elements named "toto" in the project.
```

```
new CtPathBuilder().recursiveWildcard().name("toto")

// takes the first element named "toto", a package or a class in the default package, at the
    root of your project.
new CtPathBuilder().name("toto").recursiveWildcard()
```

## Matching elements

Spoon provides a way to declaratively specify a code snippet to match, this is called a `TemplateMatcher`. For instance, the following snippet matches any if statement parametrized with a collection expression:

```
if (_col_.S().size() > 10)
  throw new IndexOutOfBoundsException();
```

It would match the following code elements:

```
// c is a local variable, a method parameter or a field;
if (c.size() > 10)
  throw new IndexOutOfBoundsException();
---
//foo() returns a collection
if (foo().size() > 10)
  throw new IndexOutOfBoundsException();
```

To define a template matcher one must:

1. specify the "holes" of the template matcher
2. write the matcher in a dedicated method
3. instantiate TemplateMatcher and call method `find` or use it as Filter of a query.

Taking again the same example.

```
public class CheckBoundMatcher {
  // Step 1:
  public TemplateParameter<Collection<?>> _col_;

  // Step 2
  public void matcher1() {
    if (_col_.S().size() > 10)
      throw new IndexOutOfBoundsException();
  }
}

// Step 3, for instance in a main
// where to find the matching specification
CtClass<?> templateKlass = factory.Class().get(CheckBoundMatcher.class);
CtIf templateRoot = (CtIf) ((CtMethod) templateKlass.getElements(new NameFilter("matcher1")).
    get(0)).getBody().getStatement(0);
TemplateMatcher matcher = new TemplateMatcher(templateRoot);
for (CtElement elems : matcher.find(aPackage)) { ... };
//or TemplateMatcher as a Filter of query
aPackage.filterChildren(matcher).forEach((CtElement elem)->{ ... });
```

For named elements, a wildcard can be specified: if the named element (eg a method) to be matched is called `f` and the template matcher class contains a template parameter called `f` (of type Object), all methods starting by `f` will be matched.

Note, the matching process ignores some information in the AST nodes: comments; position; implicitness and casts. See `roleToSkippedClass` in class ElementNode

## Spoon Patterns

Spoon patterns enables you to find code elements. A Spoon pattern is based on a one or several AST nodes, which represent the code to match, where some parts of the AST are pattern parameters. When a pattern is matched, one can access to the code matched in each pattern parameter.

The main classes of Spoon patterns are those in package `spoon.pattern`:

- classes: PatternBuilder, Pattern, Match, PatternBuilderHelper, PatternParameterConfigurator, InlinedStatementConfigurator
- eums: ConflictResolutionMode, Quantifier

See also examples in project `spoon-examples`

### Example usage

```
Factory spoonFactory = ...
Pattern pattern = PatternBuilder.create(mainClass.getMethodsByName("m1").get(0).getBody().
    clone()).configurePatternParameters().build();

//search for all occurences of the method in the root package
pattern.forEachMatch(spoonFactory.getRootPackage(), (Match match) -> {
    Map<String, Object> parameters = match.getParametersAsMap();
    CtMethod<?> matchingMethod = match.getMatchingElement(CtMethod.class);
    String aNameOfMatchedMethod = parameters.get("methodName");
    ...
});
```

### PatternBuilder

To create a Spoon pattern, one must use `PatternBuilder`, which takes AST nodes as input, and *pattern parameters* can be defined.

```
// creates pattern from the body of method "matcher1"
elem = mainClass.getMethodsByName("matcher1").get(0).getBody().clone()
Pattern t = PatternBuilder.create(elem)
    .build();
```

If you call `configurePatternParameters()`, all variables that are declared outside of the AST node are automatically declared a pattern parameter.

```
Pattern t = PatternBuilder.create(elem)
    .configurePatternParameters()
    .build();
```

One can also create specific parameters with `.configureParameters(pb -> pb.parameter("name").byXXXX` (see below)

### Pattern

Once a `PatternBuilder` returns a `Pattern`, the main methods of `Pattern` are `getMatches` and `forEachMatch`.

```
List<Match> matches = pattern.getMatches(ctClass);
```

### Match

A `Match` represent a match of a pattern on a code elements. The main methods are `getMatchingElement` and `getMatchingElements`.

### PatternBuilderHelper

`PatternBuilderHelper` is useful to select AST nodes that would act as pattern. See method to get the body (method `setBodyOfMethod`) or the return expression of a method (method `setReturnExpressionOfMethod`).

### PatternParameterConfigurator

To create pattern paramters, one uses a `PatternParameterConfigurator` as a lambda:

```
//a pattern model
void method(String _x_) {
    zeroOneOrMoreStatements();
    System.out.println(_x_);
}

//a pattern definition
Pattern t = PatternBuilder.create(...select pattern model...)
    .configureParameters(pb ->
        // creating a pattern parameter called "firstParamName"
        pb.parameter("firstParamName")
            //...select which AST nodes are parameters...
            //e.g. using parameter selector
            .bySimpleName("stmt")

        //... you can define as many parameters as you need...

        // another parameter (all usages of variable "_x_"
        pb.parameter("lastParamName").byVariable("_x_");
    )
    .build();
```

`ParametersBuilder` has many methods to create the perfect pattern parameters, incl:

- `byType(Class|CtTypeReference|String)` - all the references to the type defined by Class, CtTypeReference or qualified name are considered as pattern parameter
- `byLocalType(CtType<?> searchScope, String localTypeSimpleName)` - all the types defined in `searchScope` and having simpleName equal to `localTypeSimpleName` are considered as pattern parameter
- `byVariable(CtVariable|String)` - all read/write variable references to CtVariable or any variable named with the provided simple name are considered as pattern parameter
- `byInvocation(CtMethod<?> method)` - all invocations of `method` are considered as pattern parameter
- `byVariable(CtVariable|String... variableName)` - each `variableName` is a name of a variable which references instance of a class with fields. Each such field is considered as pattern parameter.
- `byFilter(Filter)` - any pattern model element, where `Filter.accept(element)` returns true is a pattern parameter.
- `byRole(CtRole role, Filter filter)` - the attribute defined by `role` of all pattern model elements, where `Filter.accept(element)` returns true is a pattern parameter. It can be used to define a varible on any CtElement attribute. E.g. method modifiers or throwables, ...
- `byString(String name)` - all pattern model string attributes whose value is equal to `name` are considered as pattern parameter.This can be used to define full name of the methods and fields, etc.
- `bySubstring(String stringMarker)` - all pattern model string attributes whose value contains whole string or a substring equal to `stringMarker`are pattern parameter. Note: only the `stringMarker` substring of the string value is substituted, other parts of string/element name are kept unchanged.
- `byNamedElement(String name)` - any CtNamedElement identified by it's simple name is a pattern parameter.
- `byReferenceName(String name)` - any CtReference identified by it's simple name is a pattern parameter.

Any parameter of a pattern can be configured like this:

- `setMinOccurence(int)` - defines minimal number of occurences of the value of this parameter during **matching**, which is needed by matcher to accept that value.
  - `setMinOccurence(0)` - defines optional parameter
  - `setMinOccurence(1)` - defines mandatory parameter
  - `setMinOccurence(n)` - defines parameter, whose value must be repeated at least n-times

- `setMaxOccurence(int)` - defines maximal number of occurences of the value of this parameter during **matching**, which is accepted by matcher to accept that value.
- `setMatchingStrategy(Quantifier)` - defines how to matching engine arehave when two pattern nodes may accept the same value.
  - `Quantifier#GREEDY` - Greedy quantifiers are considered "greedy" because they force the matcher to read in, or eat, the entire input prior to attempting the next match. If the next match attempt (the entire input) fails, the matcher backs off the input by one and tries again, repeating the process until a match is found or there are no more elements left to back off from.
  - `Quantifier#RELUCTANT` - The reluctant quantifier takes the opposite approach: It start at the beginning of the input, then reluctantly eat one character at a time looking for a match. The last thing it tries is the entire input.
  - `Quantifier#POSSESSIVE` - The possessive quantifier always eats the entire input string, trying once (and only once) for a match. Unlike the greedy quantifiers, possessive quantifiers never back off, even if doing so would allow the overall match to succeed.
- `setValueType(Class type)` - defines a required type of the value. If defined the pattern matched, will match only values which are assigneable from the provided `type`
- `matchCondition(Class<T> type, Predicate<T> matchCondition)` - defines a `Predicate`, whose method `boolean test(T)`, are called by pattern matcher. Template matcher accepts that value only if `test` returns true for the value. The `setValueType(type)` is called internally too, so match condition assures both a type of value and condition on value.
- `setContainerKind(ContainerKind)` - defines what container are used to store the value.
  - `ContainerKind#SINGLE` - only single value is accepted as a parameter value. It can be e.g. single String or single CtStatement, etc.
  - `ContainerKind#LIST` - The values are always stored as `List`.
  - `ContainerKind#SET` - The values are always stored as `Set`.
  - `ContainerKind#MAP` - The values are always stored as `Map`.

### InlinedStatementConfigurator

It is possible to match inlined code, eg:

```
System.out.println(1);
System.out.println(2);
System.out.println(3);
```

can be matched by

```
for (int i=0; i<n; i++) {
  System.out.println(n);
}
```

One mark code to be matched inlined using method `configureInlineStatements`, which receives a InlinedStatementConfigurator as follows:

```
Pattern t = PatternBuilder.create(...select pattern model...)
    //...configure parameters...
    configureInlineStatements(ls ->
        //...select to be inlined statements...
        //e.g. by variable name:
        ls.byVariableName("intValues")
    ).build();
```

The inlining methods are:

- `inlineIfOrForeachReferringTo(String varName)` - all CtForEach and CtIf statements whose expression references variable named `varName` are understood as inline statements
- `markAsInlined(CtForEach|CtIf)` - provided CtForEach or CtIf statement is understood as inline statement

#### Generator

All patterns can be used for code generation. The idea is that one calls `#generator()` on a pattern object to get a `Generator`. This class contains methods that takes as input a map of string,objects where each string key points to a pattern parameter name and each map value contains the element to be put in place of the pattern parameter.

#### Notes on patterns

The unique feature of Spoon pattern matching is that we are matching on AST trees and not source code text. It means that:

- source code formating is ignored. For example:

```
void m() {}
//matches with
void    m(){
}
```

- comments are ignored. For example:

```
void m() {}
//matches with
/**
 javadoc is ignored
*/
/* was public before */ void m(/*this is ignored too*/) {
    //and line comments are ignored too
}
```

- implicit and explicit elements are considered the same. For example:

```
if (something)
    list = (List<String>) new ArrayList<>(FIELD_COUNT);
//matches with
if (something) {
    OuterType.this.list = (java.util.List<java.lang.String>) new java.util.ArrayList<java.lang
        .String>(Constants.FIELD_COUNT);
}
```

- casts are skipped. For example:

```
f(x);
//matches with
(Object) f(x);
```

# Code Transformation

## Transformation processors

We'll make a first transformation that adds a field to a class and initializes it in the constructor of the current class.

#### Factories and setters to create AST elements

With `Factory` (javadoc), you can get and create all elements of the meta model. For example, if you want to create a class with the name "Tacos", use the factory to create an empty class and fill information on the created element to set its name.

```
CtClass newClass = factory.Core().createClass();
newClass.setSimpleName("Tacos");
```

First, create a new field. To do that, create the type referenced by our field. This type is a `java.util.List` which have a `java.util.Date` as generic type.

```
final CtTypeReference<Date> dateRef = getFactory().Code().createCtTypeReference(Date.class);
final CtTypeReference<List<Date>> listRef = getFactory().Code().createCtTypeReference(List.
    class);
listRef.addActualTypeArgument(dateRef);
```

`dateRef`, a `CtTypeReference` (javadoc), is created by our factory from `Date.class` given by Java. We also created `listRef` which is created by our factory from `List.class` and we add our `dateRef` as actual type argument which represents the generic type of the list.

Now, create the field. A field has a name, a type and private.

```
final CtField<List<Date>> listOfDates = getFactory().Core().<List<Date>>createField();
listOfDates.setSimpleName("dates");
listOfDates.setType(listRef);
listOfDates.addModifier(ModifierKind.PRIVATE);
```

We have created a field named "dates", with a private visibility and typed by our previous type reference, `listRef`, which is `java.util.List<java.util.Date>`.

Second, create the constructor. Before the creation of a `CtConstructor` (javadoc), create all objects necessary for this constructor and set them in the target constructor. The constructor has a parameter typed by the same type of the field previously created and has a body to assign the parameter to the field.

```
final CtCodeSnippetStatement statementInConstructor = getFactory().Code().
    createCodeSnippetStatement("this.dates = dates");

final CtBlock<?> ctBlockOfConstructor = getFactory().Code().createCtBlock(
    statementInConstructor);

final CtParameter<List<Date>> parameter = getFactory().Core().<List<Date>>createParameter();
parameter.setType(listRef);
parameter.setSimpleName("dates");

final CtConstructor constructor = getFactory().Core().createConstructor();
constructor.setBody(ctBlockOfConstructor);
constructor.setParameters(Collections.<CtParameter<?>>singletonList(parameter));
constructor.addModifier(ModifierKind.PUBLIC);
```

*Wow! Wait ... What is `CtCodeSnippetStatement`?*

You can convert any string in a `CtStatement` (javadoc) with `createCodeSnippetStatement(String statement)` or in `CtExpression` (javadoc) with `createCodeSnippetExpression(String expression)`. In our case, we convert `this.dates = dates` in a `CtAssignement` (javadoc) with an assignment and an assigned elements.

With this last example, you have created a statement that you have put in a block. You have created a parameter typed by the same type as the field and you have put all these objects in the constructor.

Finally, apply all transformations in your processor:

```
public class ClassProcessor extends AbstractProcessor<CtClass<?>> {
    @Override
    public void process(CtClass<?> ctClass) {
        // Creates field.
        final CtTypeReference<Date> dateRef = getFactory().Code().createCtTypeReference(Date.
            class);
        final CtTypeReference<List<Date>> listRef = getFactory().Code().createCtTypeReference(
            List.class);
        listRef.addActualTypeArgument(dateRef);
```

```
        final CtField<List<Date>> listOfDates = getFactory().Core().<List<Date>>createField();
        listOfDates.<CtField>setType(listRef);
        listOfDates.<CtField>addModifier(ModifierKind.PRIVATE);
        listOfDates.setSimpleName("dates");

        // Creates constructor.
        final CtCodeSnippetStatement statementInConstructor = getFactory().Code().
            createCodeSnippetStatement("this.dates = dates");
        final CtBlock<?> ctBlockOfConstructor = getFactory().Code().createCtBlock(
            statementInConstructor);
        final CtParameter<List<Date>> parameter = getFactory().Core().<List<Date>>
            createParameter();
        parameter.<CtParameter>setType(listRef);
        parameter.setSimpleName("dates");
        final CtConstructor constructor = getFactory().Core().createConstructor();
        constructor.setBody(ctBlockOfConstructor);
        constructor.setParameters(Collections.<CtParameter<?>>singletonList(parameter));
        constructor.addModifier(ModifierKind.PUBLIC);

        // Apply transformation.
        ctClass.addField(listOfDates);
        ctClass.addConstructor(constructor);
    }
}
```

#### Refactoring transformations

Spoon provides some methods for automated refactoring:.

- Local Variable Refactoring class, renames local variables and includes extra checking to ensure program correctness after renaming,
- Generic Variable Refactoring class, renames any variable type (field, parameter, local), but does not do any extra checking to ensure program correctness.
- `Refactoring` contains helper methods for refactoring, incl. one for automated removal of deprecated methods.

### Examples of Spoon Transformation Usages

We provide examples for learning and teaching Spoon in https://github.com/SpoonLabs/spoon-examples/. Don't hesitate to propose new examples as pull-request!

#### Transformation with API

The NotNullProcessor example adds a not-null check for all method parameters.

The LogProcessor example is an example of Spoon for tracing, it adds a log statement when entering a method.

The MutationProcessor example randomly mutates some parts of the abstract syntax tree for mutation testing.

#### Transformation with Annotations

The Bound example adds runtime checks based on annotations

The Database access example shows how to use annotation processing to add persistence into a POJO, ie. to implement a simple URL with Spoon (also uses templates).

The Nton example introduces a Nton design pattern (extension of singleton but for N instances) into a target class. It inserts static fields, methods, and initializer code into constructors.

The Visitor example implements a visitor pattern by automatically introducing an accept method in a visited type hierarchy.

The Field Access example implements a refactoring that introduces setters and getters for the fields annotated with the Access annotation and that replaces all the direct accesses to these fields by calls to its new getters and setters.

### Transformation with Templates

The RetryTemplate example creates retriable methods in case of exceptions if annotated by `@RetryOnFailure`.

### Transformation with Patterns

TBD, see https://github.com/INRIA/spoon/issues/3140

The Distributed Calculus example creates a fun new language for distributed computing using Java and Spoon.

## Processor for annotations

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` 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.

### Annotation Processing Example

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. The code below shows both the definition of the `NotNull` annotation type, and an example of its use.

```
@Target({ElementType.PARAMETER})
@Retention(RetentionPolicy.SOURCE)
public @interface NotNull{}

class Person{
    public void marry(@NotNull Person so){
        if(!so.isMarried())
            // Marrying logic...
    }
}
```

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.

### The Annotation Processor Interface

In Spoon, the full code model can be used for compile-time annotation processing. To this end, Spoon provides a special kind of processor called `AnnotationProcessor` whose interface is:

```
public interface AnnotationProcessor<A extends Annotation, E extends CtElement> extends
    Processor<E> {
    void process(A annotation, E element);
    boolean inferConsumedAnnotationType();
    Set<Class<? extends A>> getProcessedAnnotationTypes();
    Set<Class<? extends A>> getConsumedAnnotationTypes();
    boolean shoudBeConsumed(CtAnnotation<? extends Annotation> annotation);
}
```

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 four methods (`getProcessedAnnotationTypes`, `getConsumedAnnotationTypes`, `inferConsumedAnnotationTypes` and `shoudBeConsumed`) 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. 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.

```
class NotNullProcessor extends AbstractAnnotationProcessor<NotNull, CtParameter> {
    @Override
    public void process(NotNull anno, CtParameter param){
        CtMethod<?> method = param.getParent(CtMethod.class);
        CtBlock<?> body = method.getBlock();
        CtAssert<?> assertion = constructAssertion(param.getSimpleName());
        body.insertBegin(assertion);
    }
}
```

The `NotNullProcessor` 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. 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. 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.

## Transformation with Templates

Spoon provides developers a way of writing code transformations called **code templates**. Those templates are statically type-checked, in order to ensure statically that the generated code will be correct.

A Spoon template is a regular Java class that taken as input by the Spoon templating engine to perform a transformation. This is summarized in Figure below. 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.

[Overview of Spoon's Templating System]({{ "/images/template-overview.svg" | prepend: site.baseurl }})

### Definition of templates

Class `CheckBoundTemplate` below defines a Spoon template.

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

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` (javadoc). In this case, the template parameter is meant to be an expression (`CtExpression`) that returns a Collection (see constructor, line 3). All meta-model classes, incl. `CtExpression` (javadoc), implement interface `TemplateParameter`. A template parameter has a special method (named `S`, for Substitution) 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 which is described above. Consequently, the template source is well-typed, compiles, but the binary code of the template is thrown away.

### Template Instantiation

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

The code at the end of this page shows how to use the check-bound of template, `CheckBoundTemplate`, presented in the previous section. One first instantiates a template, then one sets the template parameters, and finally, one calls the template engine. In 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.

```
// 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);
```

### Kinds of templating

There are different kinds of templating.

#### Subclassing `StatementTemplate`

Using method `apply()` enables to get a new statement (see example `CheckBoundTemplate` above)

#### Subclassing `BlockTemplate`

Using method `apply()` enables to get a new block.

#### Subclassing `ExpressionTemplate`

Using method `apply()` enables to get a new expression. The core template method must be called `expression` and only contain a return with the expression to be templated.

#### Subclassing `ExtensionTemplate`

Using method `apply()` enables to get a new class where all possible templating in all methods. In addition, the following class level transformations are made:

1) methods and field of the templates are injected in the target class

```
public class ATemplate1 extends ExtensionTemplate {
  int i;
  void foo() {};
}

// inject `i` and `foo` in aCtClass
Substitution.insertAll(aCtClass, new ATemplate1());
```

2) parametrized superinterfaces are injected in the target class

```
class ATemplate2 extends ExtensionTemplate implements Serializable, A, B {
    // interface templates supports TypeReference
    @Parameter
    Class A;
}

Template t = new ATemplate2();
t.A = Comparable.class
Substitution.insertAll(aCtClass, t);
// aCtClass now implements Serializable and Serializable
```

3) method parameters are replaced

```
class ATemplate3 extends ExtensionTemplate {
    public void methodWithTemplatedParameters(Object params) {
        // code
    }

    @Parameter
    public List<CtParameter> params;
```

```
}

Template t = new ATemplate3();
t.params = ...
Substitution.insertAll(aCtClass, t);
// aCtClass  contains methodmethodWithTemplatedParameters with specific parameters
```

### Template parameters

#### AST elements

All meta-model elements can be used as template parameter. There are two ways of defining such a template parameter.

1) Using a subtype of TemplateParameter

The following template uses a block as template parameter. 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.

```
public class TryCatchOutOfBoundTemplate extends BlockTemplate {
        // CTBlock is a subtype of TemplateParameter as most metamodel elements
    CtBlock _body_; // the body to surround

    @Override
    public void block() {
        try {
            _body_.S();
        } catch (OutOfBoundException e) {
            e.printStackTrace();
        }
    }
}
```

One can also type the field directly with `TemplateParameter`:

```
public class TryCatchOutOfBoundTemplate extends BlockTemplate {
    TemplateParameter<Void> _body_; // the body to surround

    @Override
    public void block() {
        try {
            _body_.S();
        } catch (OutOfBoundException e) {
            e.printStackTrace();
        }
    }
}
```

2) Using annotation `@Parameter`

Fields annotated with `@Parameter` are template parameters.

```
@Parameter
CtInvocation invocation;
```

and then all `invocation.S()` will be replaced by the actual invocation.

### Literal template Parameters

For literals, Spoon provides developers with *literal template parameters*. When the parameter is known to be a literal (primitive types, `Class` or a one-dimensional array of these types), a template parameter, annotated with `@Parameter` enables one to have concise template code.

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

String parameters are not working like other primitive type parameters, since we're using String parameters only to rename elements of the code like fields and methods.

```
// with String template parameter, which is used to substitute method name.
@Parameter
String methodName;
...
methodName = "generatedMethod";
...
void methodName() {
    //a body of generated method
}
```

To use a parameter with a type String like other primitive types, use CtLiteral.

```
// with CtLiteral<String> template parameter, which is used to substitute String literal
@Parameter
CtLiteral<String> val;
...
val = factory.Code().createLiteral("Some string");
...
String someMethod() {
    return val.S(); //is substituted as return "Some string";
}
```

or String literal can be optionally generated like this

```
// with CtLiteral<String> template parameter, which is used to substitute String literal
@Parameter
String val;
...
val = "Some string";
...
String someMethod() {
    return "val";   //is substituted as return "Some string";
}
```

### Inlining foreach expressions

Foreach expressions can be inlined. They have to be declared as follows:

```
@Parameter
CtExpression[] intValues;
...
template.intValues = new CtExpression[2];
template.intValues[0] = factory.Code().createLiteral(0);
template.intValues[1] = factory.Code().createLiteral(1);
```

and then,

```
for(Object x : intValues) {
        System.out.println(x);
}
```

is transformed into:

```
{
    java.lang.System.out.println(0);
    java.lang.System.out.println(1);
}
```

## Testing code transformations

Spoon module testing is a Java library that provides a fluent api for writing assertions. Its main goal is to propose an easy way to test Java source code transformation.

This module is directly integrated in the spoon project and can be used as soon as the dependency is specified in your project.

### The Assert class

The Assert class is the entry point for assertion methods for different data types. Each method in this class is a static factory for the type-specific assertion objects. The purpose of this class is to make test code more readable.

All methods in this class are named `assertThat` and take only one argument. For example, if you use the method `assertThat(File)`, you will be able to use the method `isEqualTo(File)` to check the equality between these two files.

```
Assert.assertThat(new File("actual.java")).isEqualTo(new File("expected.java"));
```

Spoon provides a way to test transformations as follows.

```
import static spoon.testing.Assert.assertThat;
...
assertThat('Foo.java').withProcessor(new AProcessor()).isEqualTo('FooTransformed.java');
```

### Assertion Types

There are three types of assertions:

| Assert type | Description |
|---|---|
| FileAssert | Assertions available on a file. |
| CtElementAssert | Assertions available on a `CtElement`. |
| CtPackageAssert | Assertions available between two `CtPackage`. |

### Assertion example

Let's say that you have a processor which change the name of all fields by the name "j".

```
class MyProcessor extends AbstractProcessor<CtField<?>> {
    @Override
    public void process(CtField<?> element) {
        element.setSimpleName("j");
    }
}
```

To check that the transformation is well done when you apply it on a class, see the following example

```
final SpoonAPI spoon = new Launcher();
```

```
spoon.addInputResource("path/of/my/file/Foo.java");
spoon.run();

final CtType<Foo> type = spoon.getFactory().Type().get(Foo.class);
assertThat(type.getField("i")).withProcessor(new MyProcessor()).isEqualTo("public int j;");
```

Note that, method `withProcessor` takes as parameter either with a processor instance, a processor class name, a class object.

# Usage

## Using Spoon as a Java library

### The Launcher class

The Spoon `Launcher` (JavaDoc) is used to create the AST model of a project. It can be as short as:

```
CtClass l = Launcher.parseClass("class A { void m() { System.out.println(\"yeah\");} }");
```

Or with a plain object:

```
Launcher launcher = new Launcher();

// path can be a folder or a file
// addInputResource can be called several times
launcher.addInputResource("<path_to_source>");

launcher.buildModel();

CtModel model = launcher.getModel();
```

### Pretty-printing modes

Spoon has three pretty-printing modes:

**Fully-qualified** Spoon can pretty-print code where all classes and methods are fully-qualified. This is the default behavior on `toString()` on AST elements. This is not readable for humans but is useful when name collisions happen. If `launcher.getEnvironment().getToStringMode()== FULLYQUALIFIED`, the files written on disk are also fully qualified.

**Autoimport** Spoon can pretty-print code where all classes and methods are imported as long as no conflict exists.

```
launcher.getEnvironment().setAutoImports(true);
```

The autoimport mode computes the required imports, add the imports in the pretty-printed files, and writes class names unqualified (w/o package names). This involves changing the field `implicit` of some elements of the model, through a set of `ImportAnalyzer`, most notable `ImportCleaner` and `ImportConflictDetector`. When pretty-printing, Spoon reformats the code according to its own formatting rules that can be configured by providing a custom `TokenWriter`.

**Sniper mode** The sniper mode enables to rewrite only the transformed AST elements, so that the rest of the code is printed identically to the origin version. This is useful to get small diffs after automated refactoring.

```
launcher.getEnvironment().setPrettyPrinterCreator(() -> {
   return new SniperJavaPrettyPrinter(launcher.getEnvironment());
  }
);
```

**Comments** In addition, depending on the value of `Environment#getCommentEnabled`, the comments are removed or kept from the Java files saved to disk (call `Environment#setCommentEnabled(true)` to keep comments).

### The MavenLauncher class

The Spoon `MavenLauncher` (JavaDoc) is used to create the AST model of a Maven project. It automatically infers the list of source folders and the dependencies from the `pom.xml` file. This Launcher handles multi-module Maven projects.

```
// the second parameter can be APP_SOURCE / TEST_SOURCE / ALL_SOURCE
MavenLauncher launcher = new MavenLauncher("<path_to_maven_project>", MavenLauncher.
    SOURCE_TYPE.APP_SOURCE);
launcher.buildModel();
CtModel model = launcher.getModel();

// list all packages of the model
for(CtPackage p : model.getAllPackages()) {
  System.out.println("package: " + p.getQualifiedName());
}
// list all classes of the model
for(CtType<?> s : model.getAllTypes()) {
  System.out.println("class: " + s.getQualifiedName());
}
```

Note that by default, MavenLauncher relies on an existing local maven binary to build the project's classpath. But a constructor allowing the user to skip this step and to provide a custom classpath is available.

```
MavenLauncher launcher = new MavenLauncher("<path_to_maven_project>",
        MavenLauncher.SOURCE_TYPE.APP_SOURCE,
        new String[] {
            "/home/user/.m2/repository/org/my/jar/1.0/org-my-jar-1.0.jar"
        }
    );
launcher.buildModel();
CtModel model = launcher.getModel();
```

To avoid invoking maven over and over to build a classpath that has not changed, it is stored in a file `spoon.classpath.tmp` (or depending on the scope `spoon.classpath-app.tmp` or `spoon.classpath-test.tmp`) in the same folder as the `pom.xml`. This classpath will be refreshed is the file is deleted or if it has not been modified since 1h.

### Analyzing bytecode with JarLauncher

There are two ways to analyze bytecode with spoon:

- Bytecode resources can be added in the classpath, (some information will be extracted through reflection)
- A decompiler may be used, and then, the analyzes will be performed on the decompiled sources.

The Spoon `JarLauncher` (JavaDoc) is used to create the AST model from a jar. It automatically decompiles class files contained in the jar and analyzes them. If a pom file corresponding to the jar is provided, it will be used to build the classpath containing all dependencies.

```
//More constructors are available, check the JavaDOc for more information.
JarLauncher launcher = JarLauncher("<path_to_jar>", "<path_to_output_src_dir>", "<path_to_pom>
    ");
launcher.buildModel();
CtModel model = launcher.getModel();
```

Note that the default decompiler CFR can be changed by providing an instance implementing `spoon.decompiler.Decompiler` as a parameter.

```
JarLauncher launcher = new JarLauncher("<path_to_jar>", "<path_to_output_src_dir>", "<
    path_to_pom>",
    new Decompiler() {
        @Override
        public void decompile(String inputPath, String outputPath, String[] classpath) {
            //Custom decompiler call
        }
    }
);
```

Spoon provides two out of the shelf decompilers, CFR by default, and Fernflower. You can use the later like this:

```
JarLauncher launcher = new JarLauncher(
        "<path_to_jar>",
        "<path_to_output_src_dir>",
        "<path_to_pom>",
        new FernflowerDecompiler(new File("<path_to_output_src_dir>/src/main/java"))
);
```

Optionally, the classic launcher can be used with `DecompiledResource` like this:

```
Launcher launcher = new Launcher();
launcher.addInputResource(
    new DecompiledResource(baseDir.getAbsolutePath(), new String[]{}, new CFRDecompiler(),
        pathToDecompiledRoot.getPath())
);
```

**Warning** The `JarLauncher` feature (and all features relying on decompilation) are not included in `spoon-core` but in `spoon-decompiler`. If you want to use them you should declare a dependency to `spoon-decompiler`.

#### Resolution of elements and classpath

Spoon analyzes source code. However, this source code may refer to libraries (as a field, parameter, or method return type). There are two cases:

- Full classpath: all dependencies are in the JVM classpath or are given to the Laucher with `launcher.getEnvironment().setSourceClasspath("<classpath_project>");` (optional)
- No classpath: some dependencies are unknown and `launcher.getEnvironment().setNoClasspath(true)` is set.

This has a direct impact on Spoon references. When you're consider a reference object (say, a TypeReference), there are three cases:

- Case 1 (code available as source code): the reference points to a code element for which the source code is present. In this case, reference.getDeclaration() returns this code element (e.g. TypeReference.getDeclaration returns the CtType representing the given java file). reference.getTypeDeclaration() is identical to reference.getDeclaration().
- Case 2 (code available as binary in the classpath): the reference points to a code element for which the source code is NOT present, but for which the binary class is in the classpath (either the JVM classpath or the –source-classpath argument). In this case, reference.getDeclaration() returns null and reference.getTypeDeclaration returns a partial CtType built using runtime reflection. Those objects built using runtime reflection are called shadow objects; and you can identify them with method isShadow. (This also holds for getFieldDeclaration and getExecutableDeclaration)
- Case 3 (code not available, aka noclasspath): the reference points to a code element for which the source code is NOT present, but for which the binary class is NOT in the classpath. This is called in Spoon the noclasspath mode. In this case, both reference.getDeclaration() and reference.getTypeDeclaration() return null. (This also holds for getFieldDeclaration and getExecutableDeclaration)

## Declaring the dependency to Spoon

### Maven

```
<dependency>
    <groupId>fr.inria.gforge.spoon</groupId>
    <artifactId>spoon-core</artifactId>
    <version>{{site.spoon_release}}</version>
</dependency>
```

### Gradle

```
compile 'fr.inria.gforge.spoon:spoon-core:{{site.spoon_release}}'
```

## Advanced launchers

### Incremental Launcher

`IncrementalLauncher` (JavaDoc) allows cache AST and compiled classes. Any spoon analysis can then be restarted from where it stopped instead of restarting from scratch.

```
final File cache = new File("<path_to_cache>");
Set<File> inputResources = Collections.singleton(new File("<path_to_sources>"));
Set<String> sourceClasspath = Collections.emptySet(); // Empty classpath

//Start build from cache
IncrementalLauncher launcher = new IncrementalLauncher(inputResources, sourceClasspath, cache)
    ;

if (launcher.changesPresent()) {
    System.out.println("There are changes since last save to cache.");
}

CtModel newModel = launcher.buildModel();
//Model is now up to date

launcher.saveCache();
//Cache is now up to date
```

### Fluent LauncherAPI

`FluentLauncher` (JavaDoc) allows simple, fluent launcher usage with setting most options directly.

For the classic launcher it's simply:

```
CtModel model = new FluentLauncher()
                .inputResource("<path_to_sources>")
                .noClasspath(true)
                .outputDirectory("<path_to_outputdir>")
                .processor(....)
                .buildModel();
```

If you want to use other launchers like the MavenLauncher:

```
MavenLauncher launcher = new MavenLauncher(....);
CtModel model = new FluentLauncher(launcher)
                .processor(....)
                .encoding(...)
                .buildModel();
```

## Using Spoon from the Command Line

To run Spoon in command line, you first have to download the corresponding jar file on Maven Central (take the version with all dependencies).

When you have downloaded the desired version of Spoon, you can directly use it. To know how you use the jar file, launch it with `--help` argument. You see the output at the end of this page for the current release version of Spoon.

The basic usage of Spoon consists in defining the original source location and the list of compiled processors to be used.

```
$ java -classpath /path/to/binary/of/your/processor.jar:spoon-core-{{site.spoon_release}}-with
    -dependencies.jar spoon.Launcher -i /path/to/src/of/your/project -p fr.inria.gforge.spoon.
    processors.CatchProcessor
```

If you plan to repeatedly run Spoon from the command line, it may be a good idea to combine all of your commands into a single bash script. An example of this can be found here.

Note that when you use Spoon in command line, you manually handle the classpath. In particular, if the to-be-transformed source files depend on libraries, specify them with the `--source-classpath` flag.

```
Options :

  [-h|--help]

  [--tabs]
        Use tabulations instead of spaces in the generated code (use spaces by
        default).

  [--tabsize <tabsize>]
        Define tabulation size. (default: 4)

  [--level <level>]
        Level of the ouput messages about what spoon is doing. (default: OFF)

  [--with-imports]
        Enable imports in generated files.

  [--compliance <compliance>]
        Java source code compliance level (1,2,3,4,5, 6, 7 or 8). (default: 8)

  [--encoding <encoding>]
        Forces the compiler to use a specific encoding (UTF-8, UTF-16, ...).
        (default: UTF-8)

  [(-i|--input) <input>]
        List of path to sources files.

  [(-p|--processors) <processors>]
        List of processor qualified name to be used.

  [(-t|--template) <template>]
        List of path to templates java files.

  [(-o|--output) <output>]
        Specify where to place generated java files. (default: spooned)

  [--source-classpath <source-classpath>]
        An optional classpath to be passed to the internal Java compiler when
        building or compiling the input sources.

  [--template-classpath <template-classpath>]
        An optional classpath to be passed to the internal Java compiler when
        building the template sources.

  [(-d|--destination) <destination>]
        An optional destination directory for the generated class files.
        (default: spooned-classes)

 --cpmode <cpmode>
        Classpath mode to use in Spoon: NOCLASSPATH; FULLCLASSPATH (default:
        NOCLASSPATH)
```

```
[--output-type <output-type>]
      States how to print the processed source code:
      nooutput|classes|compilationunits (default: classes)

[--compile]
      Enable compilation and output class files.

[--lines]
      Set Spoon to try to preserve the original line numbers when generating
      the source code (may lead to human-unfriendly formatting).

[-g|--gui]
      Show spoon model after processing

[-r|--no-copy-resources]
      Disable the copy of resources from source to destination folder.

[-j|--generate-javadoc]
      Enable the generation of the javadoc. Deprecated, use enable-comments
      argument.

[-c|--enable-comments]
      Adds all code comments in the Spoon AST (Javadoc, line-based comments),
      rewrites them when pretty-printing.

[(-f|--generate-files) <generate-files>]
      Only generate the given fully qualified java classes (separated by ':'
      if multiple are given).

[-a|--disable-model-self-checks]
      Disables checks made on the AST (hashcode violation, method signature
      violation and parent violation). Default: false.
```

### Using Spoon from Maven

The main documentation of the Spoon Maven plugin is in the README of https://github.com/SpoonLabs/spoon-maven-plugin.

Pull requests on this documentation should be done on https://github.com/SpoonLabs/spoon-maven-plugin and not here.

### Using Spoon from Gradle

The main documentation of the Spoon Gradle plugin is in the README of https://github.com/SpoonLabs/spoon-gradle-plugin.

Pull requests on this documentation should be done on https://github.com/SpoonLabs/spoon-gradle-plugin and not here.

## Spoon Meta model

### Structural elements

A programming language can have different meta models. An abstract syntax tree (AST) or model, is an instance of a meta model. Each meta model – and consequently each AST – is more or less appropriate depending on the task at hand. For instance, the Java meta model of Sun's compiler (javac) has been designed and optimized for compilation to bytecode, while, the main purpose of the Java meta model 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 meta model 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 meta model 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 meta model can be split in three parts.

- The structural part contains the declarations of the program elements, such as interface, class, variable, method, annotation, and enum declarations.
- The code part 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 the figure, all elements inherit from `CtElement` (javadoc) 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" which means "compile-time".

As of Spoon 6.1.0, Spoon metamodel contains CtModule element to represent a module in Java 9, and CtModuleDirective to represent the different directives of the module.

:warning: The root of the model is then no longer an unnamed package, but an unnamed module.

[Structural part of the Spoon Java 8 metamodel]({{ "/images/structural-elements.png" | prepend: site.baseurl }})

## Code elements

Figure at the end of this page shows the meta model for Java executable code. There are two main kinds of code elements. First, statements `CtStatement` (javadoc) are untyped top-level instructions that can be used directly in a block of code. Second, expressions `CtExpression` (javadoc) are used inside the statements. For instance, a `CtLoop` (javadoc) (which is a statement) points to `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` (javadoc) inheriting from both interfaces `CtStatement` and `CtExpression`. The generic type of `CtExpression` is used to add static type-checking when transforming programs.

[Code part of the Spoon Java 8 metamodel]({{ "/images/code-elements.png" | prepend: site.baseurl }})

### CtArrayRead

(javadoc)

```
int[] array = new int[10];
System.out.println(
array[0] // <-- array read
);
```

### CtArrayWrite

(javadoc)

```
Object[] array = new Object[10];
// array write
array[0] = "new value";
```

### CtAssert

(javadoc)

```
assert 1+1==2
```

### CtAssignment

(javadoc)

```
    int x;
    x = 4; // <-- an assignment
```

### CtBinaryOperator

(javadoc)

```
    // 3+4 is the binary expression
    int x = 3 + 4;
```

### CtBlock

(javadoc)

```
{ // <-- block start
 System.out.println("foo");
}
```

### CtBreak

(javadoc)

```
    for(int i=0; i<10; i++) {
        if (i>3) {
                break; // <-- break statement
        }
    }
```

### CtCase

(javadoc)

```
int x = 0;
switch(x) {
    case 1: // <-- case statement
      System.out.println("foo");
}
```

### CtConditional

(javadoc)

```
    System.out.println(
       1==0 ? "foo" : "bar" // <-- ternary conditional
    );
```

### CtConstructorCall

(javadoc)

```
new Object();
```

### CtContinue

(javadoc)

```
for(int i=0; i<10; i++) {
    if (i>3) {
            continue; // <-- continue statement
    }
}
```

### CtDo

(javadoc)

```
int x = 0;
do {
    x=x+1;
} while (x<10);
```

### CtExecutableReferenceExpression

(javadoc)

```
java.util.function.Supplier p =
  Object::new;
```

### CtFieldRead

(javadoc)

```
class Foo { int field; }
Foo x = new Foo();
System.out.println(x.field);
```

### CtFieldWrite

(javadoc)

```
class Foo { int field; }
Foo x = new Foo();
x.field = 0;
```

### CtFor

(javadoc)

```
// a for statement
for(int i=0; i<10; i++) {
    System.out.println("foo");
}
```

### CtForEach

(javadoc)

```
java.util.List l = new java.util.ArrayList();
for(Object o : l) { // <-- foreach loop
    System.out.println(o);
}
```

### CtIf

(javadoc)

```
if (1==0) {
    System.out.println("foo");
} else {
    System.out.println("bar");
}
```

### CtInvocation

(javadoc)

```
// invocation of method println
// the target is "System.out"
System.out.println("foo");
```

### CtJavaDoc

(javadoc)

```
/**
 * Description
 * @tag a tag in the javadoc
*/
```

### CtLambda

(javadoc)

```
java.util.List l = new java.util.ArrayList();
l.stream().map(
  x -> { return x.toString(); } // a lambda
);
```

### CtLiteral

(javadoc)

```
int x = 4; // 4 is a literal
```

### CtLocalVariable

(javadoc)

```
// defines a local variable x
int x = 0;


// local variable in Java 10
var x = 0;
```

### CtNewArray

(javadoc)

```
// inline creation of array content
int[] x = new int[] { 0, 1, 42}
```

### CtNewClass

(javadoc)

```
// an anonymous class creation
Runnable r = new Runnable() {
     @Override
     public void run() {
       System.out.println("foo");
     }
};
```

### CtOperatorAssignment

(javadoc)

```
int x = 0;
x *= 3; // <-- a CtOperatorAssignment
```

### CtReturn

(javadoc)

```
Runnable r = new Runnable() {
     @Override
     public void run() {
       return; // <-- CtReturn statement
     }
};
```

### CtSuperAccess

(javadoc)

```
    class Foo { int foo() { return 42;}};
    class Bar extends Foo {
    int foo() {
      return super.foo(); // <-- access to super
    }
    };
```

### CtSwitch

(javadoc)

```
int x = 0;
switch(x) { // <-- switch statement
    case 1:
      System.out.println("foo");
}
```

### CtSwitchExpression

(javadoc)

```
int i = 0;
int x = switch(i) { // <-- switch expression
    case 1 -> 10;
    case 2 -> 20;
    default -> 30;
};
```

### CtSynchronized

(javadoc)

```
   java.util.List l = new java.util.ArrayList();
   synchronized(l) {
        System.out.println("foo");
   }
```

### CtThisAccess

(javadoc)

```
    class Foo {
    int value = 42;
    int foo() {
      return this.value; // <-- access to this
    }
    };
```

### CtThrow

(javadoc)

```
throw new RuntimeException("oops")
```

### CtTry

(javadoc)

```
try {
    System.out.println("foo");
} catch (Exception ignore) {}
```

### CtTryWithResource

(javadoc)

```
// br is the resource
try (java.io.BufferedReader br = new java.io.BufferedReader(new java.io.FileReader("/foo"))
    ) {
  br.readLine();
}
```

### CtTypeAccess

(javadoc)

```
// access to static field
java.io.PrintStream ps = System.out;


// call to static method
Class.forName("Foo")


// method reference
java.util.function.Supplier p =
  Object::new;


// instanceof test
boolean x = new Object() instanceof Integer // Integer is represented as an access to type
     Integer


// fake field "class"
Class x = Number.class
```

### CtUnaryOperator

(javadoc)

```
int x=3;
--x; // <-- unary --
```

### CtVariableRead

(javadoc)

```
String variable = "";
System.out.println(
  variable // <-- a variable read
);
```

### CtVariableWrite

(javadoc)

```
String variable = "";
variable = "new value"; // variable write


String variable = "";
variable += "";
```

### CtWhile

(javadoc)

```
int x = 0;
while (x!=10) {
    x=x+1;
};
```

### CtYieldStatement

(javadoc)

```
int x = 0;
x = switch ("foo") {
    default -> {
                x=x+1;
                yield x; //<--- yield statement
                }
};


int x = 0;
x = switch ("foo") {
    default -> 4; //<---  implicit yield statement
};
```

### CtAnnotation

(javadoc)

```
// statement annotated by annotation @SuppressWarnings
@SuppressWarnings("unchecked")
java.util.List<?> x = new java.util.ArrayList<>()
```

#### CtClass

(javadoc)

```
// a class definition
class Foo {
   int x;
}
```

## References

The reference part of the meta model expresses the fact that program references elements that are not necessarily reified into the meta model (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 meta model 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.

{{site.data.alerts.note}} From Spoon 5.0.0, CtReference is a subclass of CtElement. {{site.data.alerts.end}}

[References of the Spoon Java 8 metamodel]({{ "/images/references-elements.png" | prepend: site.baseurl }})

#### How are references resolved?

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.

#### Do targets of references have to exist before you can reference them?

Since the references are weak, the targets of references do not have to exist before one references them.

#### How does this limit transforming code?

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()` (javadoc).

## Creating AST elements

#### Create elements with factories

When you design and implement transformations, with processors or templates, you need to create new elements, fill their data and add them in the AST built by Spoon.

To do that, use `Factory` (javadoc). `Factory` is the entry point for all factories of Spoon. Each factory have a goal specific and help you in the creation of a new AST.

- `CoreFactory` (javadoc) allows the creation of any element in the meta model. To set up the objects, there are setters to initialize the object.
- `CodeFactory` (javadoc) contains utility methods to create code elements and asks minimal information to create a valid object.
- `PackageFactory` (javadoc) contains utility methods to create and get package reference.
- `TypeFactory` (javadoc) contains utility methods with a link to `CtType` (javadoc). You can get any type from its fully qualified name or a .class invocation
  and create all typed references like `CtTypeReference` (javadoc).
- `ClassFactory` (javadoc) is a sub class of `TypeFactory` but specialized for `CtClass` (javadoc).
- `EnumFactory` (javadoc) is a sub class of `TypeFactory` but specialized for `CtEnum` (javadoc).
- `InterfaceFactory` (javadoc) is a sub class of `TypeFactory` but specialized for `CtInterface` (javadoc).

- `ExecutableFactory` (javadoc) contains utility methods with a link to `CtExecutable` (javadoc). You can create executable objects and their parameters.
- `ConstructorFactory` (javadoc) is a sub class of `ExecutableFactory` but specialized for `CtConstructor` (javadoc).
- `MethodFactory` (javadoc) is a sub class of `ExecutableFactory` but specialized for `CtMethod` (javadoc).
- `FieldFactory` (javadoc) contains utility methods to create a valid field or a field reference.
- `AnnotationFactory` (javadoc) contains utility methods to annotate any elements or create a new one.

All these factories contribute to facilitate the creation of elements. When you have created an element from a factory, set it in an existing element to build a new AST.

### Generating Spoon code for an element

With `SpoonifierVisitor`, it is possible to visit an existing Spoon AST to generate calls to the factory that recreates the same AST.

Example:

```
SpoonifierVisitor v = new SpoonifierVisitor(true);
Launcher.parseClass("class A { public String sayHello() { return \"Hello World!\";}}")
        .getMethodsByName("sayHello")
        .get(0)
        .accept(v);
System.out.println(b.getResult());
```

will print:

```
CtMethod ctMethod0 = factory.createMethod();
ctMethod0.setSimpleName("sayHello");
Set<ModifierKind> ctMethod0Modifiers = new HashSet<>();
ctMethod0Modifiers.add(ModifierKind.PUBLIC);
ctMethod0.setModifiers(ctMethod0Modifiers);
    CtTypeReference ctTypeReference0 = factory.createTypeReference();
    ctTypeReference0.setSimpleName("String");
    ctMethod0.setValueByRole(CtRole.TYPE, ctTypeReference0);
        CtPackageReference ctPackageReference0 = factory.createPackageReference();
        ctPackageReference0.setSimpleName("java.lang");
        ctPackageReference0.setImplicit(true);
        ctTypeReference0.setValueByRole(CtRole.PACKAGE_REF, ctPackageReference0);
    CtBlock ctBlock0 = factory.createBlock();
    ctMethod0.setValueByRole(CtRole.BODY, ctBlock0);
        CtReturn ctReturn0 = factory.createReturn();
        List ctBlock0Statements = new ArrayList();
        ctBlock0Statements.add(ctReturn0);
            CtLiteral ctLiteral0 = factory.createLiteral();
            ctLiteral0.setValue("Hello World!");
            ctReturn0.setValueByRole(CtRole.EXPRESSION, ctLiteral0);
                CtTypeReference ctTypeReference1 = factory.createTypeReference();
                ctTypeReference1.setSimpleName("String");
                ctLiteral0.setValueByRole(CtRole.TYPE, ctTypeReference1);
                    CtPackageReference ctPackageReference1 = factory.
                        createPackageReference();
                    ctPackageReference1.setSimpleName("java.lang");
                    ctTypeReference1.setValueByRole(CtRole.PACKAGE_REF,
                        ctPackageReference1);
    ctBlock0.setValueByRole(CtRole.STATEMENT, ctBlock0Statements);
```

## Comments and position

In Spoon there are four different kinds of comments:

- File comments (comment at the begin of the file, generally licence) `CtComment.CommentType.FILE`
- Line comments (from // to end line) `CtComment.CommentType.INLINE`
- Block comments (from /* to */) `CtComment.CommentType.BLOCK`
- Javadoc comments (from /** to */) `CtComment.CommentType.JAVADOC`

The comments are represented in Spoon with a `CtComment` class (javadoc). This class exposes an API to get the content `CtComment.getContent()`, the type `CtComment.getCommentType()` and the position `CtComment.getPosition()` of an comment.

We also try to understand to which element they are attached. We use some simple heuristics that work well in nominal cases but cannot address all specific cases. You can retrieve the comments of each `CtElement` via the API `CtElement.getComments()` which returns a `List<CtComment>`.

The parsing of the comments can be enabled in the Environment via the option `Environment.setCommentEnabled(boolean)` or the command line argument `--enable-comments` (or `-c`).

### Javadoc Comments

The Javadoc comments are also available via the API `CtElement.getDocComment()` but this API returns directly the content of the Javadoc as `String`.

### Comment Attribution

- Each element can have multiple comments
- Comments in the same line of a statement are attached to the statement
- Comments which are alone in one line (or more than one line) are associated to the first element following them
- Comments cannot be associated to other comments
- Comments at the end of a block are considered as orphan comments
- Comments before a class definition are attached to the class

Class comment

```
// class comment
class A {
  // class comment
}
```

Statement comment

```
// Statement comment
int a; // Statement comment
```

Orphan comment

```
try {

} exception (Exception e) {
  // Orphan comment
}
```

Multiple line comment

```
// Statement comment 1
// Statement comment 2
// Statement comment 3
int a;
```

### Processing Comments

You can process comments like every `CtElement`.

```
public class CtCommentProcessor extends AbstractProcessor<CtComment> {

    @Override
    public boolean isToBeProcessed(CtComment candidate) {
        // only process Javadoc
        if (candidatate.getCommentType() == CtComment.CommentType.JAVADOC) {
            return true;
        }
        return false;
    }

    @Override
    public void process(CtComment ctComment) {
        // process the ctComment
    }
}
```

### Source Positions

`SourcePosition` (javadoc) defines the position of the `CtElement` (javadoc) in the original source file. SourcePosition is extended by three specialized positions:

- `DeclarationSourcePosition` (javadoc)
- `BodyHolderSourcePosition` (javadoc).

These three specializations are used to define the position of specific CtElement. For example DeclarationSourcePosition is used to define the position of all declarations (variable, type, method, …). This provides an easy access to the position of the modifiers and the name. The BodyHolderSourcePosition is used to declare the position of all elements that have a body.