Contents

1 Eclipse JDT - Abstract Syntax Tree (AST)

1.1 Introduction

The JDT provide APIs to access and manipulate Java source code via:

1. the Java Model ;
2. the AST.

1.2 The Java Model

1.2.1 Introduction

1. **principle**:
    - each Java project is internally represented via a lightweight and fault-tolerant model.
    - the model does not contain as many information as the AST but is fast to create.

2. **plugin**: `org.eclipse.jdt.core`.
3. **representation**: a tree structure, visualized in the "Package Explorer" view, and described in the table below

Project Element	Java Model Element	Description
Java project	`IJavaProject`	the Java Model root element designating a Java project and containing `IPackageFragmentRoot` as child nodes
src/bin folders or external libraries	`IPackageFragmentRoot`	source or binary files designating folders or libraries (zip/jar file) of a Java project
Java package	`IPackageFragment`	each package is a child node of `IPackageFragmentRoot`, whether a sub-package or a root package, and contains `ICompilationUnits` or `IClassFiles` depending on whether the `IPackageFragmentRoot` is a src or bin folder
Java source file	`ICompilationUnit`	each source file is a child node of a parent package
importations, types, fields, methods, initializers	`IImportDeclaration`, `IType`, `IField`, `IMethod`, `IInitializer`	importations, types, fields, methods and initializations, children of `ICompilationUnit`

1.2.2 Getting compilation units from a Java project programmatically

```
// getting the root workspace
IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();

// getting the project "someJavaProject" from the root workspace
IProject project = root.getProject("someJavaProject");
```

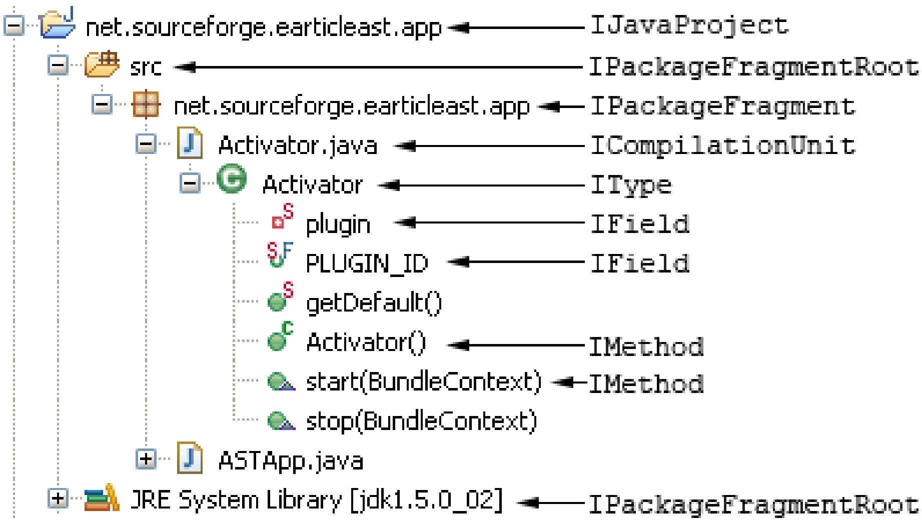


Figure 1: Example of a project's Java model

```
// opening the java project
project.open(null /*IProgressMonitor*/);

// getting the java project handle
IJavaProject javaProject = JavaCore.create(project);

// getting a type in the java project
IType lwType = javaProject.findType("some.package.somewhere.Type");

// getting the compilation unit corresponding to the type
ICompilationUnit lwCompilationUnit = lwType.getCompilationUnit();
```

1.3 AST

1.3.1 Introduction

1. **definition**:
   - a detailed tree representation of the Java source code, defining an API to modify, create, read and delete source code.
   - created based on a `ICompilationUnit` from the Java Model.
2. **utility**:
   - the base framework for many powerful tools of the Eclipse IDE (refactoring, quick fix, quick assist).

- more convenient to analyze and modify source code programmatically than text-based source.

3. **workflow**:
   - provide some Java code source to parse: a Java file in the project or a `char[]` that contains Java source code ;
   - parse code source via `org.eclipse.jdt.core.dom.ASTParser`, returning an AST, and possibly including additional computed information called “bindings” ;
   - manipulate the obtained AST to modify the source code by:
     1. directly modifying the AST ;
     2. noting the modifications in a separate protocol, handled by an instance of `ASTRewrite`.
   - write changes back into the source code from the AST via the `IDocument` interface, a wrapper for the source code.

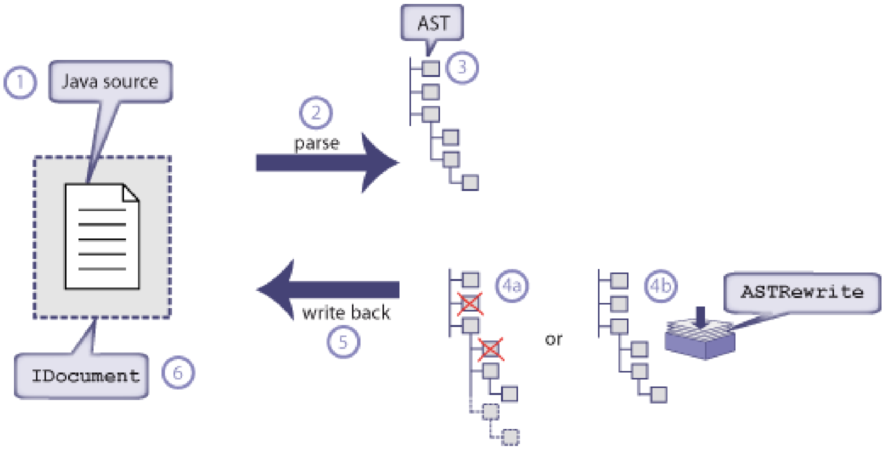


Figure 2: AST Workflow

4. **visualisation**: the AST Viewer plugin, allowing to display the source code in the editor in a tree in the AST Viewer View.

1.3.2 Classes

1. **package**: `org.eclipse.jdt.core.dom` in the `org.eclipse.jdt.core` plugin.
2. `ASTNode`: abstract superclass of all AST nodes, specialized for every element of the Java programming language.
3. `MethodDeclaration`: method declarations.
4. `VariableDeclarationFragment`: variable declarations.
5. `SimpleName`: any string that is not a Java keyword, `true`, `false` or `null`.

1.3.2.1 ASTParser

1. **definition**: a **class** defining an **AST parser**.
2. **methods**:

```
/*
* returns an AST parser according to a Java Language Specification,
* level "level"
*
* level:
* 1. ASTParser.JLS3: third edition of the JLS,
including all new syntax additions (up to Java 5)
*/
public static ASTParser newParser(int level);

/*
* specifies what kind of input the parser will parse
*
* kind:
* 1. ASTParser.K_COMPILATION_UNIT: input = entire java source file
(ICompilationUnit) or a char[] containing the entire source code
* 2. ASTParser.K_EXPRESSION: input = char[] containing a single java expression
like new String() or 4+6 or i
* 3. ASTParser.K_STATEMENTS: input = char[] containing a sequence of java
statements like new String() or synchronized(this){...}
* 4. ASTParser.K_CLASS_BODY_DECLARATIONS: input = char[] containing
elements of a java class like method declarations, field declarations,
static blocks, etc.
*/
public void setKind(int kind);

/*
* sets the source file "source" as the source to parse
*/
public void setSource(ICompilationUnit source);

/*
* sets the char array "source" as the source to parse
*/
public void setSource(char[] source);

/*
* the AST parser should provide binding information to the created AST nodes
* or not
*/
public void setResolveBindings(boolean enabled);
```

```
/*
* the AST parser should provide the error recovery service or not
*/
public void setStatementsRecovery(boolean enabled);

/*
* creates an AST, possibly using a monitor
* to report progress or request cancellation
* or not (monitor = null)
*/
public ASTNode createAST(IProgressMonitor monitor);
```

1.3.3 Finding an AST Node

1. **problem**: scanning all levels of an AST is possible to find an AST node, but not convenient
2. **solution**: using an instance of `ASTVisitor`.

1.3.3.1 `ASTVisitor`

1. **definition**: an **abstract class** implementing the **visitor pattern**, to **visit AST nodes**, to be specialized by all concrete visitors of an AST.
2. **methods**:

```
/*
* visits the given type-specific AST node "_concreteNode_"
*
* returns true if the children of this _concreteNode_ should be visited
* false otherwise
*/
public boolean visit(_concreteNode_);

/*
* ends visit for the given type-specific AST node "_concreteNode_"
*/
public void endVisit(_concreteNode_);

/*
* visits the given AST node "node" prior to the type-specific visit
* (i.e. before invoking visit(node))
*/
public void preVisit(ASTNode node);

/*
* visits the given AST node "node" following the type-specific visit
* (i.e. after invoking endVisit(node))
*/
public void postVisit(ASTNode node);
```

3. **workflow**:
   - `preVisit(node);`
   - `visit(concreteNode);`
   - **children** of `concreteNode` are **recursively processed** if `visit()` returns `true`;
   - `endVisit(concreteNode);`
   - `postVisit(node);`
4. **example**:

```
/*
*a LocalVariableDetector visitor, subclass of ASTVisitor, used to collect
all variable declarations of a compilation unit, implementing visit()
*/
@Override
public boolean visit(VariableDeclarationStatement node){
  for (Iterator iter = node.fragments().iterator(); iter.hasNext();){
    VariableDeclarationFragment fragment =
      (VariableDeclarationFragment) iter.next();
    // store these fragments somewhere
  }

  return false; // prevent SimpleName to be interpreted as a reference
}
```

1.3.4 Structural properties

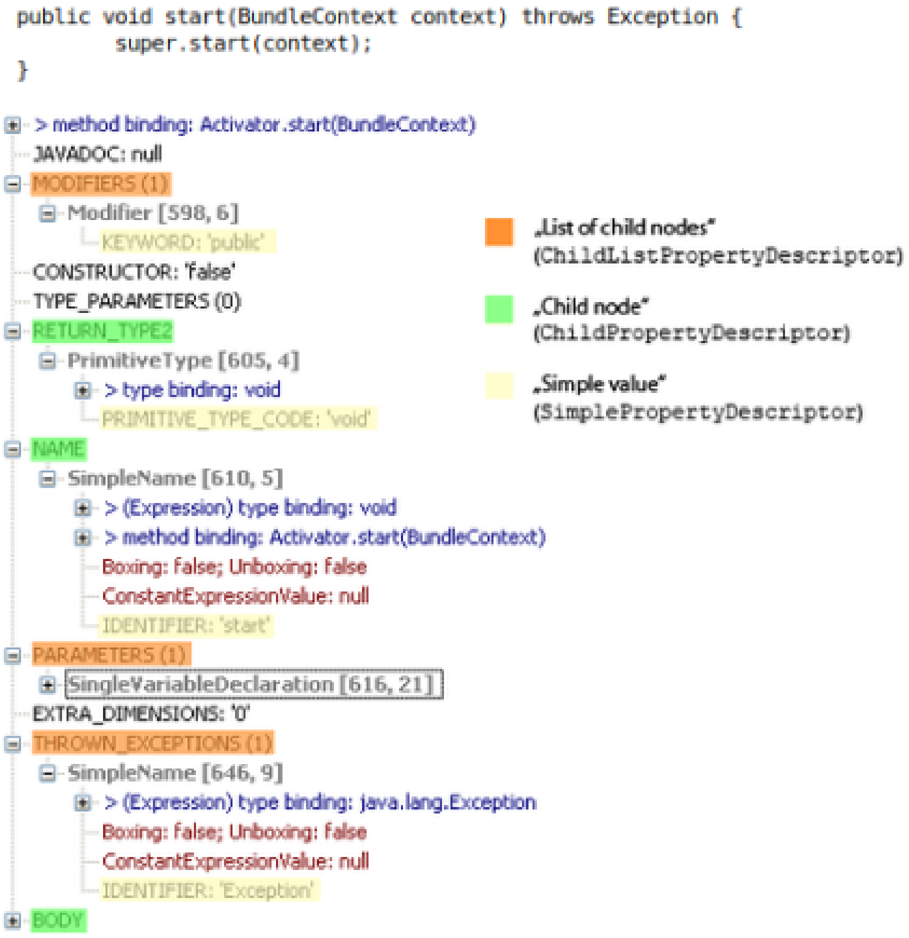
1. **definition**: the information of an `ASTNode`.
2. **example**: `MethodDeclaration` will contain information about the *name, return type, parameters, etc.* of a **method declaration**.

1.3.4.1 Structural property descriptors

1.3.4.1.1 Introduction

1. **definition**: structural property descriptors are descriptors for directly accessing the values of structural properties of any AST node.
2. **example**: `MethodDeclaration.NAME_PROPERTY`;

1.3.4.1.2 `StructuralPropertyDescriptor`



```
public void start(BundleContext context) throws Exception {
        super.start(context);
}
```

```
> method binding: Activator.start(BundleContext)
JAVADOC: null
MODIFIERS (1)
    Modifier [598, 6]
        KEYWORD: 'public'
CONSTRUCTOR: 'false'
TYPE_PARAMETERS (0)
RETURN_TYPE2
    PrimitiveType [605, 4]
        > type binding: void
        PRIMITIVE_TYPE_CODE: 'void'
NAME
    SimpleName [610, 5]
        > (Expression) type binding: void
        > method binding: Activator.start(BundleContext)
        Boxing: false; Unboxing: false
        ConstantExpressionValue: null
        IDENTIFIER: 'start'
PARAMETERS (1)
    SingleVariableDeclaration [616, 21]
EXTRA_DIMENSIONS: '0'
THROWN_EXCEPTIONS (1)
    SimpleName [646, 9]
        > (Expression) type binding: java.lang.Exception
        Boxing: false; Unboxing: false
        ConstantExpressionValue: null
        IDENTIFIER: 'Exception'
BODY
```

🟧 „List of child nodes" (ChildListPropertyDescriptor)

🟩 „Child node" (ChildPropertyDescriptor)

🟨 „Simple value" (SimplePropertyDescriptor)

Figure 3: Example of a MethodDeclaration structural properties

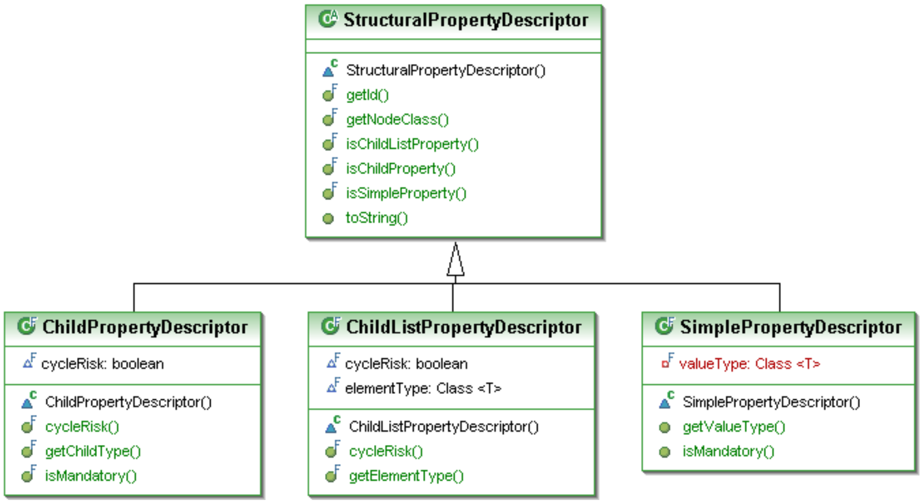


Figure 4: `StructuralPropertyDescriptor` class hierarchy

1. **definition**: an **abstract class**, superclass of all **structural properties descriptors** of an **AST node**.

1.3.4.1.3 `SimplePropertyDescriptor`

1. **definition**: a **simple structural property** with **boxed primitive type values** (*e.g.* `int` → `Integer`, `boolean` → `Boolean`, *etc.*) or **simple type values** (*e.g.* `String`, *etc.*).

1.3.4.1.4 `ChildPropertyDescriptor`

1. **definition**: a **structural property** having an **instance** of `ASTNode` as **value**.

1.3.4.1.5 `ChildListPropertyDescriptor`

1. **definition**: a **structural property** having a **list** of `ASTNode` **instances** as **value**.

1.3.4.2 Obtaining information from an AST node

1. **dedicated getter methods** for each **concrete AST node**: *e.g.* `getName()`, `thrownExceptions()` for `MethodDeclaration`.
2. **generic methods** manipulating **structural property descriptors** defined for every **structural property** of a given **AST node**:

```
/*
* returns the value of the given structural property "property"
* depending on the kind of the property
*/
public final Object getStructuralProperty(StructuralPropertyDescriptor property);

/*
* returns the list of structural property descriptors for the current AST node
*/
public final List structuralPropertiesForType()
```

1.3.5 Bindings

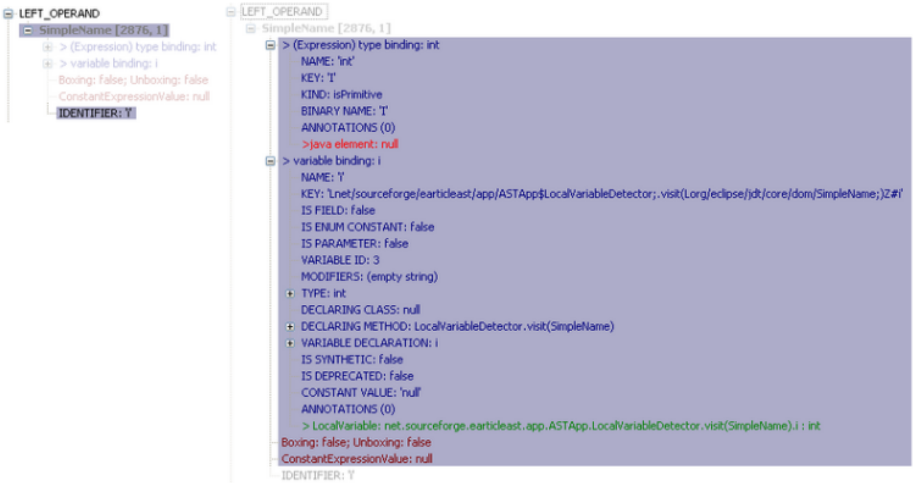
1.3.5.1 Introduction

1. **definition**: extended resolved information for several nodes of the AST.
2. **implementation**:
   - various **subclasses** of `ASTNode` have **binding information**, retrieved by calling `resolveBinding()` on these classes.
   - in some cases, **more than one binding is available**: e.g. for the class `MethodInvocation`:
     1. `resolveMethodBinding()`: returns a **binding** to the **invoked method**.
     2. `resolveTypeBinding()`: returns a **binding** to the **return type** of the **method**.
     3. `resolveBoxing()`: **information** about whether the **method invocation** is involved into a **boxing**.
     4. `resolveUnboxing()`: **information** about whether the **method invocation** is involved into an **unboxing**.
3. **usage**: explicitly requesting the binding service at parse time via `setResolveBindings(true)` invoked on the `ASTParser` instance, before the source is being parsed.
4. **example**: given a **reference** to a **variable** `i`, represented by an **instance** of `SimpleName` with "i" as `IDENTIFIER` **property-value**, **bindings** tell us, furthermore, that it is a **reference** to a **local variable** of **type** `int`.

```
int i = 7;
System.out.println("Hello!");
int x = i * 2;
```

1.3.5.2 Bindings, declarations, and references

1. **principle**: bindings allow to comfortably find out:
   - to which declaration a reference belongs



```
LEFT_OPERAND
  SimpleName [2876, 1]
    > (Expression) type binding: int
    > variable binding: i
    Boxing: false; Unboxing: false
    ConstantExpressionValue: null
    IDENTIFIER: 'i'
```

```
LEFT_OPERAND
  SimpleName [2876, 1]
    > (Expression) type binding: int
      NAME: 'int'
      KEY: 'I'
      KIND: isPrimitive
      BINARY NAME: 'I'
      ANNOTATIONS (0)
      >java element: null
    > variable binding: i
      NAME: 'i'
      KEY: 'Lnet/sourceforge/earticleast/app/ASTApp$LocalVariableDetector;.visit(Lorg/eclipse/jdt/core/dom/SimpleName;)Z#i'
      IS FIELD: false
      IS ENUM CONSTANT: false
      IS PARAMETER: false
      VARIABLE ID: 3
      MODIFIERS: (empty string)
      TYPE: int
      DECLARING CLASS: null
      DECLARING METHOD: LocalVariableDetector.visit(SimpleName)
      VARIABLE DECLARATION: i
      IS SYNTHETIC: false
      IS DEPRECATED: false
      CONSTANT VALUE: 'null'
      ANNOTATIONS (0)
      > LocalVariable: net.sourceforge.earticleast.app.ASTApp.LocalVariableDetector.visit(SimpleName).i : int
    Boxing: false; Unboxing: false
    ConstantExpressionValue: null
    IDENTIFIER: 'i'
```

Figure 5: Example of an AST reference node binding

- whether two elements are references to the same element, in which case the bindings returned by reference-nodes and declaration-nodes are identical.

2. **example**:
   - all `SimpleName`s representing a reference to a local variable `i` return the same instance of `IVariableBinding` from `SimpleName.resolveBindings()`.
   - the declaration node, `VariableDeclarationFragment.resolveBinding()`, returns the same instance of `IVariableBinding` too.
   - if there is another declaration of a local variable `i` (*within another method or block*), another instance of `IVariableBinding` is returned.

1.3.6 Error recovery

1. **principle**:
   - the ability to **recover** from code with **syntax errors** at **statement level** (*e.g. a missing semi-colon is no longer a problem to retrieve the statements of a method*)
   - it cannot recover from all kinds of syntax errors.
2. **usage**: explicitly requesting the recovery service at parse time via `setStatementsRecovery(true)` invoked on the `ASTParser` instance, before the source is being parsed.
3. **implementation**:
   - **approach**: any instance of a subclass of `ASTNode` can be tagged with some **bits** providing information about the **node's creation**.
   - **bits**: retrieved via `getFlags()` invoked on the `ASTNode`, containing:
     1. `MALFORMED`: the node is syntactically malformed.

    2. `ORIGINAL`: the node is originally created by the `ASTParser`.
    3. `PROTECT`: the node is protected from further modification.
    4. `RECOVERY`: the node (*or part of it*) recovered from source code contains a syntax error.

4. **utility**: when checking the flags of traversed nodes, a node flagged with `RECOVERED` might not contain the expected nodes.
5. **example**:

```
public static void main(String[] args){
  System.out.println("Hello"); // statement with no problem
  System.out.println(",") // statement with a RECOVERED flag
  System.out.println("World!"); // statement with no problem
}
```

1.3.7 How to apply changes

1.3.7.1 Introduction

1. **need**: new AST nodes may have to be created and applied back to the corresponding source code.
2. `AST`:
    - a **final class** designating an **AST** and offering methods to create every AST node type.
    - an instance of it:
        1. is created when source code is parsed ;
        2. can be obtained from every node of the tree: `node.getAST()`.
3. **constraints**:
    - the newly created nodes can only be added to the tree that the instance of `AST` was retrieved from.
    - AST nodes cannot be re-parented: once connected to an AST, they cannot be attached to a different place in the tree.
4. **creating a copy from an AST subtree**:
    - **pro**: it is convenient to reuse an existing subtree of an AST and just change some details
    - **implementation**:

```
/*
* returns a deep copy of the subtree of AST nodes rooted at "node".
* The resulting nodes are owned by "target", which could be different
* from "node"'s AST (i.e. established by another parser run)
*
* target: the target AST used to own newly copied nodes
* node: the root of the subtree to copy
*/
public static copySubtree(AST target, ASTNode node);
```

5. **modifying an AST and tracking the modifications**:

- noting the modifications in a separate protocol, handled by an instance of `ASTRewrite`.
- directly modifying the AST ;

1.3.7.2 Using `ASTRewrite`

1. **approach**: the changes are noted by an instance of `ASTRewrite`, the original AST is left untouched.
2. **pro**: create more than one instance of `ASTRewrite` for the same AST, which means that different change logs can be set up.
3. **note**: the more sophisticated and preferable way to track modifications on an AST.

1.3.7.2.1 Example of adding a child node into a list of child nodes (child list property value) using `ASTRewrite`

```
// creating an instance of rewrite on the CompilationUnit "unit"'s AST
ASTRewrite rewrite = ASTRewrite.create(unit.getAST());

VariableDeclarationStatement statement = createNewVariableDeclarationStatement(
manager, ast); // creating a new variable declaration statement

// getting the first reference index of the block in which to add the statement
int firstReferenceIndex = getFirstReferenceIndex(manager, block);

// recover the list of statements of the block to rewrite
ListRewrite statementsListRewrite = rewrite.getListRewrite(
block, Block.STATEMENTS_PROPERTY);

// inserting the statement into the list of statements at the 1st reference index
statementsListRewrite.insertAt(statement, firstReferenceIndex, null);
```

1.3.7.2.2 Example of modifying a child node (single-child property value) using `ASTRewrite`

```
ASTRewrite rewrite = ASTRewrite.create(unit.getAST());
// renaming the name of the method invocation
rewrite.set(methodInvocation, MethodInvocation.NAME_PROPERTY, newName, null);

// same effect using a different method
rewrite.replace(methodInvocation.getName(), newName, null);
```

1.3.7.3 Directly modifying the AST

1. the AST is modified directly.

2. the change recording has to be enabled on the root of the AST (i.e. the `CompilationUnit`): `unit.recordModifications()`
3. internally, the changes are logged to an `ASTRewrite` instance as well.

1.3.7.3.1 Example of adding a child node into a list of child nodes (child list property value) directly on the AST

```
// enable modification recording at the root of the AST
unit.recordModifications();
// ...
VariableDeclarationStatement statement = createNewVariableDeclarationStatement(
manager, ast); // creating a new variable declaration statement

// getting the first reference index of the block in which to add the statement
int firstReferenceIndex = getFirstReferenceIndex(manager, block);

// adding the statement at the beginning of the block
block.statements().add(firstReferenceIndex, statement);
```

1.3.7.4 Writing back into the source code

1. **principle**: a `org.eclipse.text.TextEdit` object, containing character based modification information, has to be created to write changes on the AST back into the source code.
2. **note**: an existing Java source file that has already been parsed should not have anything written above, since it might be edited by other editors. Thus, a shared working copy of the file should be used instead.
3. **implementation**:

```
/*
* document: the source code file parsed by ASTParser
* options: source code formatter options, (null for default options)
*/

// invoked on the ASTRewrite instance if used
TextEdit rewriteAST(IDocument document, Map options);

// invoked on the CompilationUnit if the AST is directly modified
TextEdit rewrite(IDocument document, Map options);

/*if source code = char[] and not a java file*/
IDocument document = new org.eclipse.jface.text.Document(stringSource);
```

4. **generic approach**:

```
// get a file buffer manager
ITextFileBufferManager bufferManager = FileBuffers.getTextFileBufferManager();
```

```
// get the path of the source file "unit" (CompilationUnit)
IPath path = unit.getJavaElement().getPath();

try{
  /* connect a path of a file buffer manager
  * after this call, the document of the file described by "path"
  * can be obtained and modified
  */
  bufferManager.connect(path, null);

  // retrieve the text file buffer
  ITextFileBuffer textFileBuffer = bufferManager.getTextFileBuffer(path);

  // ask the buffer for a working copy of the document
  IDocument document = textFileBuffer.getDocument();

  /*edit the document*/

  // commit changes to the underlying file
  textFileBuffer.commit(null /*ProgressMonitor*/, false /*Overwrite*/);
}

catch (Exception e){
  /*handle exception*/
}

finally {
  /* disconnect the file buffer manager from the path
  * after this call, the document of the file described by "path"
  * should no longer be modified
  */
  bufferManager.disconnect(path, null);
}
```

1.3.8 Managing comments

1. **adding a comment**:

```
CompilationUnit astRoot = ...; // current compilation unit

// creating an instance of rewrite on the CompilationUnit "astRoot"'s AST
ASTRewrite rewrite = ASTRewrite.create(astRoot.getAST());

// retrieving the body of the first method of the first class in the unit
```

```
Block block = ((TypeDeclaration) astRoot.types().get(0))
  .getMethods()[0].getBody();

// retrieve the list of statements of the body
ListRewrite listRewrite = rewrite.getListRewrite(block, Block.STATEMENTS_PROPERTY);

// creating a comment
Statement placeHolder = rewrite.createStringPlaceholder("//mycomment",
ASTNode.EMPTY_STATEMENT);

// inserting the comment prior to the method's body
listRewrite.insertFirst(placeHolder, null);

// retrieving the text-based modifications introduced to the AST of "astRoot"
TextEdit textEdits = rewrite.rewriteAST(document, null);

// committing the changes to the document
textEdits.apply(document);
```

2. **utility methods**:
    - `unit.getCommentList()`: returns a **read-only** list of the unit's comments in **ascending order**.
    - `unit.getExtendedLength(node)` or `unit.getExtendedStartPosition(node)`: retrieve the range of a node containing surrounding comments and whitespaces.
    - `declaration.getJavadoc()`: get the **javadoc comment** defined prior to a method/field/type declaration (types = classes, interfaces, annotations and enums)

2 Spoon

2.1 Introduction

1. **definition**: an open-source library to transform and analyze Java source code.
2. **approach**:
    - provides a complete and fine-grained Java metamodel where any program element can be accessed both for reading and modification.
    - takes source code as input and produces transformed source code ready to be compiled.
3. **note**: supports bytecode analysis through decompilation.

2.2 The Metamodel

2.2.1 Review

1. a programming language can have different metamodels
2. a model or an AST is an instance of a metamodel
3. each metamodel (and its instances) is more or less appropriate depending on the task at hand.
4. **metamodel examples**:
   - the Java metamodel of Sun's compiler (javac), designed and optimized for bytecode compilation
   - the Java metamodel of Eclipse IDE (JDT), designed and optimized to support different software development tasks in an integrated manner (code completion, quick fix of compilation errors, debugging, etc.)

2.2.2 The Spoon Java Metamodel

1. **design**:
   - to be easily understood by Java developers who want to write programs to perform analyses and transformations.
   - complete, as it contains all the required information to derive compilable and executable Java programs
2. **architecture**:
   - the **structural** part: declarations of program elements (e.g. interfaces, classes, variables, methods, annotations, enums)
   - the **code** part: executable Java code (e.g. method bodies)
   - the **reference** part: references to program elements (e.g. a reference to a type)

2.2.3 Structural elements

1. **root**: `CtElement` interface is the root of the Spoon Java metamodel, declaring a parent element denoting the containment relation in the source file
2. **containment relation**: a parent node contain many children nodes (*e.g. the parent of a method node is a class*).
3. **naming**: all element names are prefixed with "**Ct**" (*compile-time*).

2.2.3.1 `CtElement` interface

1. **definition**: the root interface of the Spoon Java metamodel
2. **methods**:

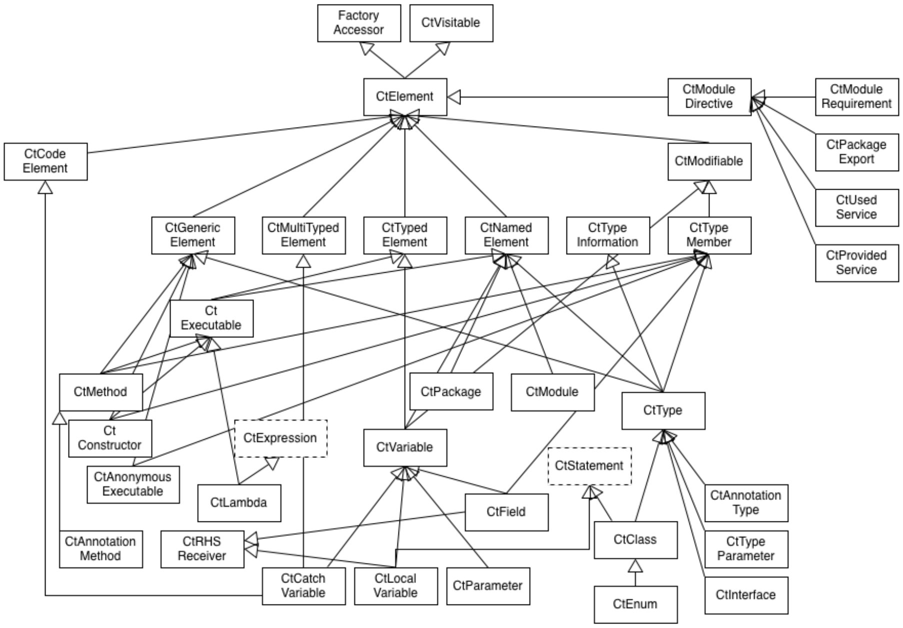


Figure 6: Spoon Java Metamodel of Structural Elements

```
/*
* the list of comments attached to the element
* note: uses simple heuristics that work well in nominal cases
* but cannot address all specific cases
*/
List<CtComment> getComments();

/*
* list of the children nodes contained within the current parent node
*/
List<CtElement> getDirectChildren();

/*
* returns the javadoc documentation comment of this element
* as a String
*/
String getDocComment();

/*
* returns all children elements matching the filter
*/
<E extends CtElement> List<E> getElements(Filter<E> filter);
```

```
/*
* returns the path from the model root element to this element
*/
CtPath getPath();

/*
* get the element position in input source files
*/
SourcePosition getPosition();
```

2.2.4 Code elements

1. **statements**:
   - **definition**: untyped top-level instructions that can be used directly within a code block
   - **implementation**: `CtStatement` interface.
2. **expressions**:
   - **definition**: typed entities used inside statements
   - **implementation**: `CtExpression<T>` generic interface, where `T` is the expression's type, added for static type checking when transforming programs.
3. **note**: some elements are both statements and expressions (interface multiple inheritance).
4. **examples**:
   - `CtLoop`: a loop statement with a `CtExpression<T>` expressing its boolean conditiion
   - `CtInvocation<T>`: an invocation statement, which is both a statement and an expression (inheriting from both `CtStatement` and `CtExpression`), where `T` is the return type of the invocation
   - `CtAssignment<T, A extends T>`: an assignment statement, which is both a statement and an expression, where `T` is the type of the assigned expression and `A` is the type of the expression to assign.

2.2.4.1 `CtArrayRead<T>`

1. **definition**: an element defining a read access to an array of elements of type `T`
2. **example**:

```
int[] array = new int[10];
array[0]; // array read
```

2.2.4.2 `CtArrayWrite<T>`

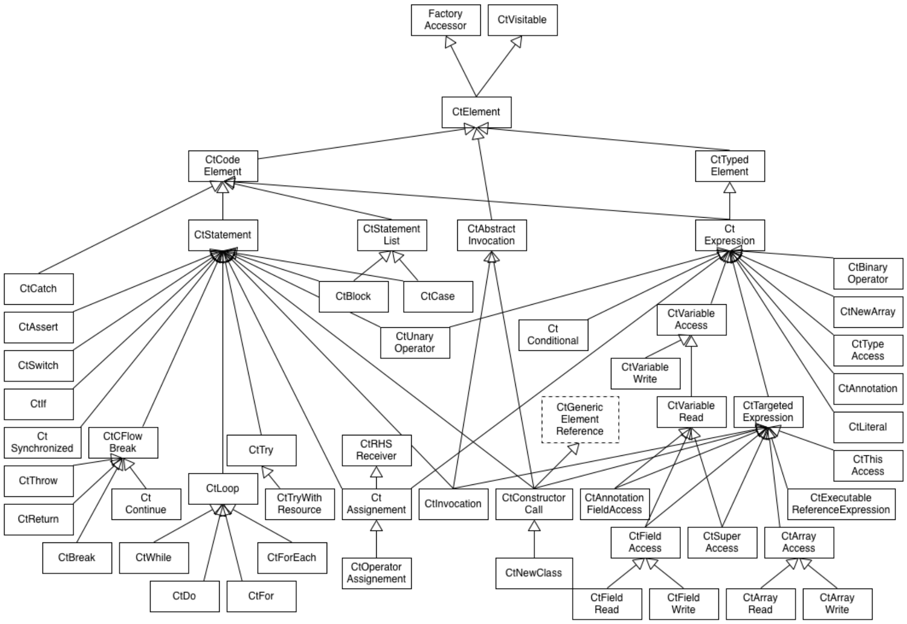


Figure 7: Spoon Java Metamodel of Code Elements

1. **definition**: an element defining a write access to an array of elements of type `T`
2. **example**:

```
int[] array = new int[10];
array[0] = 20; // array write
```

2.2.4.3 `CtAssert<T>`

1. **definition**: an element defining an assertion clause with possibly an expression of type `T`.
2. **example**:

```
// assertion clause
assert 1+1==2;
```

2.2.4.4 `CtAssignment<T, A>`

1. **definition**: an element defining an assignment, where `T` is the type of the assigned expression and `A` is the type of the expression to assign.
2. **example**:

```
int x;
x = 4; // assignment
```

2.2.4.5 CtBinaryOperator<T>

1. **definition**: an element defining a binary operator, where T is the type of the expression returned by the operator.
2. **example**:

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

2.2.4.6 CtBlock<R>

1. **definition**: an element defining a code block, where R is the type of the return statement in the block if it exists.
2. **example**:

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

2.2.4.7 CtBreak

1. **definition**: an element defining a break statement.
2. **example**:

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

2.2.4.8 CtCase<S>

1. **definition**: an element defining a case statement in a switch statement, where S is the type of the selector expression.
2. **example**:

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

2.2.4.9 CtConditional<T>

1. **definition**: an element defining a conditional expression using the ternary operator, where T is the type of the ternary expression returned.
2. **example**:

```
System.out.println(1==0? "foo": "bar"); // ternary expression returning a string
```

2.2.4.10 CtConstructorCall<T>

1. **definition**: an element defining a constructor call, where T is the type of the created object.
2. **example**:

```
new Object(); // constructor call, returning an object of type "Object"
```

2.2.4.11 CtContinue

1. **definition**: an element defining a continue statement.
2. **example**:

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

2.2.4.12 CtDo

1. **definition**: an element defining a do while loop.
2. **example**:

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

2.2.4.13 CtExecutableReferenceExpression<T, E extends CtExpression<?>>

1. **definition**: an element defining an executable reference expression, where T is the type of an interface with one method designating the executable reference, and S is the return type of the interface's method
2. **example**:

```
// an executable reference expression
// T = Supplier
// S = Object
java.util.function.Supplier p = Object::new;
```

2.2.4.14 CtFieldRead<T>

1. **definition**: an element defining a read access to a field of type T.
2. **example**:

```
Class Foo { int field; }
Foo x = new Foo();
System.out.println(x.field); // field read access
```

2.2.4.15 CtFieldWrite<T>

1. **definition**: an element defining a write access to a field of type T.
2. **example**:

```
Class Foo { int field; }
Foo x = new Foo();
x.field = 0; // field write access
```

2.2.4.16 CtFor

1. **definition**: an element defining a for loop.
2. **example**:

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

2.2.4.17 CtForEach

1. **definition**: an element defining a foreach loop.
2. **example**:

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

2.2.4.18 CtIf

1. **definition**: an element defining a if/else statement.
2. **example**:

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

2.2.4.19 CtInvocation<T>

1. **definition**: an element defining an invocation expression, where T is the return type of the invocation.
2. **example**:

```java
// invocation of println()
// target is System.out
// T = void
System.out.println("foo");
```

2.2.4.20 CtJavaDoc

1. **definition**: an element defining a javadoc comment
2. **example**:

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

2.2.4.21 CtLambda<T>

1. **definition**: an element defining a lambda statement, where T is the return type of the lambda.
2. **example**:

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

2.2.4.22 CtLiteral<T>

1. **definition**: an element defining a literal value, where T is the type of the literal value.
2. **example**:

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

2.2.4.23 CtLocalVariable<T>

1. **definition**: an element defining a local variable, where T is type of the variable.
2. **example**:

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

2.2.4.24 CtNewArray<T>

1. **definition**: an element defining the inline creation of an array, where T is the type of the array's elements.
2. **example**:

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

2.2.4.25 CtNewClass<T>

1. **definition**: an element defining the creation of an anonymous class, where T is the type of the created anonymous class.
2. **example**:

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

2.2.4.26 CtOperatorAssignment<T, A extends T>

1. **definition**: an element defining a self-operated assignment (i.e. shortcut operator assignment syntax).
2. **example**:

```
int x = 0;
x *= 3; // an self-operated assignment of x
```

2.2.4.27 CtReturn<R>

1. **definition**: an element defining a return statement, where R is the return type of the statement.
2. **example**:

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

2.2.4.28 CtSuperAccess<T>

1. **definition**: an element defining an access to super, where T is the type of super.
2. **example**:

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

2.2.4.29 CtSwitch<S>

1. **definition**: an element defining a switch statement where S is the type of the selector expression.
2. **example**:

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

2.2.4.30 CtSynchronized

1. **definition**: an element defining a synchronized statement.
2. **example**:

```
java.util.List<Object> l = new java.util.ArrayList<>();
synchronized(l){ // a synchronized statement
  System.out.println("foo");
}
```

2.2.4.31 CtThisAccess<T>

1. **definition**: an element defining an access to this, where T is the type of this.
2. **example**:

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

2.2.4.32 CtThrow

1. **definition**: an element defining a throw statement.
2. **example**:

```
// a throw statement
throw new RuntimeException("oops");
```

2.2.4.33 CtTry

1. **definition**: an element defining a try statement.
2. **example**:

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

2.2.4.34 CtTryWithResource

1. **definition**: an element defining a try with resource statement.
2. **example**:

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

2.2.4.35 CtTypeAccess<A>

1. **definition**: an element defining a type reference usable as an expression, used for:
   - static accesses ;
   - Java 8 method references ;

- instanceof binary expressions ;
- `.class` access
- where `A` is the access type.

2. **example**:

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

// access to a static method
Class.forName("Foo");

// Java 8 method reference
java.util.function.Supplier function = Object::new;

// instanceof test
Boolean x = new Object();
System.out.println(x instanceof Integer);

// .class access
Class x = Number.class;
```

2.2.4.36 `CtUnaryOperator<T>`

1. **definition**: an element defining a unary operator, where `T` is the type of the expression returned by the operator.
2. **example**:

```
int x = 3;
--x; // -- is a unary operator
```

2.2.4.37 `CtVariableRead<T>`

1. **definition**: an element defining a read access to a variable outside an assignment, where `T` is the type of the variable.
2. **example**:

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

2.2.4.38 `CtVariableRead<T>`

1. **definition**: an element defining a write access to a variable inside an assignment, where `T` is the type of the variable.
2. **example**:

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

2.2.4.39 `CtWhile`

1. **definition**: an element defining while loop.
2. **example**:

```
int x = 0;
while (x != 10){ // a while loop
  x++;
}
```

2.2.4.40 `CtAnnotation<A>`

1. **definition**: an element defining the annotation of an element, where `A` is the type of the annotation.
2. **example**:

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

2.2.4.41 `CtClass<T>`

1. **definition**: an element defining a class declaration statement.
2. **example**:

```
// a class declaration statement
class Foo {
  int x;
}
```

2.2.5 References

1. **referencing**: a program having weak references to elements that are not necessarily reified into the metamodel, as they may belong to third party libraries.
2. **example**: referencing a String object is a type reference of String and not a reference to the compile-time model of String.java, since the source code of String is (usually) not a part of the application code under analysis.
3. **utility**: flexibility in constructing/modifying a program model, without having strong references to all referred elements.
4. **reference resolution**:
    - happens when the model is built

- resolved references = references pointing to classes whose source code path is available as input to Spoon
- targets of references don't have to necessarily exist before one can reference them, since references are weak.

5. **code transformation limitation**: low coupling entails the necessity of a navigation chain from the element, to its reference, all the way to its target (*e.g.* `field.getType().getDeclaration()` to navigate from a field to its type)

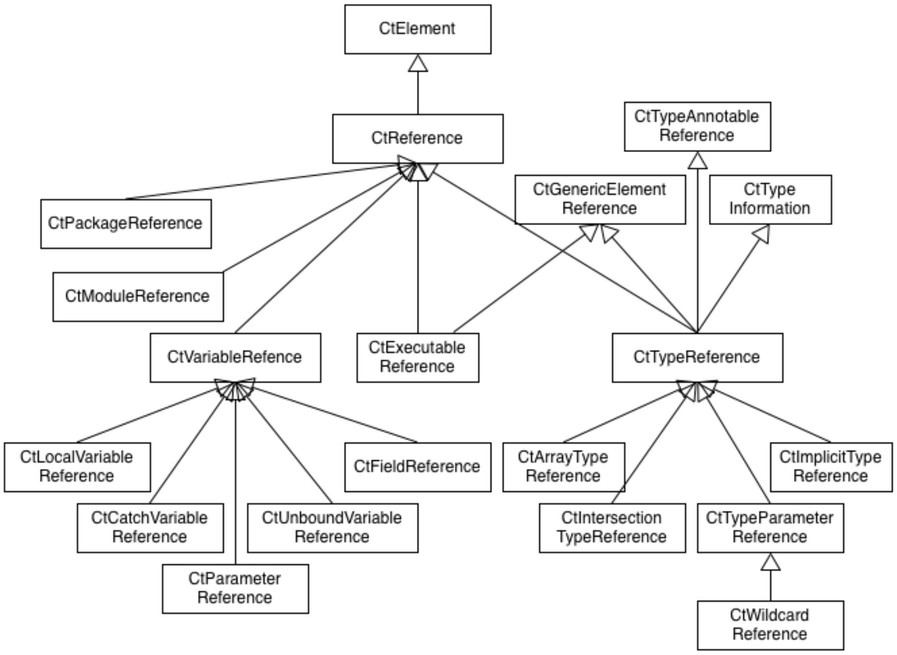


Figure 8: Spoon Java Metamodel of Reference Elements

2.2.6 Factories

1. **need**: when designing or implementing transformations, using templates or processors, we have to create new elements, fill their data, and add them to the built AST.
2. **approach**: using goal specific factories to facilitate the creation of new AST nodes.

2.2.6.1 `Factory` interface

1. **definition**: the entry point for all Spoon sub-factories, providing access to all of them.

2.2.6.2 `SubFactory` class

1. **definition**: the abstract superclass of all Spoon sub-factories

2.2.6.3 `CoreFactory` interface

1. **definition**: a factory to create any metamodel element, through core creation methods and setters used to initialize it.

2.2.6.4 `CodeFactory` interface

1. **definition**: a factory to create code elements, through utility methods requiring minimal information.

2.2.6.5 `PackageFactory` interface

1. **definition**: a factory to create and get package references, through utility methods, linked to the `CtPackage` metamodel element.

2.2.6.6 `TypeFactory` interface

1. **definition**: a factory:
   - containing utility methods linked to the `CtType` metamodel element;
   - allowing to get any type from its fully qualified type or .class invocation;
   - allowing to create typed references like `CtTypeReference`

2.2.6.7 `ClassFactory` interface

1. **definition**: a subclass of `TypeFactory` defining a factory specialized for `CtClass` metamodel elements.

2.2.6.8 `InterfaceFactory` interface

1. **definition**: a subclass of `TypeFactory` defining a factory specialized for `CtInterface` metamodel elements.

2.2.6.9 `EnumFactory` interface

1. **definition**: a subclass of `TypeFactory` defining a factory specialized for `CtEnum` metamodel elements.

2.2.6.10 `ExecutableCodeFactory` interface

1. **definition**: a factory:
   - containing utility methods linked to the `CtExecutable` metamodel element;
   - allowing to create executable objects and set their parameters.

2.2.6.11 `ConstructorFactory` interface

1. **definition**: a subclass of `ExecutableFactory` defining a factory specialized for `CtConstructor` metamodel elements.

2.2.6.12 `MethodFactory` interface

1. **definition**: a subclass of `ExecutableFactory` defining a factory specialized for `CtMethod` metamodel elements.

2.2.6.13 `FieldFactory` interface

1. **definition**: a factory to created a valid field or a field reference.

2.2.6.14 `AnnotationFactory` interface

1. **definition**: a subclass of `TypeFactory` defining a factory specialized for `CtAnnotationType` metamodel elements, containing utility methods to annotate any element or create new annotations.

2.2.7 Comments

2.2.7.1 `CtComment.CommentType` enumeration

1. **File comments**:
   - **definition**: at the beginning of a file, generally describing the licence
   - **implementation**: `CtComment.CommentType.FILE`
2. **Line comments**:
   - **definition**: `// inline comment`
   - **implementation**: `CtComment.CommentType.LINE`
3. **Block comments**:
   - **definition**: `/* block comment */`
   - **implementation**: `CtComment.CommentType.BLOCK`
4. **Javadoc comments**:
   - **definition**: `/** javadoc comment */`
   - **implementation**: `CtComment.CommentType.JAVADOC`

2.2.7.2 CtComment interface

1. **definition**: a class defining an API to handle comments in Spoon
2. **methods**:

```
/*
* get the comment type
*/
CtComment.CommentType getCommentType();

/*
* get the comment content
*/
String getContent();
```

2.2.7.3 Comment attribution

1. comments before a class or within it are attached to it

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

1. comments above or inline to a statement element are attached to it.

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

1. any element can have multiple comments above it.

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

1. comments at the end of a block are considered as orphan comments.

```
try {

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

1. comments that are alone on one or more lines are associated to the first element following them.

```
int a;
```

```
// lonely comment on line 1
// lonely comment on line 2
// lonely comment on line 3


int b; // element to which the comments are attached
```

6. comments cannot be associated to other comments.

2.2.8 Source Position

2.2.8.1 `SourcePosition` interface

1. **definition**: an interface defining the position of a `CtElement` in the original source code file.

2.2.8.2 `CompoundSourcePosition` interface

1. **definition**: a subinterface of `SourcePosition` specializing in expression and statement elements.

2.2.8.3 `DeclarationSourcePosition` interface

1. **definition**: a subinterface of `CompoundSourcePosition` specializing in all declaration statement elements (class, interface, enum, etc.), providing easy access to modifiers, names, etc.

2.2.8.4 `BodyHolderSourcePosition` interface

1. **definition**: a subinterface of `CompoundSourcePosition` specializing in method or type declaration elements, providing easy access to modifiers, names, etc.

2.3 AST Traversal

2.3.1 Visualizing the Spoon AST of a Java source file

```
java -cp <spoon-jar> spoon.Launcher -i <class>.java --gui --noclasspath
```

2.3.2 Getters

1. appropriate child node getters to any AST element:

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

2. generic getter based on the role played by the child node within its parent:

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

3. getter of all children nodes for any AST element (not recommended):

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

2.3.3 Filters

2.3.3.1 Introduction

1. **principle**: selecting nodes that match a predicate, defined by the means of a boolean method acting as a filter (e.g. `CtElement::getElement(Filter)` or `CtQueryable::filterChildren(Filter)`)
2. **examples**:

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

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

// 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();
```

2.3.3.2 `Filter<T extends CtElement>` interface

1. **definition**: an interface defining a filter for metamodel elements, where `T` is the type of the filtered elements.
2. **methods**:

```
/*
* the matcher indicating whether to filter an element or not
*/
boolean matches(T element);
```

2.3.4 Scanners

1. **principle**: a simple way to visit a node and its children through a process of scanning

2.3.4.1 CtScanner interface

1. **definition**:
   - a scanner implementing a deep-search scan on the model, ensuring that all children nodes are visited once.
   - visit = three method calls (enter(), scan(), and exit())
2. **example**:

```
// Scanner counting the number of times a field has been accessed for writing
class CounterScanner extends CtScanner {
  private int visited = 0;

  @Override
  public <T> void visitCtFieldWrite(CtFieldWrite<T> fieldWrite){
    visited++;
  }

  public int getVisited(){ return visited; }
}

CounterScanner scanner = new CounterScanner();

// running the scanner on the CtClass element named FieldAccessRes
launcher.getFactory().Class().get("FieldAccessRes").accept(scanner);

assertEquals(1, scanner.getVisited());
```

2.3.4.2 EarlyTerminatingScanner interface

1. **definition**:
   - a scanner specializing CtScanner supporting early termination of the scanning process when terminate() is called
   - useful especially when searching for a specific node only.

2.3.5 Iterators

1. **principle**: a simple way to iterate over the children of a node.

2.3.5.1 CtIterator interface

1. **definition**: an iterator over the children elements in the tree of a given node, in depth-first order.
2. **example**:

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

2.3.5.2 `CtBFSIterator` interface

1. **definition**: an iterator over the children elements in the tree of a given node, in breadth-first order.

2.3.6 Queries

2.3.6.1 Introduction

1. **principle**: an improved filtering mechanism introduced in Spoon 5.5
2. **characteristics**:
   - queries can be done through Java 8 lambdas.
   - queries can be chained ;
   - queries can be reused on multiple input elements ;
3. **chaining**:
   - in a query, several steps can be chained by chaining calls to map and filter functions of `CtQuery`.
   - non-null output of a step is passed as input to the next step in the query.
   - an Iterable or array output is considered as a set of different inputs for the next step
4. **reuse**: through `setInput()` of `CtQuery`.
5. **evaluation**:
   - `first()` of `CtQuery`: evaluates the query until the first query result is found, terminating the evaluation process afterwards, and returns the result.
   - `list()` of `CtQuery`: evaluates the query and returns the `List` of results.
   - `forEach()` of `CtQuery`: sends each query result to the output consumer `accept()` method (efficient when query results can be immediately processed).

2.3.6.2 `CtQueryable` interface

1. **definition**: an object that can be subject to queries
2. **methods**:

```
/*
* query elements filtered through "filter"
*/
<R extends CtElement> CtQuery filterChildren(Filter<R> filter);
```

```
/*
* query elements based on a function
*/
<I,R> CtQuery map(CtFunction<I,R> function);

/*
* query elements based on a consumable element
*/
<I> CtQuery map(CtConsumable<I> queryStep);
```

2.3.6.3 CtQuery interface

1. **definition**: a query that can be used to traverse a Spoon AST and collect elements in several ways.
2. **methods**:

```
/*
* recursively scan all children elements of a node
*/
<R extends CtElement> CtQuery filterChildren(Filter<R> filter);

/*
* evaluates the query and returns the first element produced in the last step
* terminates the evaluation process once the result is found
*/
<R> R first();

/*
* evaluates the query and returns each element
* by calling consumer.accept(outputElement)
*/
<R> void forEach(CtConsumer<R> consumer);

/*
* evaluates the query and returns all the elements produced in the last step
*/
<R> List<R> list();

/*
* query elements based on a function. The behavior depends on R
*
* if R = Boolean: filter elements
* if R extends Object: send the result to the next step
* if R = Iterable: send each item of the collection to the next step
* if R = Object[]: send each item of the array to the next step
*/
```

```
<I,R> CtQuery map(CtFunction<I,R> function);

/*
* same as map(CtFunction) but the returned output is handled by
* a call to outputConsumer.accept(Object)
* for efficient and easy chained processing
*/interface
<I> CtQuery map(CtConsumableFunction<I> queryStep);

/*
* sets the input(s) of the query
*/
<T extends CtQuery> T setInput(Object... input);
```

2.3.6.4 QueryFactory class

1. **definition**: a subclass of `SubFactory` specialized for creating queries on the AST metamodel
2. **methods**:

```
/*
* creates an unbound query
*/
public CtQuery createQuery();
```

2.3.6.5 Examples

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

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

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

// a query which processes non-deprecated methods of deprecated classes
list4 = rootPackage
        .filterChildren((CtClass cls) ->
            cls.getAnnotation(Deprecated.class) != null)
        .map((CtClass cls) -> cls.getMethods())
        .map((CtMethod<?> method) ->
            method.getAnnotation(Deprecated.class) == null)
        .list();

// reusing a query
CtQuery q = Factory
            .createQuery()
            .map((CtClass cls) -> c.getSimpleName());
String cls1Name = q.setInput(Class1).list().get(0);
String cls2Name = q.setInput(Class2).list().get(0);

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

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

2.4 AST Paths and Roles

2.4.1 Introduction

1. **definition**: `CtPath` defines a path to a `CtElement` in the metamodel similarly to `XPath` for `XML`.
2. **example**: `.spoon.test.path.testclasses.Foo.foo#body#statement[index=0]` is the first statement of the body of method `foo()`.
3. **constitution**:
   - element names;
   - element roles.
4. **element role**:
   - **definition**: a relation between two AST nodes.
   - **implementation**: `CtRole` enum.
5. **evaluation**: paths are evaluated on given root nodes to find code elements.

2.4.2 `CtPath` interface

1. **definition**: a path to a `CtElement` in the spoon metamodel.

2. **methods**:

```
/*
* search for elements matching this path starting from start node(s) "startNode"
*/
<T extends CtElement> List<T> evaluateOn(CtElement... startNode);

/*
* returns path relative to parent node
* used to have relative paths instead of absolute paths
* from the root package
*/
CtPath relativePath(CtElement parent);
```

2.4.3 CtPathStringBuilder class

1. **definition**: a path builder creating a path from a string with a specific syntax.
2. **methods**:

```
/*
*
*/
public CtPath fromString(String pathStr) throws CtPathException;
```

3. **string syntax**:
   - `.<name>`:
     1. **description**: child element named “name”.
     2. **example**: `.fr.inria.Spoon` (*fully qualified name*)
   - `#<role>`:
     1. **description**: all children elements having the `CtRole` “role”.
     2. **example**: `.foo#body#statement[index=2]#else`, designating the else branch of the second statement of a `foo()` method’s body.
   - `name=<somename>`:
     1. **description**: filters elements having name “somename”.
     2. **example**: `.Foo#field[name=abc]`, designating the field named “abc” of class `Foo`.
   - `signature=<somesignature>`:
     1. **description**: filters methods and constructors having signature “somesignature”.
     2. **example of a method signature**: `#method[signature=compare(java.lang.String,java.l`
     3. **example of a constructor signature**: `#constructor[signature=int]`.
   - `index=<index>`:
     1. **description**: filters the “index”th element of a `List`, starting from index 0.

    2. **example**: `#typeMember[index=4]`, designating the 5th type member.

4. **example**:

```
// creating a path to the first statement of a method of the Foo test class
CtPath path = new CtPathStringBuilder()
  .fromString(".spoon.test.path.testclasses.Foo.foo#body#statement[index=0]");

// evaluating the path on a root node
List<CtElement> l = path.evaluateOn(root);
```

2.4.4 CtPathBuilder class

1. **definition**: a low-level path builder class.
2. **methods**:

```
/*
* builds the CtPath
*/
public CtPath build();

/*
* adds a name matcher to the current path
*/
public CtPathBuilder name(String name, String[]... args);

/*
* match on elements of a given type
*/
public <T extends CtElement> CtPathBuilder type(Class<T> type, String[]... args);

/*
* match on elements having a given role
*/
public CtPathBuilder role(CtRole role, String[]... args);

/*
* match current child's elements only
*/
public CtPathBuilder wildcard();

/*
* match current child's elements and their children
*/
public CtPathBuilder recursiveWildcard();
```

3. **example**:

```
// building a path using CtPathBuilder
CtPath p1 = new CtPathBuilder()
  .recursiveWildcard() // match all elements and their children
  .name("toto") // all elements named toto
  .role(CtRole.DEFAULT_VALUE) // having a default value role in the project
  .build();

// building an equivalent path using CtPathStringBuilder
CtPath p2 = new CtPathStringBuilder()
  .fromString(".**.toto#default_value");
```

2.5 Template matchers

2.5.1 Introduction

1. **principle**: match code snippets of a project in a declarative manner using templates.
2. **process**: create a matcher class containing:
   - **fields**: template parameters of the template matcher ;
   - **methods**: a dedicated method for the matcher behavior;
   - **usage**: instantiate `TemplateMatcher` with an instance the matcher class and:
     1. invoke `find()` on the template matcher instance or
     2. use it as a `Filter` in a query.
3. **example**:

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

// example of matched code snippets
// c is a local variable, a method or a field
if (c.size() > 10)
  throw new IndexOutOfBoundsException();

//foo() returns a collection
if (foo().size() > 10)
  throw new IndexOutOfBoundsException();
```

2.5.2 `TemplateParameter<T>` interface

1. **definition**: a typed template parameter, where `T` designates the type of the template parameter, used in template matchers

2. **method**:

```
/*
* returns the type of the template parameter
*/
T S();
```

2.5.3 TemplateMatcher class

1. **definition**: an engine for matching a template to pieces of code.
2. **methods**:

```
/*
* returns all target subtrees of a root node that match the current template
*/
public <T extends CtElement> List<T> find(CtElement targetRoot);
```

3. **example**:

```
/*
* a template matcher checking the bounds of a collection in
* an if statement's conditional expression
*/
public class CheckBoundMatcher {
  private TemplateParameter<Collection<?>> _col_;

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

// finding the matcher specification
Class<?> templateCls = factory.Class().get(CheckBoundMatcher.class);

// defining the matcher root
CtIf templateRoot = (CtIf) ((CtMethod) templateCls
  .getElements(new NameFilter("matcher1"))
  .get(0))
  .getBody()
  .getStatement(0);

// instantiating the template matcher with the matcher root
TemplateMatcher matcher = new TemplateMatcher(templateRoot);

// finding the code snippets that match the CheckBoundMatcher instance
for (CtElement e: matcher.find(aPackage)) {...}
```

```
// using the template matcher instance as a filter
aPackage.filterChildren(matcher)
  .forEach((CtElement e) -> {...});
```

2.6 Patterns

2.6.1 Introduction

1. **definition**: a code pattern matching mechanism applied on AST trees instead of textual source code.
2. **composition**: a pattern is composed of a list of AST models, where each model is an AST with some nodes called pattern parameters, designating the source code to match.

3. **features**:
   - source code formatting is ignored.

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

```
- comments are ignored.
```

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

```
- implicit and explicit elements are treated the same.
```

```
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);
```

2.6.2 `PatternBuilder` class

1. **definition**: a spoon pattern builder, used to create `Pattern` instances.
2. **methods**:

```
/*
* build the pattern
*/
public Pattern build();

/*
* create a pattern builder from ast nodes designating the pattern parameters
*/
public static PatternBuilder create(CtElement... elements);

/*
* configure inline statements
*/
public PatternBuilder configureInlineStatements(
Consumer<InlinedStatementConfigurator> consumer);

/*
* all variables references whose declarations are outside of the template
* model are marked as pattern parameters
*/
public PatternBuilder configurePatternParameters();

/*
* configure pattern parameters using a PatternParameterConfigurator instance
*/
public PatternBuilder configurePatternParameters(
Consumer<PatternParameterConfigurator> parametersBuilders);
```

2.6.3 Pattern class

1. **definition**: a spoon pattern class.
2. **methods**:

```
/*
* returns all target subtrees that match the pattern and
* invokes consumer.accept(Match)
*
* input: the root of the AST node to be searched
* consumer: the receiver of matches
*/
public void forEachMatch(Object input, CtConsumer<Match> consumer);

/*
* returns all target subtrees that match the pattern starting from the
* provided root of the AST node to be searched
```

```
*/
public List<Match> getMatches(CtElement root);

/*
* returns a generator which can be used to generate code using the pattern
* parameters of the current path
*/
public Generator generator();
```

2.6.4 `Match` class

1. **definition**: a single match of a spoon pattern on one or many code elements.
2. **methods**:

```
/*
* return the matching element and
* fails if more than one element matches the pattern
*/
public CtElement getMatchingElement();

/*
* same as getMatchingElement() but checks if the matching element
* is an instance of "cls", before casting it into that type
* and returning it.
* fails if more than one element matches the pattern
*/
public <T> T getMatchingElement(Class<T> cls);

/*
* return the list of matching elements
*/
public List<CtElement> getMatchingElements();

/*
* same as getMatchingElements() but checks if each matching element
* is an instance of "cls", before casting each of them into that type
* and returning them all in a list
*/
public <T> List<T> getMatchingElements(Class<T> cls);
```

2.6.5 `PatternBuilderHelper` class

1. **definition**: a class defining utility methods to select AST nodes to be used as template models by `PatternBuilder` to help in specifying pattern

parameters.

2. **methods**:

```
/*
* sets the body of the method "methodName" as a template model
*/
public PatternBuilderHelper setBodyOfMethod(String methodName);

/*
* sets the return expression of the method "methodName" as a template model
*/
public PatternBuilderHelper setReturnExpressionOfMethod(String methodName);
```

2.6.6 `PatternParameterConfigurator` class

1. **definition**: a class for the configuration of pattern parameters using lambdas
2. **methods**:

```
/*
* creates a pattern parameter with name "name"
*/
public PatternParameterConfigurator parameter(String name);

/*
* "type" and all its references are pattern parameters
*/
public PatternParameterConfigurator byType(Class<?> type);

/*
* type identified by "typeQualifiedName"
* and all its references are pattern parameters
*/
public PatternParameterConfigurator byType(String typeQualifiedName);

/*
* type referred by "type"
* and all its references are pattern parameters
*/
public PatternParameterConfigurator byType(CtTypeReference<?> type);

/*
* type identified by "localTypeSimpleName"
* within the scope of "searchScope" are pattern parameters
*/
public PatternParameterConfigurator byLocalType(CtType<?> searchScope,
String localTypeSimpleName);

/*
* variables named "name" are pattern parameters
*/
public PatternParameterConfigurator byVariable(String name)

/*
* all read/write references to variable "variable" are pattern parameters
*/
public PatternParameterConfigurator byVariable(CtVariable<?> variable);

/*
* all invocations of method named "method" are pattern parameters
*/
public PatternParameterConfigurator byInvocation(CtMethod<?> method);

/*
* all field read/write references through variables named "varName"
* designating instances of containing classes
* are pattern parameters
*/
public PatternParameterConfigurator byFieldAccessOnVariable(String varName);

/*
* all elements filtered by "filter" are pattern parameters
*/
public PatternParameterConfigurator byFilter(Filter<?> filter);

/*
* all elements filtered by "filter" and having role "role"
* are pattern parameters
*/
public PatternParameterConfigurator byRole(CtRole role, Filter<?> filter);

/*
* all String elements named "name" are pattern parameters
*/
public PatternParameterConfigurator byString(String name);

/*
* all String elements whose name contains "subString" are pattern parameters
*/
public PatternParameterConfigurator bySubString(String subString);

/*
```

```
* all named elements (i.e. CtNamedElement) named "name" are pattern parameters
*/
public PatternParameterConfigurator byNamedElement(String name);

/*
* all reference elements (i.e. CtReference) named "name" are pattern parameters
*/
public PatternParameterConfigurator byReferenceName(String name);

/*
* all elements of type "type" whose values satisfy the predicate
"matchCondition" are pattern parameters
*/
public <T> PatternParameterConfigurator byCondition(Class<T> type,
Predicate<T> matchCondition);

/*
* sets the minimum number of occurrences of this parameter's value
* to be considered during matching to accept this parameter as
* a pattern parameter
*
* min:
    - if min = 0, the pattern parameter is optional
    - if min = 1, the pattern parameter is mandatory
    - if min = n, the parameter's value must occur at least n times
*/
public PatternParameterConfigurator setMinOccurence(int min);

/*
* same as setMinOccurence() but for maximum number of occurrences instead
*/
public PatternParameterConfigurator setMaxOccurence(int max);

/*
* sets a matching strategy for the matching process using
* a value of the "Quantifier" enumeration
*/
public PatternParameterConfigurator setMatchingStrategy(Quantifier quantifier);

/*
* sets an expected type for the parameter's values, matching only occurrences
* of the parameter' values of type "valueType"
*
*/
public PatternParameterConfigurator setValueType(Class<?> valueType);
```

```
/*
* sets "kind" as the data kind for the parameter
* if not used, then the real data kind of the parameter's value is used
* if null, by default ContainerKind.SINGLE
*/
public PatternParameterConfigurator setContainerKind(ContainerKind kind);
```

2.6.7 Quantifier enumeration

1. **definition**: an enumeration of matching strategies for pattern parameters.
2. **values**:
   - Quantifier.GREEDY (default):
     1. the matcher reads the entire input before attempting the next match.
     2. if the next match fails, the matcher backs off the input by one match and tries again, until a match is found or no previous match to back off to from the input exists.
   - Quantifier.POSSESSIVE: the matcher reads the entire input, and attempts to match only once, while never backing off.
   - Quantifier.RELUCTANT: the matcher reads one character at a time, attempting to match.

2.6.8 ContaineKind enumeration

1. **definition**: an enumeration of field/role data kinds.
2. **values**:
   - ContainerKind.LIST: the field/role is a list of values.
   - ContainerKind.MAP: the field/role is a map of values.
   - ContainerKind.SET: the field/role is a set of values.
   - ContainerKind.SINGLE: the field/role is a single value.

2.6.9 InlinedStatementConfigurator class

1. **definition**: a class for the configuration of inline statements.
2. **matching example using inline statements**:

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

// can be matched to

for (int i=1; i<=n; i++)
  System.out.println(i);
```

3. **methods**:

```
/*
* marks all CtIf and CtForEach elements whose expression contains a variable
* named "name" as inline statements
*/
public InlinedStatementConfigurator inlineIfOrForeachReferringTo(String name);

/*
* marks the CtForEach element "foreachElement" as an inline statement
*/
public markAsInlined(CtForeach foreachElement);

/*
* marks the CtIf element "ifElement" as an inline statement
*/
public markAsInlined(CtIf ifElement);
```

4. **example**:

```
Pattern pattern = PatternBuilder
  .create(/*select pattern model*/)
  .configureInlineStatements((InlinedStatementConfigurator cf) ->
  /*
  * foreach and if statements containing variables named "intValues"
  * are marked as inline statements
  */
    cf.inlineIfOrForeachReferringTo("intValues")
  )
  .build();
```

2.7 Generators

2.7.1 Introduction

1. **definition**: a code generation mechanism using spoon patterns, by mapping pattern parameters, referenced by their names (`String`), to their replacing objects (`Object`).

2.7.2 `Generator` interface

1. **definition**: an interface defining a code generator based on Spoon patterns.
2. **methods**:

```
/*
* adds type members (fields and methods) to the type "targetType"
* by using the replacement map of pattern parameters "params"
* and verifying that the generated members are of type "valueType"
*/
<T extends CtTypeMember> List<T> addToType(Class<T> valueType,
Map<String, Object> params, CtType<?> targetType);

/*
* generate new AST elements using the replacement map of pattern parameters
* "params" and verifies that the generated elements are of type "valueType"
*/
<T extends CtElement> List<T> generate(Class<T> valueType
Map<String, Object> params);

/*
* generate a new type element named "fullyQualifiedName"
* using the replacement map of pattern parameters "params"
*/
<T extends CtType<?>> T generateType(String fullyQualifiedName,
Map<String, Object> params);
```

2.8 Usage

2.8.1 Launcher class

1. **definition**:
   - a CLI launcher for processing programs at compile-time using the JDT-based builder (Eclipse).
   - it takes arguments for building, processing, printing and compiling Java programs.
   - it creates a spoon model for a project (an AST).
2. **methods**:

```
/*
* adding the source file at "path" as an input to be processed by Spoon
*/
public void addInputResource(String path);

/*
* builds the AST model and returns it
*/
public CtModel buildModel();

/*
```

```
* returns the current launching environment of the spoon program.
*/
public Environment getEnvironment();

/*
* return the model built from the provided sources
*/
public CtModel getModel();

/*
* process the string "source" containing the source code
* and return the corresponding AST class element
*/
public static CtClass<?> parseClass(String source);
```

3. **usage**:

```
// processing source code as a string
CtClass l = Launcher.parseClass("class A { void m() { System.out.println(\"okay\");} }");

// processing source code as files using their paths
Launcher launcher = new Launcher();

launcher.addInputResource("path/to/src");
launcher.buildModel();

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

2.8.2 Environment interface

1. **definition**: an interface representing the environment in which Spoon is launched, used primarily to report messages, warnings, and errors.
2. **methods**:

```
/*
* returns the expected pretty printing mode of spoon model elements
*/
Environment.PRETTY_PRINTING_MODE getPrettyPrintingMode();

/*
* return the source classpath of the spoon model
*/
String[] getSourceClasspath();

/*
* checks if comment parsing is enabled or not
```

```
*/
boolean isCommentsEnabled();

/*
* enable auto imports pretty-printing mode in the target project or not
*
* equivalent to CLI argument: --with-imports
*/
void setAutoImports (boolean enabled);

/*
* enable comment parsing in the target project or not
*
* equivalent to CLI arguments: -c or --enable-comments
*/
void setCommentEnabled(boolean enabled);

/*
* enable no classpath mode for the spoon model during code analysis or not
*
* equivalent to CLI arguments: -x or --noclasspath
*/
void setNoClassPath(boolean enabled);

/*
* creates a pretty-printer and sets it for the current environment
*/
void setPrettyPrinterCreator(Supplier<PrettyPrinter> creator);

/*
* sets the source class path for the spoon model
*/
void setSourceClasspath(String[] sourceClasspath);
```

2.8.3 Pretty-printing

2.8.3.1 `PrettyPrinter` interface

1. **definition**: an interface defining pretty printers.

2.8.3.2 `DefaultJavaPrettyPrinter` class

1. **definition**: default implementation of `PrettyPrinter`, consisting of a visitor generating and printing formatted code based on the origin source code AST model.

2.8.3.3 SniperJavaPrettyPrinter class

1. **definition**: implementation of PrettyPrinter, consisting of a visitor copying as much as possible from the origin source code and printing only transformed elements, which makes it useful to get small differences between generated and original source code.

2.8.3.4 Environment.PRETTY_PRINTING_MODE

1. **definition**: an enumeration defining pretty printing modes.
2. **values**:
   - Environment.PRETTY_PRINTING_MODE.FULLYQUALIFIED:
   1. all classes and methods' names are fully-qualified
   2. the default pretty printing mode
   3. not human-readable but useful to avoid collisions
   - Environment.PRETTY_PRINTING_MODE.AUTOIMPORT:
   1. all classes and methods are imported as long as no conflict arises.
   2. the required imports are computed, added to the pretty-printed files, and names are written in unqualified format.
   - Environement.PRETTY_PRINTING_MODE.DEBUG: all classes and methods' names are printed using a DefaultJavaPrettyPrinter pretty printer.

2.8.3.5 Examples

```
// autoimport pretty-printing
launcher.getEnvironment().setAutoImports(true);

// sniper mode pretty-printing
launcher.getEnvironment()
  .setPrettyPrinterCreator(
    () -> new SniperJavaPrettyPrinter(launcher.getEnvironment())
  );
```

2.8.4 IncrementalLauncher class

1. **definition**: a spoon launcher for incremental build, allowing to cache spoon model and binary, so that any spoon analysis can be restarted from where it stopped instead of restarting from scratch.
2. **methods**:

```
/*
* checks for code changes since last code build
*/
public boolean changesPresent();
```

```
/*
* caches current spoon model and binary files
*
* should be called only after the model had been built
*/
public void saveCache();
```

3. **example**:

```
// prepare arguments for launcher
final File cache = new File("path/to/cache");
Set<File> sourceInputs = Collections.singleton(new File("path/to/src"));
Set<String> sourceClasspath = Collections.emptySet(); // empty classpath

// create launcher and start build from cache
IncrementalLauncher launcher = new IncrementalLauncher(
sourceInputs, sourceClasspath, cache);

// check if changes have been introduced since last build
if(launcher.changesPresent()){
  System.out.println("There are changes since last save to cache");
  CtModel newModel = launcher.buildModel(); // update built model
  launcher.saveCache(); // save the new model and binaries into cache
}
```

2.8.5 Bytecode analysis

2.8.5.1 JarLauncher class

1. **definition**: a launcher for source code analysis, creating an AST model from a jar containing bytecode.
2. **approach**:
   - add bytecode resources to the classpath or, if a `pom.xml` file is provided for the jar, build the classpath containing the dependencies.
   - automatically use a decompiler to extract source code from bytecode ;
   - perform code analysis on the decompiled source code.
3. **methods**:

```
/*
* returns a JarLauncher instance from the jar at "jarPath", having a pom file
* at "pom", analyzing decompiled source code at "decompiledSrcPath"
* using the default decompiler provided
*/
JarLauncher(String jarPath, String decompiledSrcPath, String pom);

/*
```

```
* returns a JarLauncher instance from the jar at "jarPath", having a pom file
* at "pom", analyzing decompiled source code at "decompiledSrcPath"
* using the decompiler "decompiler"
*/
JarLauncher(String jarPath, String decompiledSrcPath, String pom,
Decomipler decompiler);
```

4. **note**: `JarLauncher` is a part of `spoon-decompiler` jar, and not `spoon-core`.

2.8.5.2 Decompiler interface

1. **definition**: an interface defining bytecode decompilers.
2. **methods**:

```
/*
* decompile the bytecode at "inputPath" into "outputPath", using
* classpaths defined in "classPath"
*/
void decompile(String inputPath, String outputPath, String[] classPath);
```

2.8.5.3 Example

```
// creating a JarLauncher with the default decompiler
JarLauncher launcher = new JarLauncher("path/jar", "path/decompiledSrc",
"path/pom.xml");

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

// creating a JarLauncher with a custom decompiler
JarLauncher launcher = new JarLauncher("path/jar", "path/decompiledSrc",
"path/pom.xml", new Decompiler(){
  @Override
  public void decompile(String inputPath, String outputPath,
    String[] classpath){
      // custom decompiler call
  }
});
```

2.8.6 Classpath and Spoon references

1. classpath modes during code analysis by Spoon:
    - **full classpath**: all dependencies are in the JVM classpath or given to the launcher (i.e. `launcher.getEnvironment().setSourceClasspath("path/to/bin")`).

  - **no classpath**: some dependences are unknown (`launcher.getEnvironment().setNoClassPath(tr` is set)

2. consequence on type references:
    - **case 1**: the source code of the element referred to by the reference is available:
        1. `reference.getDeclaration()` returns the code element type (i.e. its `CtType`).
        2. idem for `reference.getTypeDeclaration()`.
    - **case 2**: the source code of the element referred to by the reference is absent, but the binary file is available in the classpath (through JVM or the launcher):
        1. `reference.getDeclaration()` returns `null`.
        2. `reference.getTypeDeclaration()` returns a partial type of the code element using runtime reflection, called a *shadow object*.
    - **case 3**: the source code of the element referred to by the reference and its binary file are absent (*i.e. no classpath mode*):
        1. `reference.getDeclaration()` returns `null`.
        2. `reference.getTypeDeclaration()` returns `null`.

2.9 Code analysis

2.9.1 Processors

1. **definition**:

- a program analysis is combination of query and analysis code, reified in the concept of a processor.
- a spoon processor is a class that analyzes one type of program elements.

2. **approach**:

- define a processor by extending `AbstractProcessor<E extends CtElement>`
- process the requested element as input and perform analysis
- explicitly interrupt the processing at any time using `interrupt()` and proceed to the next step (*pretty printing*).
- apply one or many processors using the launcher, in the order they have been declared.

3. **implementation**: visitor design pattern: each node of the metamodel implements an `accept()` method so that it can be visited by a visitor object.

2.9.2 AbstractProcessor<E extends CtElement>

1. **definition**: the root generic abstract class for all metamodel element processors, to be specialized by all specializing processors, where the type of model element to be analyzed is bound to the generic parameter.
2. **methods**:

```
/*
* interrupts the processing of this processor
* while changes on the AST are preserved
* and the processing of other possible processors continues
*/
public void interrupt();

/*
* tells if the element should be processed or not (true by default)
*/
public boolean isToBeProcessed(E candidate);

/*
*a callback method to access information about the generic element E
*
* method inherited from "Processor" interface
*/
public void process(E Element);
```

3. **applying the processor**:
   - identify the source code location and the list of compiled processors to be used.
   - compile each processor and all its dependencies into a corresponding `processor.jar` file.
   - specify all dependencies in the classpath
   - specify all processors in fully qualified name

```
java -classpath /path/bin/yourProcessor.jar:<spoon-jar> spoon.Launcher -i /path/src/yourProj
```

2.9.3 Example of a catch clause processor

```
public class CatchProcessor extends AbstractProcessor<CtCatch> {
  /*attributes*/
  // empty catch clauses
  List<CtCatch> emptyCatchs = new ArrayList<>();

  @Override
  public boolean isToBeProcessed(CtCatch candidate){
    // process only empty catch clauses
    return candidate.getBody().getStatements().isEmpty();
  }

  @Override
  public void process(CtCatch element){
    getFactory()
    .getEnvironment()
    .report(this, Level.WARN, "empty catch clause"
      + element.getPosition().toString());

    emptyCatchs.add(element);
  }
}
```

2.9.4 Example of a comment processor

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

  @Override
  public boolean isToBeProcessed(CtComment candidate){
    // process only javadoc comments
    return candidate.getCommentType() == CtComment.CommentType.JAVADOC;
  }

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