Universidade Federal do Pampa

Introduction to Variability Mechanisms in Java

Marcelo Schmitt Laser

April 23, 2017

Introduction

In order to properly implement the design decisions made for the PLeTs v4 project, it is necessary that all professionals involved be aware of certain mechanisms, both theoretical and practical, that will be used in its design and development. This document is aimed to serve both as a basis for the study and explanation of these techniques and as a guide for the appropriate representation and implementation of them in PLeTs v4.

In Chapter 1 we will present some variability management mechanisms, as well as explain which ones to use, when each is appropriate and why some should not be used. First a basic theory of the concept is presented, based on [1]. The mechanisms of using Parameters, Design Patterns, Frameworks, Components and Services, and Preprocessors are presented. Then, the idea behind the particular variability mechanisms of PLeTs v4 are presented, with their in-depth descriptions being left to 2.

In Chapter 2 we will present the basics of the software design patterns and architectural patterns used in PLeTs v4, with a focus on explaining their goals and implementation. First the Factory Method design pattern will be presented [7] alongside the data structure catalogue system implemented in PLeTs v3. We then briefly expose the basic concepts of Software Connectors [12], with a particular emphasis on Linkage and Arbitrator connectors. With a basis on these concepts, we then proceed to present the architectural pattern proposed for the composition of products in PLeTs v4, which is largely based on the Distributed Objects architectural style [8].

Chapter 3 is meant to serve as a guide to the components designed within PLeTs v4 and their specifications. This chapter is focused on the architectural characteristics of these components, and is not meant to serve as a guide to the software testing methods applied in PLeTs.

Chapter 4 is reserved for track keeping of the evolution of the PLeTs v4 architecture project. It is my intention that this chapter be constantly updated and kept as a snapshot of the current state of the project. Any impreciseness or incorrectness in this chapter is to be taken as architectural erosion, and must be corrected as soon as possible.

Appendix A is reserved to the study of basic concepts of Formal Languages and Compilers, and is left as an aid for the development of parsers and script generators for PLeTs v4.

Should this document at any point prove too dense, complicated, unclear

or in any way lacking, I ask that you inform me (marcelo.laser@gmail.com) as quickly as possible so that the problem may be resolved and the document may be kept in the best possible shape. Please bear in mind that this document is to serve as the primary guide of the PLeTs v4 infrastructure, and therefore it aims to be as clear and precise as possible.

Marcelo Schmitt Laser

Acronyms

BNF Backus-Naur Form. 26

CBSE Component-Based Software Engineering. 7

CORBA Common Object Request Broker Architecture. 16

SPL Software Product Line. 7, 29

SPLE Software Product Line Engineering. 7

Glossary

binding time The moment during which the properties of a software artifact are configured. Typically, these may be: compile-time, where the configuration is handled during the compilation process; link-time, where the configuration is handled at the moment of dependency solution and access to external libraries; load-time, where the configuration is handled the moment the software product is instanced (loaded), and; run-time, where the configuration is handled during the software's execution. 7

connector From Taylor et al. [12]: "An architectural element tasked with effecting and regulating interactions among components.". 15

grammar The set of rules that define the structure of all possible strings within a language. For any one string (regardless of size) to belong to a particular language, it must match that language's grammar. Unless stated otherwise, this document uses the terms *grammar* and *context-free grammar* interchangeably. More information on context-free grammars and grammars in general can be found in [5]. 8, 22, 26, 29

regular expression A sequence of characters that represents a string pattern. A regular expression has several uses, among which the ones we are most interested in are searching a stream of text for matching strings and validating a string against a pre-determined pattern. More information on regular expressions can be found in [5]. 22–26

variability From SEI [2]: "Variability is the ability of a system, an asset, or a development environment to support the production of a set of artifacts that differ from each other in a preplanned fashion.". 7–9

variant From SEI [2]: "The realization of a variable part, meaning the result of exercising the variation mechanism(s), is called a variant.". 7, 8, 10–14

variation point Also known as variable part, it is a well-defined point in the SPL where modifications may be introduced [2].. 11–13

Contents

Chapter 1

Variability Mechanisms

In order to control variability, it is important to have mechanisms in place to manage that variability in terms of configuration (both in terms of products and in terms of modules), traceability (identifying portions of code and linking them to models), solution building (again, both in terms of products and in terms of modules), access security (management of code-access permissions), non-functional properties (managing variability in qualitative terms such as performance and reliability) and several other concerns that exist in the fields of Software Engineering that deal with variability, such as Component-Based Software Engineering (CBSE) and Software Product Line Engineering (SPLE).

Variability mechanisms can be considered to have a "binding time" (such as compile, link, load, run, etc.). The mechanisms presented in this chapter have varying binding times, but for simplification purposes, we consider them to be either bound at compile-time, load-time or run-time. While PLeTs v3 was primarily bound during compile-time (with run-time binding being theoretically possible but untested), the new architecture aims to permit a more varying binding time so as to allow the evolution into a *dynamic* SPL.

Section 1.2 re-evaluates the primary mechanism of PLeTs v1, component interfaces, as well as presenting a new form of using them for selecting variants automatically in PLeTs v4. Section 1.1 presents a parameter-based form of variability management in Java, relying on run-time conditionals. Section 1.3 shows how to use Java code reflection to implement variability points using a variation of the Factory Method [4]. Section 1.4 presents the javapp tool [6] to include preprocessing directives to Java in the fashion of C directives. Finally, Section 1.5 briefly presents the Ant build system, which can be used to automate the building of PLeTs v4 products.

1.1 Parameters

One way to get around the problem of Java not having a preprocessor is by abusing the optimization features that Java compilers do generally have. In the

standard Java compilers, any code that is unreachable is automatically removed by the compiler's optimizer, which means that we are able to remove certain parts of code by making sure they can never be executed. Essentially, we create conditions of the form if (true) and if (false) to encapsulate our variant code.

To do this, one can use what is called a Static Final attribute. When a conditional loop evaluates itself with a single boolean attribute, and that attribute is always false, that loop will be removed by the optimizer due to being unreachable. Usually the compiler cannot know if the condition will always be false or not, given that this would require a complex analysis of a state machine, but since a Static Final attribute can never change its value, the optimizer will be able to tell. Listings 1 and 2 show an example of how this can be used.

```
1  public class Preprocessor
2  {
3   public static final boolean FULLACCESS = false;
4    ...
5  }
```

Listing 1: Preprocessor.java, encapsulating variability points

In the example, a Preprocessor class is declared in Listing 1 which contains all of the variability point constants for free access from the rest of the project's packages; essentially, we are centralizing the variability management in a single artifact. The packages may then use these attributes in conventional conditional loops to transform them into conditional compilation loops. In Listing 2, the method getEnd1 is conditionally compiled so that if the selected product has full access (a variability point of the AutoREST application [10] that represents security access level), a reference to end1Element is returned. Meanwhile, if the selected product does not have full access, only a copy is returned, ensuring that the object cannot be modified from outside. This ensures that if the product does not have full access, the user will not be able to directly alter the end1Element variable, being forced to use other methods to do so.

This approach has one very important limitation: it can only be used where normal conditional loops may be used. Therefore, it is impossible to conditionally compile an entire method or class, as Java does not permit conditional loop statements to be outside method bodies (remember grammars: the rule for *if* statements would be inside the rule for *methods*, meaning that the syntactic analyser would only accept the token "IF" when it is found inside a method). While it works in the example above, it does work in the example presented in Listing 3.

In this example, an entire constructor should be conditionally compiled so that this particular overload is only available if the configured product has full access. While it is possible to place the method's inner code inside a conditional loop, that would mean this overload would still exist, and a programmer using

```
1  import Preprocessor;
2
3  public class UmlAssociation extends UmlElement implements
   ↪  Serializable
4  {
5  ...
6    private UmlElement end1Element;
7  ...
8    public UmlElement getEnd1()
9    {
10     if(Preprocessor.FULLACCESS)
11     {
12       return this.end1Element;
13     }
14     else
15     {
16       return DeepCopy.copy(this.end1Element);
17     }
18   }
19 ...
20 }
```

Listing 2: Variability point being resolved by the Preprocessor.java attributes, taken from [10]

an IDE like NetBeans or Eclipse would be fooled into thinking it is available, when in fact it is not. The ideal way of solving this problem would be to remove the method entirely from the compiled .jar file, so that the programmer is not tricked into thinking the method exists.

The Static Final form of variability management is perfectly valid for situations where a single method may have two different implementations depending on product configuration. It is a simpler way of controlling variability than the other approaches discussed ahead, but it must be used with caution. An especially dangerous effect of using Static Final attributes for variability management is that the documentation (Javadoc) cannot be conditionally compiled using this approach, making it virtually impossible to document a variable method properly.

In order to get around variability situations like the one in Listing 3, where a method's availability is dependent on product configuration, it is possible to use one of two other approaches: Java interfaces and Java code reflection,.

```
1  import Preprocessor;
2
3  public class UmlAssociation extends UmlElement implements
   ↪  Serializable
4  {
5  ...
6    //INCORRECT JAVA SYNTAX, DOES NOT COMPILE!!!!!
7    if(FULLACCESS)
8    {
9      public UmlAssociation()
10     {
11       BuildIt(null, null, null, null, null, null, null,
   ↪  null, null);
12     }
13   }
14 ...
15 }
```

Listing 3: Limitations of Static Final attributes, taken from [10]

1.2 Component Interfaces as Variant-Selection Mechanisms

One viable solution for the problem of selectively enabling methods raised at the end of Section **??** is to encapsulate the public interface of a class through use of explicit interfaces. Through the Separated Interface pattern [3], we are able to make packages with the express purpose of encapsulating variants, choosing which parts of a class (usually a facade) will be available to the rest of the system.

The basic notion of the Separated Interface pattern is for the system to only access a package's interface, with the implementation being plugged in through a factory.

1.3 Factory Method with Reflection

While compilation directives are not readily available in Java, it is possible to create a mechanism equal to (and perhaps simpler than) the Variant Factory proposed in [7] by using code reflection. Code reflection[1] is a type of mechanism known as *introspection* where software artifacts are able to analyse and modify themselves during runtime, enabling them to alter their own behaviour based on the software's current conditions. Reflection is one of the oldest programming

[1] Written with sizeable contributions from Anderson Domingues

```
1  package ParserPackageA;
2
3  public interface ParserA
4  {
5    Object method1();
6  }
```

Listing 4: Interface making only method1 available.

```
1  package ParserPackageB;
2
3  public interface ParserB
4  {
5    Object method2();
6  }
```

Listing 5: Interface making only method2 available.

concepts, dating back to the notion of *self-modifying code* which was present in the earliest assembly language codes. Back then, the idea was basically to keep code inside the data portions of a software and then pull them into the processing portions, essentially changing the execution of the program. Of course, this caused several maintainability problems back then, and occasioned numerous bugs resulting from unexpected runtime modifications. Eventually, the practice was dropped entirely and labeled as bad practice.

The modern notion of reflection is slightly different from self-modifying code. Control mechanisms are set in place in modern programming languages that implement introspection, and the degree to which a program may be modified during runtime is greatly reduced to avoid unexpected behaviour or "write-only" code. Among these mechanisms is the restriction that is placed on reflection to be used only to alter runtime behaviour, rather than making permanent modifications to executables and libraries. Essentially, this means reflection cannot alter code or executables. One classic example of reflection is the implementation of generics (such as in List<E >), where the language identifies the object inside a generic reference by effectivelly asking it for its type.

The process of creating a Variant Factory in Java using reflection is similar to that used in C#: an interface is created to represent a variation point, declaring its public signature, and a Factory class is used to instantiate variants based on that interface. In this implementation, however, we simplify the code related to choosing a variant by using reflection to analyse the available variants for a suitable one. This both removes the need to alter the Factory code when a

```
1  package ConcreteParserPackage;
2
3  import ParserPackageA.ParserA;
4  import ParserPackageB.ParserB;
5
6  public class ConcreteParser extends ParserA, ParserB
7  {
8    Object method1()
9    {
10     ...
11   }
12
13   Object method2()
14   {
15     ...
16   }
17 }
```

Listing 6: Concrete Parser implementation.

variant is created and makes it possible for the system to use a single Factory to realize all variants.

First, the interface for a particular variation point is implemented in its own isolated deployment package, so as to make any dependencies to it restricted and well-controlled. This interface must specify all of the necessary public methods of any variant that implements this variation point, and may be declared with default behavior when appropriate (see Listing 8).

Next, the variants must be created, also in their own deployment packages, with a facade that implements the variation point interface. Listing 9 shows an example of this.

At this point, all of the necessary components are in place for us to build a Variant Factory. Listing 10 presents an example of a Variant Factory implementation. In particular, this implementation uses the notion of *constraints* to select which available variant best satisfies the request. This and other improvements on the Variant Factory pattern will be reviewed in Chapter 2. For now, focus on the use of reflection and assume that the statement `i. satisfies (v)` always evaluates to true.

First, the code in Listing 10 acquires a list of all the variants of the variation point "Parser". Then, iterating over that list, it evaluates each variant against the constraints imposed by the requesting module, and if a feature satisfies those constraints, an instance of that variant is returned.

By using reflection in this manner, we have solved the problem of selecting a variant represented by an entire class, which was not possible by using a Static

```
1  package ParserFactoryPackage;
2
3  import ConcreteParserPackage.ConcreteParser;
4  import ParserPackageA.ParserA;
5  import ParserPackageB.ParserB;
6
7  public class ParserFactory
8  {
9    public Parser getInstance()
10   {
11     return new ConcreteParser();
12   }
13 }
```

Listing 7: Factory to plug ConcreteParser into the available interface.

```
1  package ParserVariationPoint;
2
3  public interface IParser
4  {
5    default Object parse(Object input)
6    {
7      return null;
8    }
9  }
```

Listing 8: Example of variation point interface with default behavior

Final attribute. We are now able to implement the exact same capacities of the Variant Factory pattern described in [7], and therefore have a way of selecting variants dynamically.

1.4 Java Preprocessor

1.5 Ant Build System

```java
package ParserX;

import ParserVariationPoint.IParser;

public class ParserX implements IParser
{
  public Object parse(Object input)
  {
    //execute
  }
}
```

Listing 9: Example of variant facade

```java
package ParserFactory;

import ParserVariationPoint.IParser;

public class ParserFactory
{
  public static Object getInstanceOf(Configuration f,
  ↪ Constraints i) throws Exception
  {
    List<Feature> l = f.getFeature("Parser"));
    for(Feature v : l)
    {
      if(i.satisfies(v))
      {
        Class<?> clazz = Class.forName(v.getPath());
        Constructor<?> ctor = clazz.getConstructor();
        return ctor.newInstance();
      }
    }
    throw new Exception("Feature not available for:
  ↪ \textless" + c.getName() + "\textgreater");
  }
}
```

Listing 10: Example of Variant Factory

Chapter 2

Architectural and Design Patterns in PLeTs v4

While Chapter **??** focused on the specific code mechanisms used to implement variability management in Java, this chapter approaches the problem from a higher-level perspective, analysing patterns that may be applied to solve common problems that arise during the course of design and development. While the topics of design patterns [4] and architectural patterns [12] are very deeply studied in several excellent works, this document seeks to apply a specific few to solving the problems that were identified in PLeTs in the past (v1-3) and which are expected to arise over the course of PLeTs v4.

The differences between architectural and design patterns are substantial and very important (see [12]), but for the purposes of this document they will be treated as virtually the same. Many of the problems raised in this chapter in the current version of this document are still open problems in the design and architecture of PLeTs v4, and therefore any input provided is welcome.

How to read this chapter

This chapter is written to serve as both an introduction to the patterns used in PLeTs v4 and as a reference guide for use over the course of the project. Any information regarding the situations in which to apply a pattern or form of its implementation should first be sought in this document.

Section 2.1 briefly presents the Factory design pattern, based on the original description by Gamma et al. [4], as well as the more specific application of it in PLeTs, called the Variant Factory [7].

Section 2.2 *very* briefly presents the notions of software connectors, based on the description by Taylor et al. [12], with an emphasis on the linkage- and arbitrator-type connectors.

Section 2.3 follows up on these concepts with the idea of the architectural pattern behind the dynamic composition and communication of modules within

products that is the cornerstone of PLeTs v4. We explore two architectural styles, Distributed Objects (best known for its flagship framework, CORBA [8]) and C2 (described in detail in [12]), and how ideas from each of them were used to conceive the "Builder" (name pending) Architectural Pattern for PLeTs v4.

2.1 Factory Pattern

2.1.1 Variant Factory

2.2 Software Connectors

2.2.1 Linkage Connectors

2.2.2 Arbitrator Connectors

2.3 PLeTs v4 Architectural Pattern - "Builder"

2.3.1 Distributed Objects Architectural Style

2.3.2 C2 Architectural Style

Chapter 3

PLeTs v4 Modules

Brief introduction of what PLeTs is, presentation of the Feature Model, explanation of the purpose of the project from the point of view of the Testing domain. 2-4 paragraphs should be enough.

3.1 Variability Point 1

Description of the variability point (each abstract feature), what its role is in the product line, what are the typical input and output artifacts. 1-3 paragraphs per variability point should be enough. If the variability point has specific details, enter them in the following format or similar:

Involved constraints:
- Constraint a
- Constraint b
External dependencies:
- Dependency a
- Dependency b

Listing 11: Variant 1.A Interface

Variability point constraints are those constraints between features, such as *requires* and *excludes*. External dependencies are any dependencies that the variability point has to hardware or external software, such as "requires having Tool a installed".

3.1.1 Variant 1.A

Very brief description of the variant (1 paragraph) and definition of the interface in the following format or similar:

Artifacts will generally be files or physical data structures. If they are a feature of the product line, a reference to that feature suffices. If not, a section

In:
- Artifact 1 (described in **??**)
- Artifact 2 (described in **??**)
Out:
- Artifact 3 (described in **??**)
Inout:
- Artifact 4 (described in **??**)
Pre-Condition:
- Inner Constraint a
- Inner Constraint b
Post-Condition:
- Inner Constraint c
Invariants:
- Invariant a
Involved constraints:
- Outer Constraint a
- Outer Constraint b
External dependencies:
- Dependency a
- Dependency b

Listing 12: Variant 1.A Interface

must be created to briefly describe the artifact. Inout artifacts represent artifacts that are both input and output. Inner Constraints will typically be a brief description of the process executed by the variant, and rules as to what may or may not be modified (in the artifacts) over the course of its operation. Not all Variants will have Inner Constraints, and inner constraints are typically applied to inout artifacts. Invariants are any information that must be valid throughout the execution of the variant, and are used primarily to specify restrictions on shared-access data structures and parallel execution (generally none should be present in PLeTs). External dependencies work the same way as in Variability Points.

3.1.2 Variant 1.B

3.2 Variability Point 1.B.1

A variant may have its own variability points. These lower-level variability points and their variants must be documented in the same way as the higher-level ones.

3.2.1 Variant 1.B.1.A

3.3 Variability Point 2

3.3.1 Variant 2.A

3.4 Artifact 1

All artifacts that interact with the tool over the course of its execution must be specified or minimally described. Those artifacts that are not features of the product line, such as files or input parameters, may be described in stand-alone sections such as this one. Short descriptions of 1 paragraph should suffice for this.

Chapter 4

PLeTs v4 Architecture

How to read this chapter

4.1 Feature Model

4.1.1 Feature Traceability

4.2 Component Model

4.2.1 Dependency Control

4.3 Status Report

Appendices

Appendix A

Compilers

Most of our professionals are unaware of the exact way through which compilers work, but a basic understanding of their mechanisms is necessary to fully comprehend the uses of compilation directives. Compilers can generally be divided in three or four smaller tools that, together, execute the entire compilation process. These are the Lexical Analyser, the Syntactic Analyser, the Semantic Analyser, and, optionally, the Code Generator. Other features can be included in compilers, such as code optimizers, but these are not covered here.

Section A.1 covers the basics of regular expressions, which are fundamental tools for the appropriate implementation of almost any software system. Section A.2 briefly explains the purpose of lexical analysers, and how they make use of regular expressions to prepare a stream of text for compilation. Section A.3 attempts to give a sufficient understanding of syntactic analysis and grammars within their application in compiler technology. Finally, Section A.4 presents the basics of semantic analysis, in order to prepare the reader for the more advanced concepts presented in Section **??**.

A.1 Regular Expressions

The simplest unit of analysis that a compiler uses is a stream of text, or a string. These streams of text, while usually structured according to a grammar, may be formed of any number of compositions of characters, and identifying these compositions as the structural units of a grammar require these units to have their own individual composition rules. These rules are called regular expressions. A regular expression is a structured string of characters that define a pattern. This pattern can then be compared to any other string to see whether it conforms to the regular expression or not.

We will use as an example the typical "Hello World" Java program, based on the code presented in Listing 13, taken from [11].

This code is what is called a *valid word* within the Java language, meaning it conforms to the language's grammar. However, in order for a syntactical

```java
public class HelloWorld {
  public static void main(String[] args) {
      // Prints "Hello, World" to the terminal window.
      System.out.println("Hello, World");
     }
}
```

Listing 13: HelloWorld.java from [11]

analyser to be able to tell that this code is a valid word, it needs to know how the code, or stream of text, is *structured*. Regular expressions are the first step towards that goal. Let us take, for example, the regular expressions in Listing 14, which is a (very) simplified example of some of the regular expressions implemented in any Java compiler.

⟨*ACCESS_MODIFIER*⟩ ::= public | private | protected

⟨*CLASS*⟩ ::= class

⟨*IDENTIFIER*⟩ ::= [A-Za-z_$][A-Za-z_$0-9]*

⟨*LEFT_BRACKET*⟩ ::= {

⟨*SPACE*⟩ ::= [\r\n\t]+

Listing 14: Reduced example of possible regular expressions for Java tokens.

If we follow the rules set out in Listing 14 to analyse the code in Listing 13, we will find that the first item in the stream conforms to the rule "ACCESS_MODIFIER" (public), the second conforms to "SPACE" (the empty space between "public" and "class"), the third to "CLASS" (class) and so on. Regular expressions like "ACCESS_MODIFIER" and "CLASS", where the expected string is an entire word rather than a pattern of characters, are typically used for identifying *reserved words*, words that have special meaning in a language. The more interesting type of regular expressions however are the ones like "IDENTIFIER" and "SPACE", where a pattern is expected rather than a word.

Regular expression patterns vary from language to language, but they generally look alike: characters inside brackets ([ ]) form a set, where any of the characters inside the set is acceptable in that position; escape characters like "\r" and "\n" are counted as a single character, since they are represented by a single ASCII byte; the * symbol means that anything that comes before it, usually a set, may be matched zero or more times, and; the + character means

that anything that comes before it, usually a set, may be matched one or more times.

From these definitions, we may read that an "IDENTIFIER" is any string formed by a letter, underscore or $, followed by any number of letters, underscores, $ or digits. Likewise, a "SPACE" is any string formed by one or more space, \r, \n or \t. By using regular expressions like these it is possible to create patterns to represent any sequence of characters (the website RegexOne [9] is an excellent source of exercises for learning the basics of regular expressions).

It is important to note that when a single string matches more than one regular expression, the decision of which to use is based on the priority of the rules within the ruleset and on which rule is "best matched", that is, which rule matches the string with the most units used. For example, the string "public" could very well match the "IDENTIFIER" rule, but the "ACCESS_MODIFIER" rule has both priority over "IDENTIFIER" (reserved word rules almost always take priority) and provides a better match, since "public" matches six units of the "ACCESS_MODIFIER" but only two units of "IDENTIFIER" (each set counts as one unit).

Understanding and being able to write regular expressions is a basic skill that is expected of any programmer; it is extremely recommended that all members of the project understand this section and do the exercises on RegexOne. More than being used in compiler technology, regular expressions are a basic building block in any system that performs analysis of strings, validation of fields and a large number of other procedures.

A.2 Lexical Analyser

The lexical analyser is the part of a compiler that is responsible for *tokenizing* the input text. What this means is that the stream that is sent as input to the compiler is fully read and divided into its basic units, such as "identifiers" and "reserved words", by matching them to a list of regular expressions that define each type of token. Essentially, a stream of text is transformed into a list of values that the compiler understands. This is important for the compilation process as the steps that follow (syntactic and semantic analysis) *are not entirely concerned with the exact text included in the input, but rather with its structure.* This tokenization process is generally executed in a single pass (meaning the stream is read only once), and outputs an ordered list of tokens. To better understand the process of lexical analysis, let us take another look at Listing 13.

In analyzing the "HelloWorld" code, which is read as a single stream of text (one large string), a lexical analyser may identify tokens like "ACCESS_MODIFIER" (public), "CLASS" (class), "IDENTIFIER" (HelloWorld), "LEFT_BRACKET" ({), "ACCESS_MODIFIER" (public), and so on based on the regular expression that define its tokens. Notice that the exact contents of a token are irrelevant at this point in the analysis; rather, the lexical analyser identifies the *type* of unit that a particular piece of code represents. This is because the

exact text is irrelevant to the correct syntax of the code; as we know, we can name a Java class pretty much anything, so long as it follows a simple regular expression like the "IDENTIFIER" rule in Listing 14. Whether the name of the class is "HelloWorld" or "ClassNameHere", the form of the code will remain the same.

Also notice that blank spaces, such as the ones defined by the rule "SPACE" in Listing 14 are generally not tokenized. This is because the blank space is usually used as a wildcard for the delimitation between two tokens, which means the number of blank spaces is not important to the tokenization, and also means that the simple fact of the tokenizer separating, say, "ACCESS_-MODIFIER" and "CLASS" already acknowledges that there was indeed a space between them (the space is implicit in the tokenization). This is not always the case, however, as languages like Python do use indentation in their syntax.

The importance of understanding lexical analysis for the understanding of conditional compilation is that the conditional statements, such as #ifdef or #if, or the constant declaration #define, are in fact tokens to the compiler. Whether the token value expected by the compiler is #ifdef, #if, or something else entirely (say, #condition, for example) is not important: it is the meaning of these tokens that matters, and the lexical analyser may very well group them all as a "CONDITIONAL_CLAUSE" token class. For this reason, despite there being no specific preprocessor in Java, it is plausible to imagine a system in Java where a conditional compilation clause is created in the form of a token class "CONDITIONAL_CLAUSE" or anything of the sort, where the value is whatever we choose for it. Listing 15 shows a possible rule to represent this concept.

$\langle CONDITIONAL\_CLAUSE \rangle$ ::= #ifdef | #if | #condition

Listing 15: Example of conditional token class.

Note that the use of the escape character # before each of the possible values of "CONDITIONAL_CLAUSE" means the lexical analyser will be able to skip almost all of the text stream, whereas the uses of the same strings without an escape character would mean the lexical analyser would need to validate every string that partially matched the rules, such as "conformity" (matches "con"). This is done both in order to speed up the process and to limit the reserved words in a language. The fewer reserved words, the more freedom a programmer has to name variables and functions, and therefore make code more readable.

Understanding lexical analysis on its own may appear confusing, as it is largely just a tool to be used by the syntactic analyser. For this reason, feel free to return to this section after you understand syntactic analysis, as it may make more sense to you then.

A.3 Syntactic Analyser

The syntactic analyser is charged with making sense of the tokens provided by the lexical analyser. What it does is check the order of these tokens against a set of rules, called a grammar, to see whether the input text makes sense or not (whether it is a *valid word* of the language). The importance of this is to guarantee that the input text is in the right format for the compiler, so that it may be able to interpret the file properly. Think of it as checking the structure of a sentence in natural language, and making sure that pronouns, nouns, verbs, adjectives, etc. are used correctly.

Listing 16 presents a simplified grammar based on the code from Listing 13 and the regular expressions from Listing 14. Note that the grammar is presented in what is known as the Backus-Naur Form (BNF), the format usually used to represent a *context-free grammar*. Also note that typically, "terminal rules" (rules that translate to a lexical token) are typically written in capitals, where "non-terminal rules" (rules that translate into further rules) are typically written in non-capitals.

$\langle class \rangle ::= \langle ACCESS\_MODIFIER \rangle \quad \langle CLASS \rangle \quad \langle IDENTIFIER \rangle$
$\langle LEFT\_BRACKET \rangle \; \langle class\_inside \rangle \; \langle RIGHT\_BRACKET \rangle$

$\langle class\_inside \rangle ::= \langle comments \rangle \; \langle declaration \rangle$

$\langle comments \rangle ::= \epsilon \mid ...$

$\langle declaration \rangle ::= \langle method \rangle \mid \langle attribute \rangle \mid \langle declaration \rangle \; \langle declaration \rangle \mid \epsilon$

Listing 16: Reduced example of a possible grammar to identify a Java class code

This grammar specifies that a class is comprised of an access modifier (public, private, protected, etc.), the reserved word "class", an identifier (like "HelloWorld" or "ClassNameHere"), a left bracket ({), the contents of the class (methods, attributes, etc.), and a right bracket (}). The contents of the class ("class_inside"), in turn, are formed of comments and declarations. Comments can either be empty (represented by ϵ) or made up of a group of rules specific for formatting comments (omitted here). Declarations can either be empty, a method or an attribute; in this special case, the sequence <declaration> <declaration> is used to demonstrate that a declaration may be formed of two declarations. Essentially, this means that there can be any number of declarations in a class; it is akin to the + symbol in regular expressions.

Effectively, what the syntactic analyser will do is take the tokens given it by the tokenizer and check them against the grammar. It knows that the base rule is <class> (the first rule in the grammar), so the first thing it looks for is a token "ACCESS_MODIFIER". It goes from there until it reaches the rule <class_inside>, which causes it to shift what it is looking for from the <class> rule to the <class_inside> rule.

Let us consider a more practical example towards the problem our project is faced with: conditional compilation.

$\langle conditional \rangle ::= \langle CONDITIONAL\_CLAUSE \rangle \quad \langle IDENTIFIER \rangle \quad \langle code \rangle \quad \langle END\_CONDITIONAL\_CLAUSE \rangle$

$\langle code \rangle ::= ...$

Listing 17: Example of non-terminal conditional rule.

The rules in Listing 17 define that if the syntactic analyser finds a "CONDITIONAL_CLAUSE" token, it must verify whether the next token is an "IDENTIFIER", and so on. Adapting the HelloWorld example based on Listings 15 and 17, we could create a code that may be evaluated by the "conditional" rule like the one in Listing 18.

```
1  #define boolean PRINT = true;
2
3  public class HelloWorld {
4    public static void main(String[] args) {
5        #if PRINT
6        // Prints "Hello, World" to the terminal window.
7        System.out.println("Hello, World");
8        #endif
9      }
10 }
```

Listing 18: Adapted from HelloWorld.java from [11]

The lexical analyser will have read "#if" as CONDITIONAL_SYMBOL, "PRINT" as IDENTIFIER and "#endif" as END_CONDITIONAL_SYMBOL, with everything in-between being analysed as normal code (any number of other tokens). The syntactic analyser will then be able to realize that the code in that section falls in a conditional compilation rule, and will set it aside for special treatment by the semantic analyser or code generator. In practice, we have established that our HelloWorld class will only print "Hello, World" on the screen if we define the constant PRINT when compiling.

At this point, you should understand the basics of why we define tokens, and how they are interpreted by the syntactic analyser. However, this alone is not enough to fully give meaning to an input stream; we have decided that our code is a valid word in our language, but we do not yet know what that word *means*. That is the job of the semantic analyser.

A.4 Semantic Analyser

The semantic analyser is in charge of deciding what the input text means after the syntactic analyser has assured it that the text is well-formed. The semantic analyser will essentially be a software application itself, treating input text according to what it receives. For example, a semantic analyser for the previous example in Listing 17 would include a program, or a piece of code, stating that when the syntactic rule "conditional" is found, the string identified by the rule "code" will only be compiled if the constant named by the "IDENTIFIER" token was defined before compilation. In the case of the code in Listing 18, the semantic analyser would state that if PRINT is not defined, then the statement inside the conditional should not be compiled. Essentially, it says "if PRINT is false, ignore everything in this block".

This is where we meet our problem with the Java compiler, and where our Java Preprocessor study comes in. To be able to identify these kinds of semantic rules, C++ and C# (for example) possess what is called a preprocessor. What this preprocessor does is take in a number of constant definitions, like "int TWO = 2" or "boolean PRINT = true", and replace every instance of that constant in the code with the value of the constant. Essentially, after the preprocessor has done its work, Listing 18 would look like this:

```
1  public class HelloWorld {
2    public static void main(String[] args) {
3        #if true
4        // Prints "Hello, World" to the terminal window.
5        System.out.println("Hello, World");
6        #endif
7      }
8  }
```

Listing 19: Adapted from HelloWorld.java from [11]

As you can see, the constant definition is removed entirely from the program (since it has already served its purpose) and all of its appearances were replaced with its value. By doing this, the preprocessor allows the compiler to know that section of code should be compiled; if PRINT had been equal to false (that is, the compilation constant PRINT was not defined), then the compiler would know to ignore that section of code.

What happens in Java is that without a preprocessor we have no means of telling the compiler which parts of the code should or should not be compiled; we cannot define constants to be read in that way. In fact, the preprocessor for C# already knows what #if and #endif mean, and if PRINT had been false, the preprocessor itself would have removed that code. The compiler would never even know it existed. The reason why this is done by a preprocessor, rather

than the normal compiler, is two-fold. First, it allows the designers of the language to create two separate grammars, one for the preprocessor and one for the compiler, where both are much simpler and easier to understand than they would be if they were a single thing. Second, it reduces the number of times that the compiler must read the code before creating the executable (generating assembly code), making it much faster.

The Java Preprocessor study is therefore dedicated to finding a way to define conditional compilation in Java, so that portions of code may be compiled or not based on the configuration of a product within an SPL. Ultimately, by creating a program that takes in the Java source code and transforms it, we are able to tell the compiler exactly what we want, and what we don't want in our executables (.class and .jar files).

Bibliography

[1] S. Apel, D. Batory, C. Kstner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[2] F. Bachmann and P. C. Clements. Variability in software product lines. Technical report, Carnegie Mellon University, 09 2005.

[3] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[5] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[6] J. Kropf. Java preprocessor. slashdev.ca/javapp, 2015. Date accessed: 23/04/17.

[7] M. S. Laser, E. M. Rodrigues, A. R. P. Domingues, F. M. Oliveira, and A. F. Zorzo. Architectural Evolution of a Software Product Line: an experience report. In *27th SEKE*, pages 217–222, Pittsburgh, USA, 2015.

[8] OMG. Common object request broker architecture. http://www.omg.org/spec/CORBA/, 2012. Accessed: 01/04/2017.

[9] RegexOne. Regexone - learn regular expressions with simple, interactive exercises. https://regexone.com, 2016. Date accessed: 08/04/17.

[10] M. Schmitt Laser and L. Dumer. Autorest. https://github.com/autorest, 2017.

[11] R. Sedgewick and K. Wayne. Helloworld.java. http://introcs.cs.princeton.edu/java/11hello/HelloWorld.java.html, 2011.

[12] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.