Ptidej ANTLR4 C# Parser

- A SOEN6971 project -

Project Report

Adrien Poupa

Supervisor

Yann-Gaël Guéhéneuc

Concordia University
Faculty of Engineering and Computer Science

Faculty of Engineering
and Computer Science
Concordia University
http://www.concordia.ca

Title:
Ptidej ANTLR4 C# Parser

Theme:
Development project conducted at Concordia

Project Period:
Summer Semester 2018

Participant:
Adrien Poupa

Supervisor:
Yann-Gaël Guéhéneuc

Copies: 3

Page Numbers: 44

Date of Completion:
August 31, 2018

Contents

Abstract

Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java) is a software program written in Java, that provides tools to enhance and evaluate the quality of object-oriented programs, by promoting the use of patterns, at the language or architectural-levels.

Ptidej analyzes programs using a meta-model, PADL (Pattern and Abstract-level Description Language). This meta-model is fed by language-specific PADL parser packages. Such packages exist for Java and C++.

The purpose of this project is to create a PADL parser for the C# language, using the latest ANTLR4 (ANother Tool for Language Recognition) library available. The question investigated is: How to create an efficient PADL parser for the C# language, based on the ANTLR4 library?

A new C# PADL parser has been created, based on previous work done by Yann-Gaël Guéhéneuc and others from the Ptidej Team. C# code was parsed using an ANTLR4 C# grammar, and the PADL model was fed using visitors on the parse trees generated by the grammar.

Results were validated using a range of unit tests: based on an oracle, we confirmed that the PADL model generated by our code was working and conform to what was expected.

It is hoped that this project will help to integrate the C# parsing feature to the Ptidej software program.

Keywords: Ptidej, PADL, ANTLR4, C#

Acknowledgement

I would like to express my sincere gratitude to my supervisor Dr Yann-Gaël Guéhéneuc for giving me the opportunity to work on this project as well as providing his highly appreciated guidance, comments and suggestions throughout the course of the project.

His input has been invaluable for me to understand the underlying mechanisms of the Ptidej tool suite and choosing the best approaches to solve the problems I encountered.

Chapter 1

Introduction

Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java) is a software program written by the Ptidej Team, lead by Yann-Gaël Guéhéneuc. It is written in Java and relies on the Eclipse platform. Its purpose is to to provide tools to enhance and evaluate the quality of object-oriented programs, by promoting the use of patterns, at the language or architectural-levels.

Ptidej analyzes programs using a meta-model, PADL (Pattern and Abstract-level Description Language). The meta-model is fed by language-specific PADL parser packages. There are packages for Java and C++.

There were already two C# available in the project (`PADL Creator C# v1` and `PADL Creator C# v2`); they were deemed to be working well but relied on a legacy version of the ANTLR library. Therefore, the purpose of this project is to create a PADL parser for the C# language, using the latest ANTLR library available at the time of the project (version 4).

A new C# PADL parser was created, based on previous work done by Yann-Gaël Guéhéneuc and others from the Ptidej Team. The parser works by taking C# code as an input, parsing it using an ANTLR4 C# grammar, then the PADL model is fed using visitors on the parse trees generated by the grammar. The visitors are provided by the ANTLR4 library.

The final results were validated using a range of unit tests: using an oracle, we confirmed that the PADL meta-model generated by our code was working and conform to what we expected.

Chapter 2

The Ptidej Tool Suite

2.1 Overview

Ptidej is a tool suite that analyzes other software. It relies on PADL, a semi-language independent model. Ptidej provides various interfaces that can be used to plug new parsers to the PADL model, thus offering the possibility to support new programming languages such as C# without having to modify the core of Ptidej.

The Ptidej tool suite is based on the following:

- `CPL`
- `EPI`
- `JavaParser`
- `JChoco`
- `Mendel`
- `PADL - PADL Analyses`
- `PADL Creator AOL`
- `PADL Creator AspectJ`
- `PADL Creator C#`
- `PADL Creator C++`
- `ADL Creator ClassFile`
- `PADL Creator JavaFile`
- `PADL Design Motifs`

- PADL Generator
- PADL Micro-pattern Analysis
- PADL Statements
- PADL Statements Creator AOL
- PADL Statements Creator ClassFile
- POM
- Ptidej
- Ptidej Solver 4
- Ptidej Solver Metrics
- Ptidej UI
- Ptidej UI Analyses
- Ptidej UI AspectJ
- Ptidej UI C++
- Ptidej UI Layouts
- Ptidej UI Primitives AWT
- Ptidej UI Viewer
- Ptidej UI Viewer Extensions
- Ptidej UI Viewer Standalone Swing
- SAD
- SAD Rules
- SQUAD

Some have not been mentioned for the sake of simplicity. For the project, I have created two new packages: PADL Creator C# v3 and PADL Creator C# v3 Tests. Indeed, PADL Creator C# v1, PADL Creator C# v2 and their associated test packages already existed before I began working on the project, but they were more of a POC (proof of concept) than a working product.

The PADL Creator C# v2 was using ANTLR3 to parse the code; after extensive studies of the best solution to parse C# code, I chose to rewrite it using the newer ANTLR4 library.

2.2 Installation

The first step of the project was to install and run the Ptidej tool suite. To do so, I have followed the instructions available on the BitBucket repository.

First, one has to clone the BitBucket repository. Then, the project can be opened in the Eclipse IDE. The `Plug-in Development Environment (PDE)` must be installed.

Once it is installed, because most of the packages need it to work, one has to select the `/PADL/META-INF/MANIFEST.MF` file, right click the `MANIFEST.MF` file, selecting Plug-in Tools, Update Classpath, Select All and then Finish. When this is done, two steps should be done to make sure that the installation was successful: [2]

- Run the jUnit tests from the `POM Test` package as shown in figure 2.1
- Run the `Ptidej UI Viewer Standalone Swing` as a Java application to actually run Ptidej (see figure 2.2)

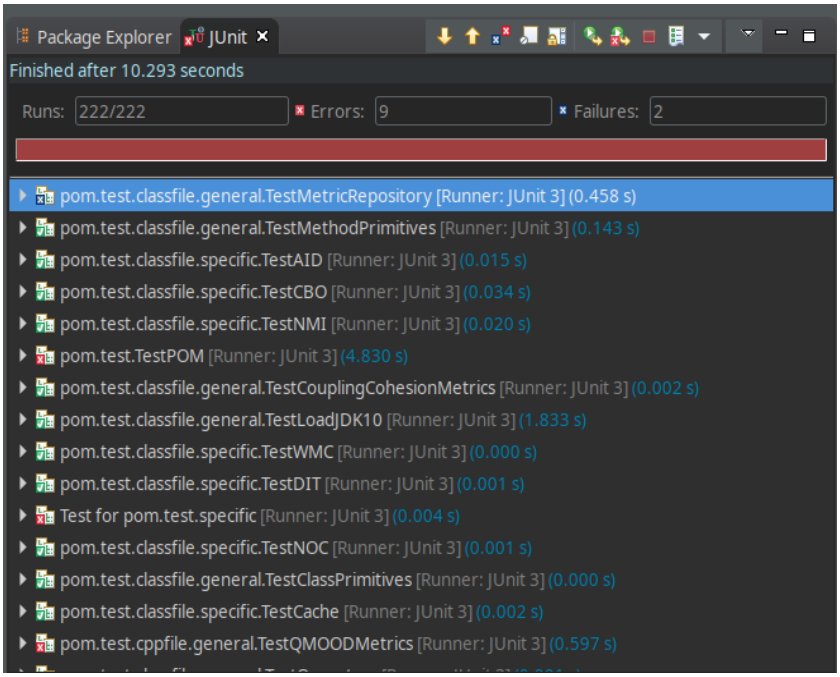

Figure 2.1: `POM Tests`; some tests are expected to fail

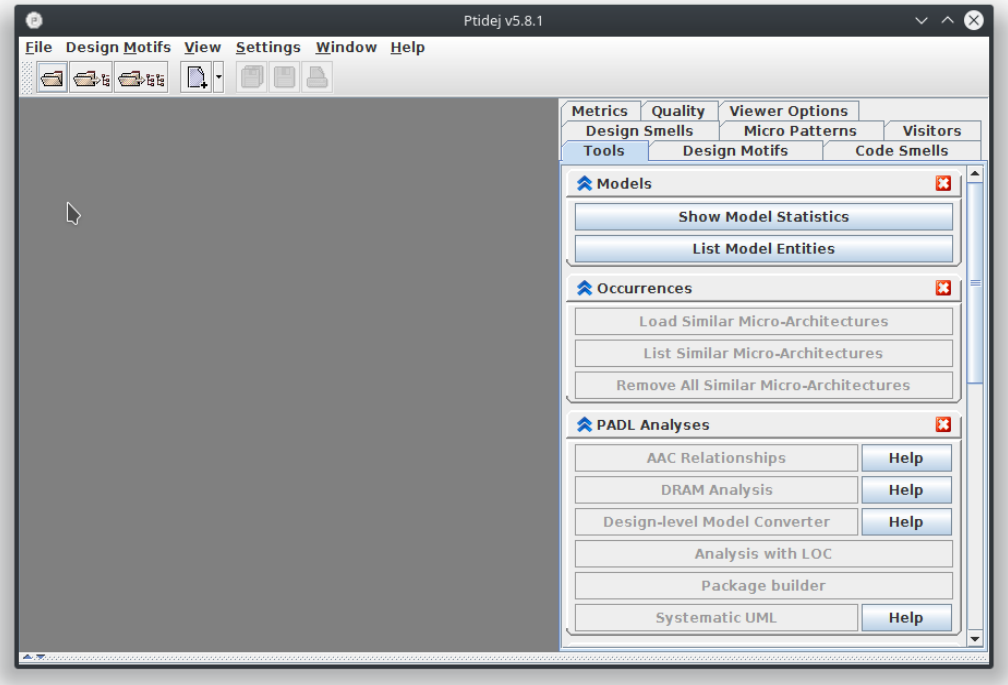

Figure 2.2: Ptidej main window

2.3 PADL Models

PADL is an acronym for Pattern and Abstract-level Description Language.

2.3.1 Levels of abstraction

It is used by the Ptidej tool suite to reflect other programs at different levels of abstractions [3]. Those levels are:

- `ICodeLevelModel`: lowest model of abstraction, containing information that is directly extracted from the sourcecode of the program (class names, function names, etc).
- `IIdiomLevelModel`: second level of abstraction, containing more specific information such as class relationships (inheritance, implementation).
- `IDesignLevelModel`: contains more advanced knowledge, like design motifs and design smells.
- `IDesignMotif`: a design motif.

The PADL package also provides two interfaces: [3]

- `IAbstractModelSerialiser`: used to serialize or deserialize a PADL model; this is useful for saving a PADL model and reloading it later without having to recompute it
- `ICodeLevelModelCreator`: the interface that has to be implemented for every new language that will be parsed as a PADL model. Ptidej currently supports Java, C/C++, AOL and AspectJ. For the needs of the project, I have created a new builder for C#.

2.3.2 Relationships

PADL models describe relationships between classes, as accurately as possible.

Class relationships are important because they allow to understand how a program works. There are seven types of relationships, from the most constraining to the least constraining. The figure 2.3 shows how they interact with each other. For the following, it is assumed that the class X has a relationship with the class Y [3].

- `IContainerComposition`: X contains a reference to an instance of Y, and has the ability to create or remove such references.
- `IComposition`: X contains a reference to an instance of Y, with no ability to create or remove such references.
- `IContainerAggregation`: X contains a reference to an instance of Y, that can possibly be shared, and has the ability to create or remove such references.
- `IAggregation`: X contains a reference to an instance of Y, that can possibly be shared, with no ability to create or remove such references.
- `IAssociation`: X calls one or more functions of Y, through one of more of Y's instances.
- `ICreation`: X creates one or more instances of Y.
- `IUseRelationship`: X uses Y; for example, a function of X has a local variable of type Y.

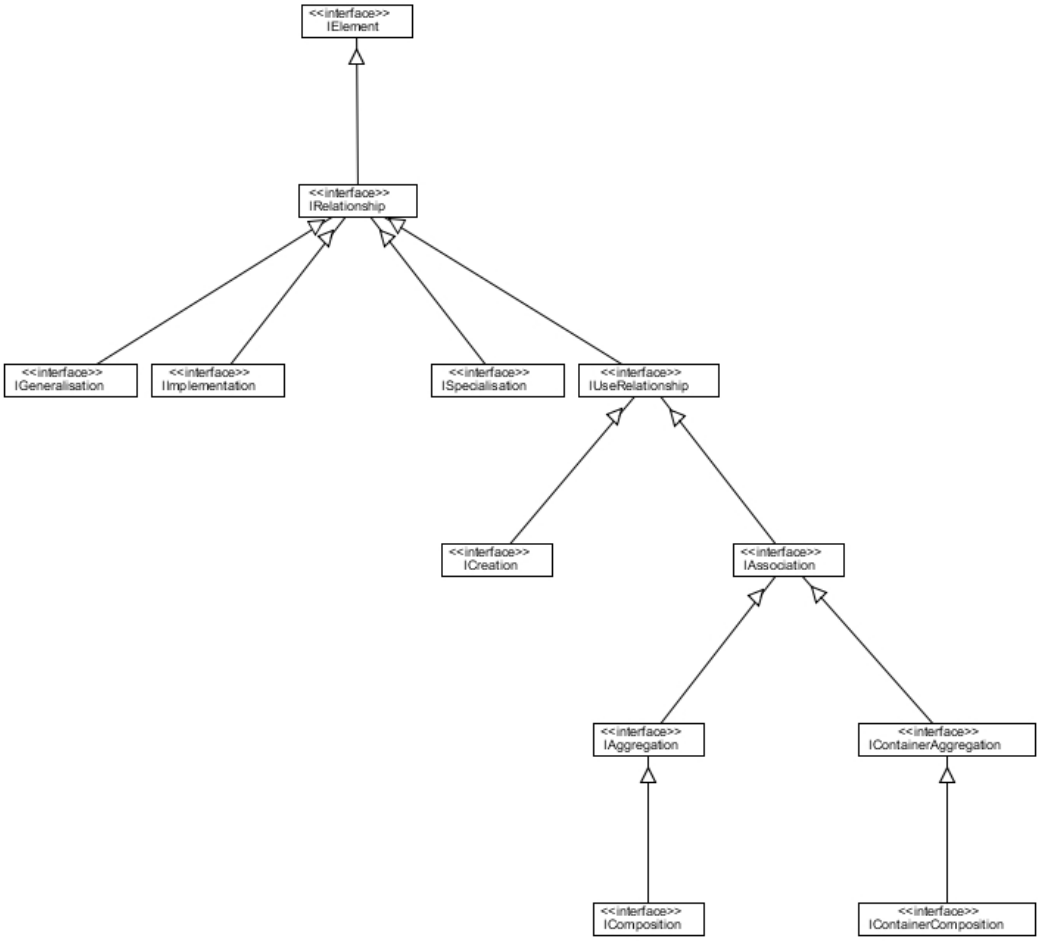

Figure 2.3: PADL relationships seen by Ptidej [3]

2.3.3 Visiting PADL models

The PADL model provides two kinds of visitors. Visitors, according to the Gang of Four:

"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates [7, p.366]." The figure 2.4 is a diagram of how the visitor pattern can be implemented.

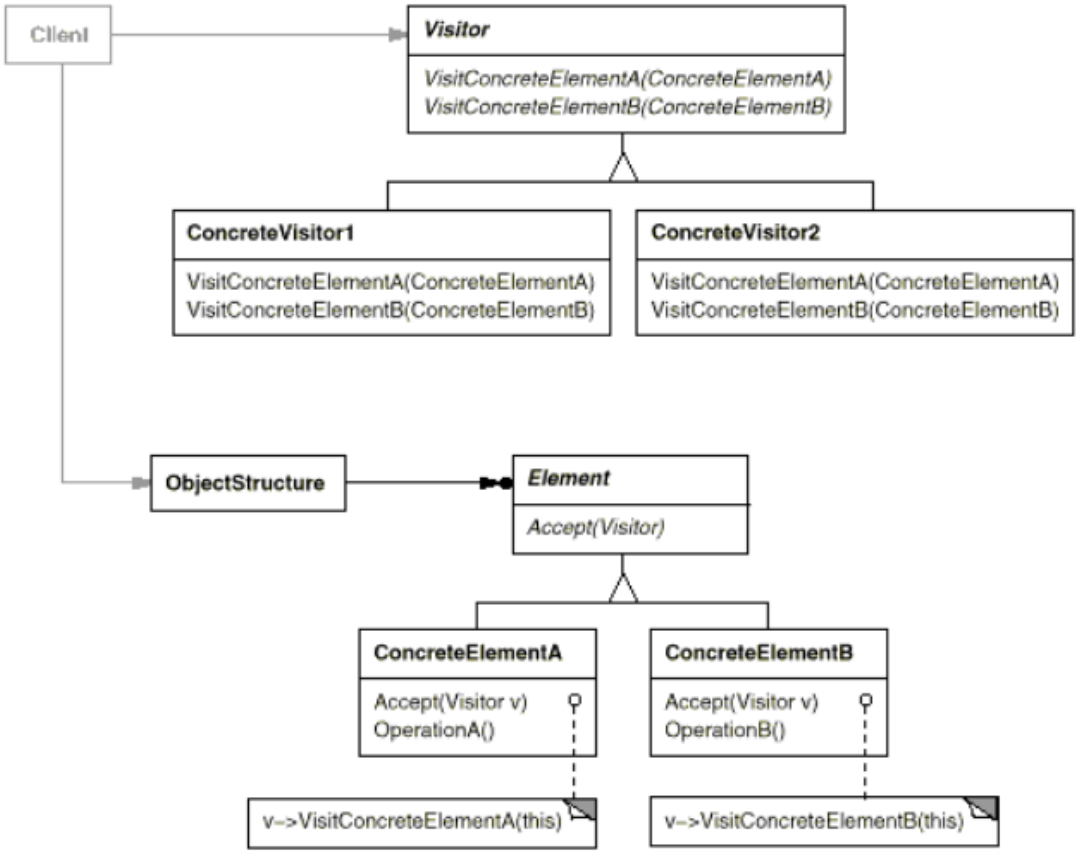

Figure 2.4: The Visitor pattern described in the Gang of Four book [7, p.369]

The visitor pattern is used for PADL models exploration because it is a convenient way to perform operations on objects within a given object structure depending on their concrete classes. It also allows doing unrelated and distinct operations on different object types without having to implement the visiting actions in each of their concrete classes, avoiding an unnecessary overhead. In addition, it enforces a consistent traversal of the models for all and any Visitor rather than to let Visitors decide how to traverse models.

PADL visitors allow third-party software to explore the models generated from the original source code. Those visitors are named `IGenerator` and `IWalker`. Listing 2.1 is an example of how a visitor could be used in a third-party software to visit a PADL model:

```
// Create the walker
final IWalker walker = new SomeWalkerImplementation();

// Create the PADL code level model from a PADL creator
final ICodeLevelModel codeLevelModel =
    Factory.getInstance().createCodeLevelModel(aName);
codeLevelModel.create(new SomePADLCreator(aSourceFileOrDirectory));

// Analyze the code level model
final IIdiomLevelModel idiomLevelModel = (IIdiomLevelModel) new
    AACRelationshipsAnalysis().invoke(codeLevelModel);

// Walk the model using the walker and display the results
idiomLevelModel.walk(walker);
System.out.println(walker.getResult());
```

Listing 2.1: Using visitors to explore a PADL model [3]

The following is a sequence diagram of the PADL visitors: when a program asks to walk a model, it opens the file from the computer's hard drive, then the model asks for a walk of the class, which is once again opened from the computer's hard drive, and the same process repeats for each method of the class, and each statement of each method.

Once everything has been visited, methods, classes and statements visitors are closed in this order. The figure 2.5 shows a sequence diagram of the PADL model visitor.

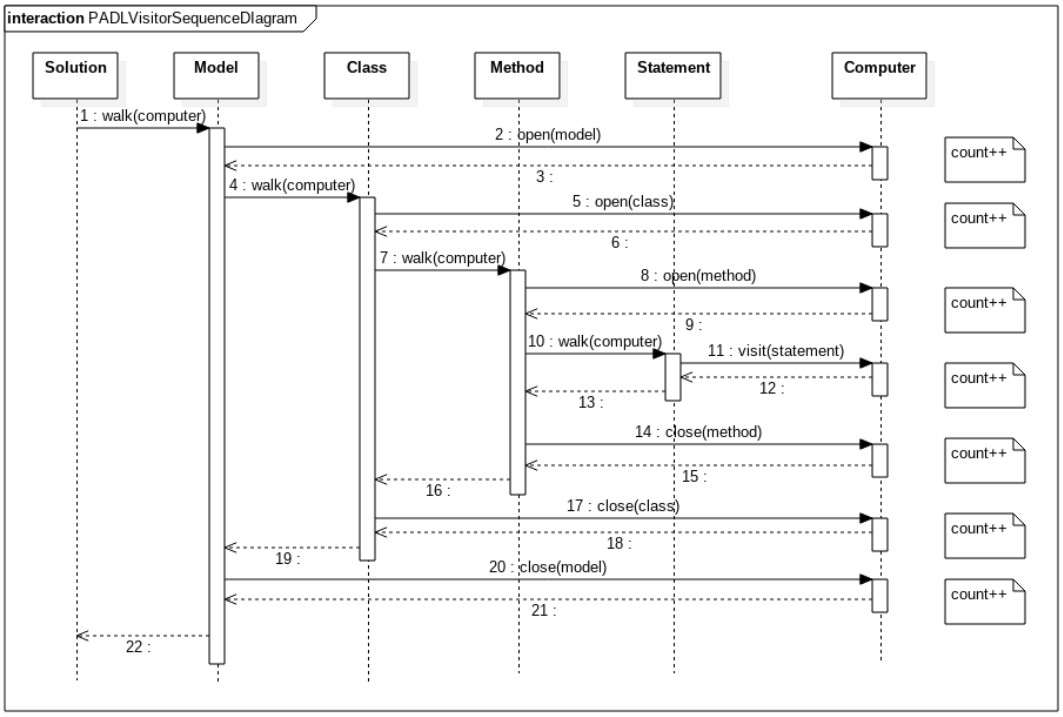

Figure 2.5: Sequence diagram of the PADL model visitor [5]

Chapter 3

The ANTLR4 Library

3.1 Context-free Languages

ANTLR, is a parser generator for context-free languages. It is provided as a Java library. Grammars can be classified in four categories. Figure 3.1 shows the scope of each type of grammar.

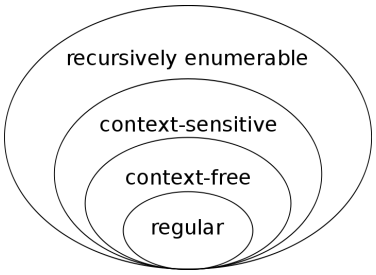

Figure 3.1: Chomsky Classification of Grammars

As shown in table 3.1, each grammar is associated to a type.

Grammar Type	Grammar Accepted	Language Accepted
Type 0	Unrestricted grammar	Recursively enumerable language
Type 1	Context-sensitive grammar	Context-sensitive language
Type 2	Context-free grammar	Context-free language
Type 3	Regular grammar	Regular language

Table 3.1: Chomsky's four Types of Grammars [12]

The type in which we are interested, that is, type 2, is constrained by the following: language must be such that A $\rightarrow$ x, where A is nonterminal and x is a string of terminal and/or non terminal.

In the formal language theory, a language can be defined as strings that are strained by some rules. Equivalently, some human languages such as English are based on groups of words that are separated by spaces. Thus, a valid sentence in the language must follow the rules of the grammar. Context-free languages are languages generated by context-free grammars [8].

3.2 LL(*) Parser

ANTLR parses input from left to right, performing a leftmost derivation. An LL parser is an LL(k) parser if it uses *k* tokens of lookahead when parsing an input. Lookahead means that the parser will "look ahead" of the stream to see what the next token(s) are, and will make a decision based on them.

An LL(*) parser is not restricted by the *k* number of lookahead tokens. LL parsing roughly corresponds to the Polish notation [6]. An example of the Polish notation is given in table 3.2. Thus, when given a parse tree, an LL parser will perform a pre-order traversal.

Notation Type	**Expression**
In-fix	1 + 2
Post-fix (Polish)	+ 1 2

Table 3.2: In-fix and Post-fix notation for the addition of 1 and 2

3.3 Parsers, Lexers, Tokens

A lexer, also known as a tokenizer, is applied first to the input the we want to parse. It explodes the input into tokens. Tokens are defined in the parser, each of them having a specific meaning and an unique identifier. They can be grouped by types; integers, identifiers or floating numbers for example.

The tokenization process is similar to how our brain reads English: we see sentences as a stream of words (tokens) rather than as a whole, at least for sentences that we are not used to see. Our brain looks at words individually, sorts them by type and recognizes the grammatical structure after that [10, p. 10].

Tokens are then parsed by the parser, which producing a parse tree [13] as shown by figure 3.2.

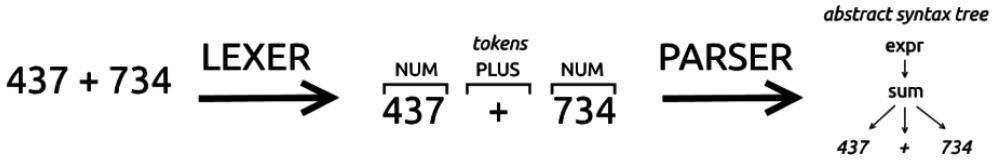

Figure 3.2: Data flow of a language recognizer [11]

3.4 Applications of ANTLR4

ANTLR4 is widely used to parse, validate, execute or process data. It is used by big names such as Twitter for its search engine, Hadoop or the NetBeans IDE [10, p. xi]. Its possibilities are diverse, from building JSON parsers, file readers configuration, wiki markup to doing DNA pattern matching [10, p. xi].

Using a grammar, ANTLR4 generates a parser that will be used to generate parse trees. Then, ANTLR4 provides a listener and a visitor that can be used to visit the trees.

3.5 ANTLR3 vs. ANTLR4

ANTLR4 uses an LL(*)-like parser, named Adaptive LL(*) or ALL(*), whereas ANTLR3 used a classical LL(*) parser. Unlike ANTLR3, ANTLR4 "performs grammar analysis dynamically at runtime rather than statically, before the generated parser executes" [10, p. xiii]. ANTLR4 also features an easier syntax for grammar rules and eliminates grammar ambiguities.

In ANTLR3, users had to extend the grammar by adding tree construction operations; this is no longer the case in ANTLR4, because it automatically creates a listener and a visitor that can be used to traverse the trees. Therefore, the grammar and the actions to perform are decoupled.

3.6 Using Parse Trees

3.6.1 Building Parse Trees

As explained above, parse trees are generated automatically by the ANTLR4 library's parser. In Java, the ANTLR4 classes used to apply the lexer, extract the tokens and apply the parser to generate the parse tree are: `CharStream`, `Lexer`, `Token`, `Parser`, `ParseTree` and `TokenStream` [10, p. 16]. Figure 3.3 depicts the correspondence between Java classes and ANTLR4 concepts.

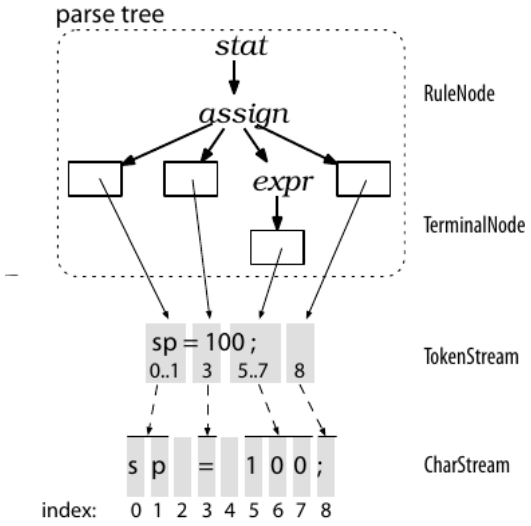

Figure 3.3: Correspondence between Java classes and parsing treatment in ANTLR4 [10, p. 16]

The memory footprint as reduced as much as possible by sharing data among ANTLR4 data structures. As shown in the diagram above, the `TokenStream` regroups tokens, and each token is made of one or more characters. Each token contains its characters, recording their start and stop indexes. No tokens are associated to white spaces [10, p. 16].

The following figure shows that `RuleNode` (subtree roots) and `TerminalNode` (leaf nodes) are extending `ParseTree`. `ParseTree` provides methods that are expected for a node of a tree, that are `getChild()`, `getParent()` and `getText()`. Therefore, `RuleNode` and `TerminalNode` provide them as well.

Specific Context classes are created by the library's parser: `StatContext`, `AssignContext`, and `ExprContext`. Context objects gather the information known about the recognition of a rule: start and stop tokens, sub-elements of the rule [10, p. 17]. Figure 3.4 shows how an expression is exploded into tokens.

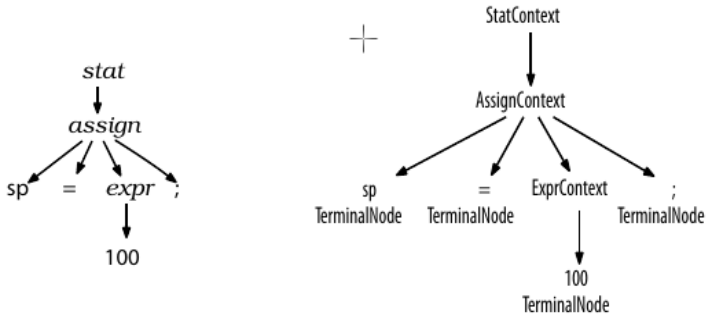

Figure 3.4: Parse Tree and Parse Tree Node Class Names [10, p. 17]

3.6.2 Traversing Parse Trees: Listeners and Visitors

We could write code for traversing the parse trees that ANTLR4 has generated for us, but there is no need to do so as the library has two built-in mechanisms to perform this action.

Listeners

The support of listeners is provided by the `ParseTreeWalker` and `ParseTreeListener` classes. Specific subclasses of those two classes are generated for each grammar file by ANTLR4. `ParseTreeWalker` walks the parse tree and trigger calls to listener, `ParseTreeListener`.

For a given rule, `assign` for example, the `ParseTreeListener` specific class will contain two function, `enterAssign` and `exitAssign`. Those will be triggered by the `ParseTreeWalker` when it finds the relevant nodes [10, p. 18]. The figure 3.5 shows those calls.

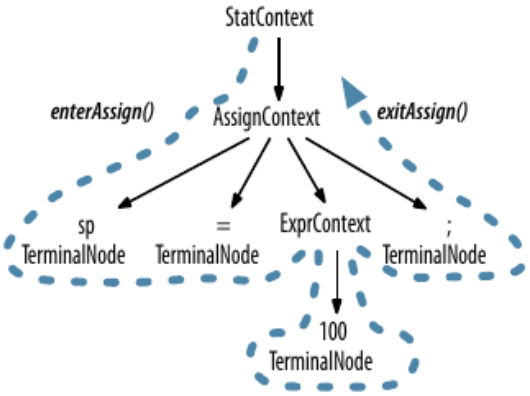

Figure 3.5: `ParseTreeWalker`'s depth-first traversal [10, p. 18]

The figure 3.6 shows the ParseTreeWalker call sequence for the previous example of $sp = 100$.

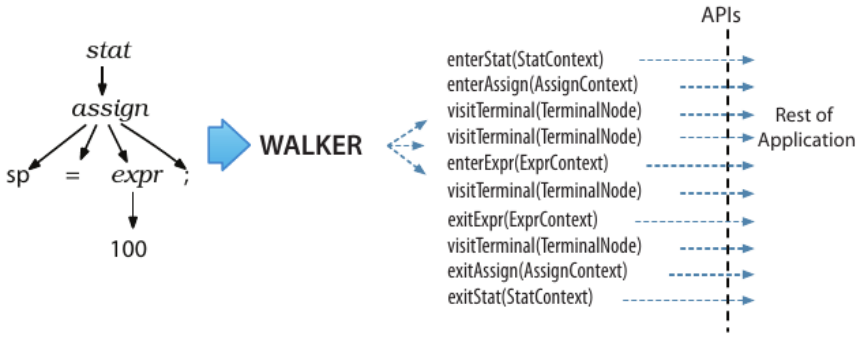

Figure 3.6: `ParseTreeWalker` call sequence [10, p. 19]

Visitors

Visitors give more control to the user than listeners, allowing them to control the walk of the parse trees by calling methods to visit children directly. Thus, one can bypass the call to a specific visit method if it is not needed. In the figure 3.7, one can see that when a visitor is launched, the method `visitStat` is first called, which is then calling the `visit` method with its children as arguments to keep walking [10, p. 19].

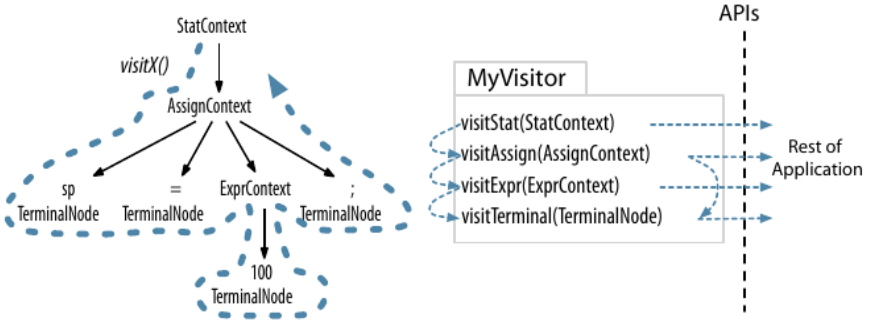

Figure 3.7: `ParseTreeVisitor` call sequence [10, p. 19]

The thick lines show the depth-first traversal of the parse tree while the thin lines show the visitor internal method call sequence [10, p. 19].

When generating a visitor, ANTLR4 creates an interface specific to the parser and an empty implementation that one can modify without having to manually override each method of the interface [10, p. 20].

Chapter 4

Parsing C# Code in the Ptidej Tool Suite with ANTLR4

4.1 Existing Work

Previous work had already been done to support C# parsing in the Ptidej Tool Suite. In 2009, Gerardo Cepeda Porras worked on the C# support and created four packages:

- `PADL Creator C# v1`
- `PADL Creator C# v1 Tests`
- `PADL Creator C# v2`
- `PADL Creator C# v2 Tests`

The first version was incomplete, and more of a feasibility study than a working product. However, studying the second version was very interesting: it was based on ANTLR3, version 3.2. The lexer, the parser and the tokens were generated automatically from the C# ANTLR3 grammar.

Since this was the way to do it in ANTLR3, the parse tree was traversed manually, using homemade recursive methods such as `findNextInherits`, `findNextSiblingOfType` or `findPreviousSiblingOfType`. They were complex and repetitive.

Those projects contained a test file and some test cases, which were complete and working out of the box. After an extensive review, I decided to use them for the new developments.

Overall, studying those packages has been helpful and I decided to follow the adage, "Do not reinvent the wheel" [4]. Instead of discarding all previous work as irrelevant, I chose to upgrade it incrementally. That is, upgrading from ANTL3 to ANTLR4 and making sure the unit tests do not fail.

4.2 Installing ANTLR4

I used the latest ANTLR4 version available at the time of the project, that is, 4.7.1 (released December 10, 2017) [9]. I downloaded the `antlr-4.7.1-complete.jar` and included it as a referenced library in the new PADL Creator C# v3 package.

4.3 Choosing Between a Listener and a Visitor

As explained previously, ANTLR4 offers two means to traverse the parse tree generated by the parser: a listener or a visitor. After looking at the possibilities and the behavior of each of those options, I decided to use the visitor. It is more flexible than the listener and suits the needs of the Ptidej Tool Suite better.

4.4 Finding an ANTLR4 C# Grammar

To generate the required parser, ANTLR4 needs a grammar. Fortunately, there exists an official one in the `antlr/grammars-v4` GitHub repository [1]. It offers two files, `CSharpLexer.g4` and `CSharpParser.g4` that will be used by the ANTLR4 library to generate the Java files (see next section).

4.5 Generating Lexer, Parser and Visitor

Once ANTLR4 is setup in the project, the next step is to generate the required files. Using the `antlr-4.7.1-complete.jar` file and the g4 grammar files mentioned above, we can generate the Java parser, lexer and visitor files. Listing 4.1 shows the commands we have to run:

```
java -jar antlr-4.7.1-complete.jar -visitor -no-listener CSharpLexer.g4
java -jar antlr-4.7.1-complete.jar -visitor -no-listener CSharpParser.g4
```

Listing 4.1: Commands required to generate the lexer, the parser and their visitors

By default, ANTLR4 generates a listener and no visitor. As we want the opposite, we use the options "-visitor" to generate the visitor and "-no-listener" to prevent the

listener generation.

When this is done, we have the following new files:

- `CSharpLexer.java`: lexer
- `CSharpParser.java`: parser
- `CSharpParserVisitor.java`: visitor interface
- `CSharpBaseVisitor.java`: default, empty implementation of `CSharpParserVisitor`

The lexer contains what is needed to tokenize the input: the list of tokens and rule names. The parser also has the tokens, the rules names. Actions can be performed when a rules is triggered, as seen in listing 4.2:

```
public class CSharpParser extends Parser {
   public static final int
      BYTE_ORDER_MARK=1, SINGLE_LINE_DOC_COMMENT=2, DELIMITED_DOC_COMMENT=3,
        ...
    public static final int
      RULE_compilation_unit = 0, RULE_namespace_or_type_name = 1, RULE_type = 2,
        ...
    public static final String[] ruleNames = {
      "compilation_unit", "namespace_or_type_name", "type", "base_type",
          "simple_type",
        ...
    }
}
```

Listing 4.2: The parser file generated by the ANTLR4 library for the C# grammar

The generated parser contains 17,625 lines of code.

The visitor interface contains 240 methods similar to `visitCompilation_unit`. They return the generic type T (the actual visitor) and take the context as parameter (see listing 4.3). The context has information about the current node, its parents and children.

```
/**
 * This interface defines a complete generic visitor for a parse tree produced
 * by {@link CSharpParser}.
 *
 * @param <T> The return type of the visit operation. Use {@link Void} for
 * operations with no return type.
 */
public interface CSharpParserVisitor<T> extends ParseTreeVisitor<T> {
   /**
    * Visit a parse tree produced by {@link CSharpParser#compilation_unit}.
    * @param ctx the parse tree
    * @return the visitor result
    */
   T visitCompilation_unit(CSharpParser.Compilation_unitContext ctx);

    ...
}
```

Listing 4.3: The interface of the visitor generated by the ANTLR4 library for the C# grammar

Finally, the parser itself is a default implementation of the interface. Every methods ends by calling `visitChildren`, which will visit the children of the current node. An excerpt of the visitor generated can be seen in listing 4.4.

```
/**
 * This class provides an empty implementation of {@link CSharpParserVisitor},
 * which can be extended to create a visitor which only needs to handle a
     subset
 * of the available methods.
 *
 * @param <T> The return type of the visit operation. Use {@link Void} for
 * operations with no return type.
 */
public class CSharpParserBaseVisitor<T> extends AbstractParseTreeVisitor<T>
    implements CSharpParserVisitor<T> {
   /**
    * {@inheritDoc}
    *
    * <p>The default implementation returns the result of calling
    * {@link #visitChildren} on {@code ctx}.</p>
    */
   @Override public T
       visitCompilation_unit(CSharpParser.Compilation_unitContext ctx) { return
       visitChildren(ctx); }

    ...
}
```

Listing 4.4: The visitor generated by the ANTLR4 library for the C# grammar

4.6 Implementing the C# PADL Parser

The C# PADL Parser is implemented in four packages:

- `padl.creator.csharpfile.v3`: contains the `CSharpCreator.java` file, entry point of the project
- `padl.creator.csharpfile.v3.parser`: contains the lexer, parser and its visitors
- `padl.creator.csharpfile.v3.parser.builder`: interface for the `BuilderContext` and `CodeBuilder`
- `padl.creator.csharpfile.v3.parser.builder.impl`: actual implementation of the builder

4.6.1 Instantiating the Project

The `padl.creator.csharpfile.v3.CSharpCreator` file is the first one to be called when parsing a new C# project. First, it creates the `ICodeLevelModel` from the `padl.kernel` package using the `ModelGenerator`, as shown in listing 4.5:

```
/**
 * Parses the given File(s) (should be a C# source file) and return it's
    modelized version.
 * Uses the ModelGenerator.
 * @param source either the File object representing the C# source file or a
    File object
 * representing a directory of C# source files.
 * @return the PADL model of the given C# source(s) file(s).
 * @throws CreationException
 * @throws java.io.IOException
 */
public static ICodeLevelModel parse(final String aSourceFileOrDirectory)
     throws CreationException {
  return ModelGenerator.generateModelFromCSharpFiles("C# Model",
      aSourceFileOrDirectory);
}

 /**
 * Create the CSharpCreator
 * @param aSourceFileOrDirectory source file or directory to be created as a
    String
 * Will be converted to File
 */
public CSharpCreator(final String aSourceFileOrDirectory) {
  this.source = new File(aSourceFileOrDirectory);
}
```

Listing 4.5: The first two functions used to parse C# projects in PADL

Then, in listing 4.6, the `ModelGenerator.generateModelFromCSharpFiles` function call the `create` function of the `CSharpCreator`:

```
/**
 * Perform 2 passes to create the CSharp model
 */
public void create(final ICodeLevelModel aCodeLevelModel)
      throws CreationException {
   try {
      // 1st pass that identifies the Classes and Interfaces
      if (this.source.isDirectory()) {
         for (final File input : this.source.listFiles()) {
            if (!input.isHidden()) {
               this.readFileFirstPass(input, aCodeLevelModel);
            }
         }
      }
      else {
         this.readFileFirstPass(this.source, aCodeLevelModel);
      }

      // 2nd pass that detects Interaction between Classes and Interfaces
      ...
   }
   catch (final IOException | RecognitionException e) {
      e.printStackTrace(ProxyConsole.getInstance().errorOutput());
      throw new CreationException(e.getMessage());
   }
}
```

Listing 4.6: The first two functions used to parse C# projects in PADL

Here, we perform two passes on the project to simplify the work of the parser. The first pass takes care of identifying the definition of classes and interfaces, whereas the second pass does the rest of the work. Each pass means a different visitor. Finally, 4.7 shows how a pass is performed:

```
/**
 * Setup a pass: create the lexer and the parser
 * @param source
 * @param aCodeLevelModel
 * @return
 * @throws IOException
 * @throws RecognitionException
 */
private CSharpParser setUpPass(final File source, final ICodeLevelModel
    aCodeLevelModel)
      throws IOException, RecognitionException {
   // sanity check
   if (source == null || !source.exists() || source.isDirectory()) {
      throw new IOException("Cannot find C# source files in " + source);
   }
   final InputStream in =
        new FileInputStream(source);
   final CSharpLexer lexer = new CSharpLexer(CharStreams.fromStream(in,
       StandardCharsets.UTF_8));
   return new CSharpParser(new CommonTokenStream(lexer));
}

/**
 * Perform the first pass
 * @param source
 * @param aCodeLevelModel
 * @throws IOException
 * @throws RecognitionException
 */
private void readFileFirstPass(final File source, final ICodeLevelModel
    aCodeLevelModel)
      throws IOException, RecognitionException {
   final CSharpParser parser = this.setUpPass(source, aCodeLevelModel);
   final CSharpParserVisitor<?> visitor =
        new CSharpParserInitialVisitor<Object>(aCodeLevelModel);
   this.visitParseTree(parser, visitor);
}
```

```
 /**
 * Visit the parse tree generated by the parser
 * @param parser
 * @param visitor
 */
private void visitParseTree(final CSharpParser parser, final
    CSharpParserVisitor<?> visitor) {
   final ParseTree tree = parser.compilation_unit();
   visitor.visit(tree);
}
```

Listing 4.7: Performing a pass in the CSharpCreator

4.6.2 Parsing the Project

In the `padl.creator.csharpfile.v3.parser`, the lexer, parser and visitor generated by the ANTLR4 library. In this part, we will only focus on code that was specifically written for the project and not generated code, which has already been described above. I have created two additional visitors extending the default `CSharpParserBaseVisitor` one. This way, there is a single visitor per pass, without having to duplicate the code and I can simply override the methods that I need.

Initial Visitor

For the first pass, the visitor, `CSharpParserInitialVisitor`, overrides the `visitClass_definition` and `visitInterface_definition` functions. We want to get the name of the classes and interfaces as well as their visibility (private, protected, public), as listing 4.8 shows:

```
@Override public T
    visitClass_definition(CSharpParser.Class_definitionContext ctx) {
   try {
      // Initial reader
      final CodeBuilder builder = new InitialClassBuilderImpl(null);
      final BuilderContext builderContext = new BuilderContext(model);
      builder.create(ctx, builderContext);
      this.model.addConstituent((IConstituentOfModel) builder
            .close());
   } catch (CreationException e) {
      e.printStackTrace();
   }
   return visitChildren(ctx);
}
```

Listing 4.8: Excerpt of the initial visitor

Advanced Visitor

In the second pass, the visitor, `CSharpParserAdvancedlVisitor`, focuses on the relationships between classes and interfaces. It overrides the following functions:

- `visitClass_definition`
- `visitInterface_definition`
- `visitConstructor_declaration`
- `visitMethod_declaration`
- `visitField_declaration`
- `visitInterface_member_declaration`

ANTLR4 makes it easy to know at what moment the function is triggered by the visitor. The functions follow a general scheme: first, create the appropriate `CodeBuilder` (`ClassBuilderImpl` for example), then a `BuilderContext`, then run the `Builder` using the `BuilderContext` and the context given by ANTLR4. This is shown in listing 4.9.

```
@Override public T
    visitClass_definition(CSharpParser.Class_definitionContext ctx) {
  try {
    // Second pass reader
    final CodeBuilder builder =
      new ClassBuilderImpl(
        this.codeElements.isEmpty() ? null
            : this.codeElements.peek());
    final BuilderContext builderContext = new BuilderContext(model);
    builder.create(ctx, builderContext);
    this.codeElements.push(builder);
  } catch (CreationException e) {
    e.printStackTrace();
  }
  return visitChildren(ctx);
}
```

Listing 4.9: Excerpt of the advanced visitor

4.6.3 Implementing the Builder

The `padl.creator.csharpfile.v3.parser.builder.impl` package is where the work gets done. It contains the following classes:

- `AbstractClassBuilderImpl`: abstract class, extended by `InitialClassBuilderImpl` and `InitialInterfaceBuilderImpl`. Contains the `findNextInherits` method to add inherited entities or interfaces to the PADL model.
- `AbstractPADLCodeBuilder`: implements the `CodeBuilder` interface, extended by `AbstractClassBuilderImpl`, `ClassMemberBuilderImpl`, `InterfaceMethodBuilderImpl`, `MethodBuilderImpl`.
- `ClassBuilderImpl`: creates a new class in the PADL model.
- `ClassConstructorBuilderImpl`: extracts the information of the constructor of a class to insert them into the PADL model.
- `ClassMemberBuilderImpl`: extracts the information of a class member (also known as attribute in Java) to insert them into the PADL model.
- `InitialClassBuilderImpl`: extracts the basic information about a class: name, visibility and abstraction.
- `InitialInterfaceBuilderImpl`: extracts the basic information about an interface: name and visibility.
- `InterfaceBuilderImpl`: extracts the information about an interface: name, visibility and inheritance.
- `InterfaceMethodBuilderImpl`: extracts the information about the methods offered by an interface: name, arguments, return type.
- `MethodBuilderImpl`: extracts the information about the methods implemented by a class: name, arguments, return type.
- `Util`: a collection of static functions used throughout the project.

The figure 4.1 shows the UML class diagram for the builder implementation.

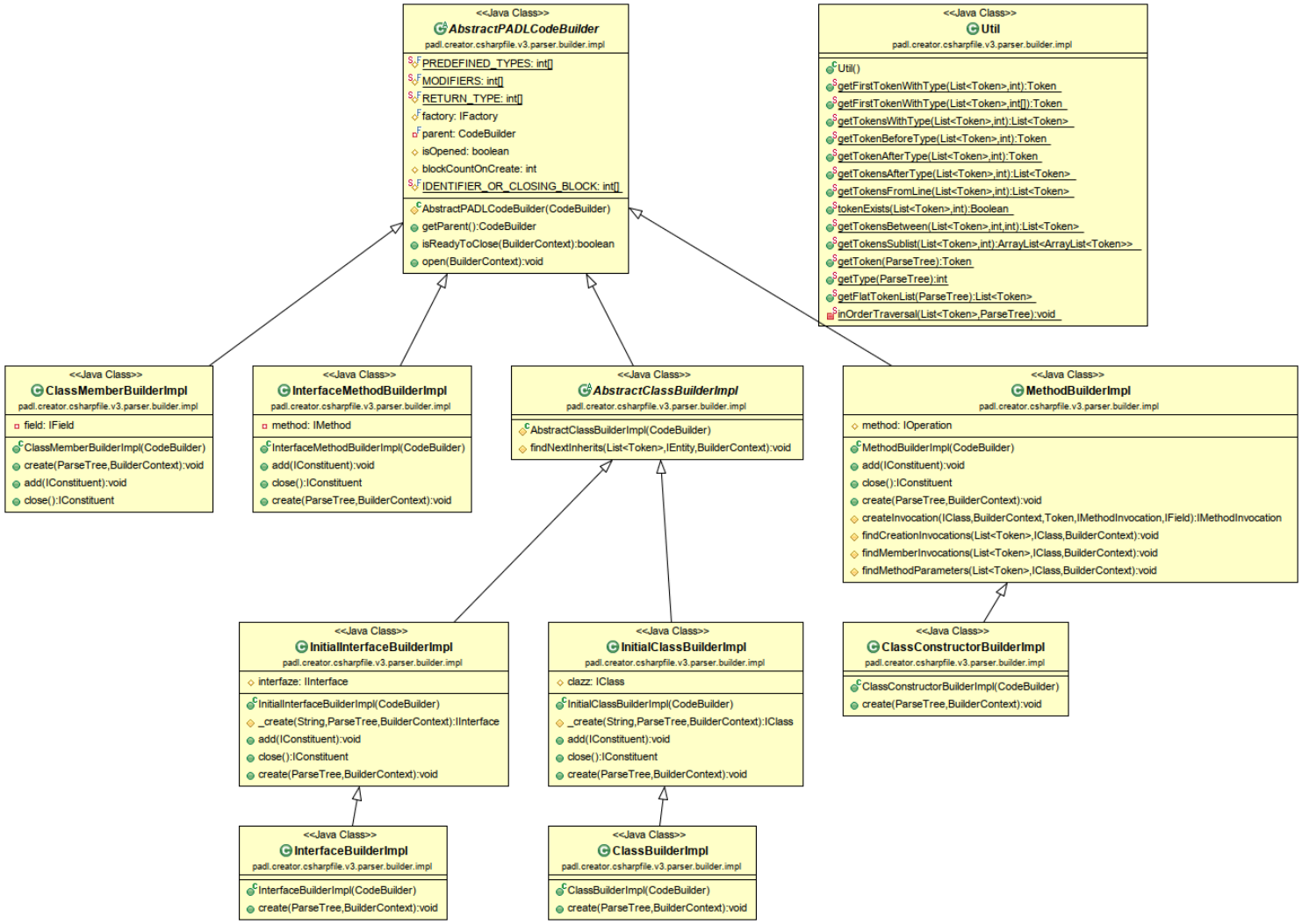

Figure 4.1: Builder Implementation UML Class Diagram

In the following, I will not detail all the functions of the package, but some patterns.

A fundamental concept of this project is to extract the list of the "futures" tokens, that is, child tokens of the current node. The current node is given by the `ctx` argument coming from the visitor, but only contains the first level of children. Thus, in listing 4.10, I use a custom in-order traversal of the current node's children and flatten them into a list of Tokens, in the `getFlatTokenList` function:

```java
/**
 * Retrieves all Tokens from the {@code tree} in an in-order sequence.
 * @param tree the parse tee to get all tokens from.
 * @return all Tokens from the {@code tree} in an in-order sequence.
 * https://stackoverflow.com/a/22770561
 */
public static List<Token> getFlatTokenList(ParseTree tree) {
    List<Token> tokens = new ArrayList<Token>();
    Util.inOrderTraversal(tokens, tree);
    return tokens;
}
/**
 * Makes an in-order traversal over {@code parent} (recursively) collecting
 * all Tokens of the terminal nodes it encounters.
 * @param tokens the list of tokens.
 * @param parent the current parent node to inspect for terminal nodes.
 */
private static void inOrderTraversal(List<Token> tokens, ParseTree parent) {
    // Iterate over all child nodes of ‘parent‘.
    for (int i = 0; i < parent.getChildCount(); i++) {
        // Get the i-th child node of ‘parent‘.
        ParseTree child = parent.getChild(i);
        if (child instanceof TerminalNode) {
            // We found a leaf/terminal, add its Token to our list.
            TerminalNode node = (TerminalNode) child;
            tokens.add(node.getSymbol());
        }
        else {
            // No leaf/terminal node, recursively call this method.
            inOrderTraversal(tokens, child);
        }
    }
}
```

Listing 4.10: Excerpt of the `Util` class: getting a flat list of tokens

Listing 4.11 is an example of how this function is used; in the `InitialInterfaceBuilderImpl`, an interface is added to the PADL model by extracting its name:

```
public void create(final ParseTree node, final BuilderContext context)
     throws CreationException {

   final List<Token> tokens = Util.getFlatTokenList(node);

   final Token interfaceNameElement = Util.getFirstTokenWithType(tokens,
       CSharpParser.IDENTIFIER);
```

Listing 4.11: Excerpt of the InitialInterfaceBuilderImpl class: creating an interface for PADL

`CSharpParser.IDENTIFIER` means the `IDENTIFIER` token present in the `CSharpParser`, which is used for class or function names. The `getFirstTokenWithType` function is a simple loop on the token list, as shown in listing 4.12:

```
public static Token getFirstTokenWithType(List<Token> tokens, int
    tokenType) {
   for (Token token: tokens) {
      if (token.getType() == tokenType) {
            return token;
        }
   }
   return null;
}
```

Listing 4.12: Excerpt of the `Util` class: getting the first token of a given type

4.7 Testing the Project

To test the correct behavior of the project, I used the test file that was already available in the version 2 of the C# parser package. It consists of 8 assertions of the PADL model generated by the parser when it is given seven C# files: two plain, simple classes; an interface and its implementation; several inheritances; constructor, destructor and attributes; a switch case. The test function is as seen in listing 4.13:

```
public void testParser() throws CreationException {
      final ICodeLevelModel model =
         CSharpCreator.parse("../PADL Creator C# v3 Tests/rsc/parser_oracles");

      // make sure we got our right number of classes
      assertEquals(11, model.getNumberOfConstituents());

      // make sure we got the 'Line' class
      assertNotNull(model.getConstituentFromName("Line"));
      // make sure the superclass was found
      assertNotNull(((IClass) model.getConstituentFromName("Line"))
         .getInheritedEntityFromName("DrawingObject".toCharArray()));

      // make sure we got the interface
      assertNotNull(model.getConstituentFromName("IMyInterface"));
      // make sure the implementation was found
      assertNotNull(((IClass) model
         .getConstituentFromName("InterfaceImplementer"))
         .getImplementedInterface("IMyInterface".toCharArray()));

      // make sure we got the class member of Outputclass
      assertTrue(((IClass) model.getConstituentFromName("OutputClass"))
         .doesContainConstituentWithName("myString".toCharArray()));

      // make sure we got the method and parameter 'myChoice' of method
          makeDecision
      assertTrue(((IClass) model.getConstituentFromName("MethodParams"))
         .doesContainConstituentWithName("makedecision".toCharArray()));
      assertNotNull(((IMethod) ((IClass) model
         .getConstituentFromName("MethodParams"))
         .getConstituentFromName("makedecision"))
         .getConstituentFromName("myChoice"));
   }
```

Listing 4.13: Excerpt of the `TestCreatorCSharpv3` class

4.8 Challenges Encountered

The first challenge I encountered was to understand ANTLR4's logic. At the beginning of the project, I was not very familiar with ANTLR in general. I read a few tutorials on the Internet, but nothing came close to reading Terence Parr's "The Definitive ANTLR 4 Reference", which I have heavily used to describe ANTLR4 in Chapter 2. I recommend reading the book to understand the underlying mechanisms of ANTLR4, as well as seeing the library in action: many real-life applications are available in the book.

Then, when I fully understood how the library worked, the next challenge was to find the right visitor in which to hook the actions. This is what took most of my time. Even though the function names are self-explanatory, sometimes, the information required by PADL were at a higher level of the parse tree, or way below the current level.

I solved the first problem by calling the parent of the actual context node, or even higher up; for the `visitMethod_declaration`, taking care of creating the methods in the PADL model, I called the `ctx.parent.parent`, and even the `ctx.parent.parent.parent` for the `visitField_declaration` method that creates fields (attributes) in the PADL model. This is safe to use because if those methods are triggered, we know that the class contains fields for example.

For the second issue, traversing the descending nodes, I came across the "token flattenization" technique while looking for answers on StackOverflow, and it struck me: this in-order traversal generating a list of tokens was the easiest way to access all the descending tokens, since by default ANTLR4 only gives direct access to the children of a node and not the grand children, their children and so on unlike ANTLR3. To extract basic information of "close" tokens (that is, not more than a few levels down) because the context already reduces the number of tokens, this was the best solution.

4.9 Future Work

While the unit tests are passing, I am not entirely satisfied with them. I think they should be extended and more test cases should be added. To do so, one could implement some of the design patterns from the Gang of Four book in C# and test them using unit tests.

We could also systematically each feature of the C# language, and integrate those that are not supported.

With the current test files, it takes about two seconds to run the ANTLR parser and generate the PADL model. We should see if it can scale to accommodate bigger code bases.

Preprocessor directives support could be added. Those directives begin with a # symbol and are similar to C's and C++'s ones. There exists a grammar in the same repository as the one I have used for those directives that could be used.

Next, once the reliability of the package has been proven, it should be added to the Ptidej Tool Suite UI so that regular users can use it directly from the user interface without having to write code to run it.

Chapter 5

Conclusion

Overall, this has been a very interesting project. I learned a lot about ANTLR, parsing and lexical analysis in general. I did not really think about how IDE's, code editors or compilers could do syntax parsing before, but now I can see how it can be done.

I am also now much more familiar with the visitor pattern, that I did not apply in real life until now.

This project was also the occasion to discover in details the Ptidej Tool Suite, that I will probably use in the future to analyze software I want to be more familiar with.

Similarly, coming from an IntelliJ Idea background, this project brought me to the Eclipse world, that I now master.

If you want to check it out, my work is now in the official repositories of the Ptidej Tool Suite, in BitBucket and GitHub.

In case you have questions, comments, suggestions or have found a bug, please do not hesitate to contact me. You can find my contact details below.

Adrien Poupa
adrien@poupa.fr
https://adrien.poupa.fr

Bibliography

[1] Christian Wulf et al. *ANTLR4 C# grammar*. https://github.com/antlr/grammars-v4/tree/master/csharp. 2018.

[2] Yann-Gaël Guéhéneuc et al. *How to Download, Install, and Contribute to the Ptidej Tool Suite*. https://wiki.ptidej.net/doku.php?id=welcome_package_for_new_members. 2018.

[3] Yann-Gaël Guéhéneuc et al. *PADL*. https://wiki.ptidej.net/doku.php?id=padl. 2018.

[4] Jeff Atwood. *Don't Reinvent The Wheel, Unless You Plan on Learning More About Wheels*. https://blog.codinghorror.com/dont-reinvent-the-wheel-unless-you-plan-on-learning-more-about-wheels/. 2009.

[5] Yann-Gaël Guéhéneuc. *Understanding PADL Visitor*. http://www.ptidej.net/team/photos/180306-UnderstandingPADLVisitor/plfng_view. 2018.

[6] Josh Haberman. *LL and LR Parsing Demystified*. http://blog.reverberate.org/2013/07/ll-and-lr-parsing-demystified.html. 2013.

[7] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[8] Karleigh Moore and Alex Chumbley. *Context Free Languages*. https://brilliant.org/wiki/context-free-languages/. 2018.

[9] Terence Parr. *Download ANTLR*. http://www.antlr.org/download.html. 2018.

[10] Terence Parr. *The Definitive ANTLR 4 Reference*. 2012.

[11] Gabriele Tomassetti. *The ANTLR Mega Tutorial*. https://tomassetti.me/antlr-mega-tutorial. 2017.

[12] tutorialspoint.com. *Chomsky Classification of Grammars*. https://www.tutorialspoint.com/automata_theory/chomsky_classification_of_grammars.htm. 2018.

[13] Wikipedia. *Lexical Analysis*. https://en.wikipedia.org/wiki/Lexical_analysis. 2018.