Universidade Federal do Pampa

Introduction to Variability Mechanisms in Java

Marcelo Schmitt Laser

April 9, 2017

Introduction

In order to properly implement the design decisions made for the PLeTs v4 project, it is necessary that all professional 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 the notion of using compilation directives for conditional compilation. First a basic theory of the concept is presented, based largely on the author's experience. The mechanism of using Static Final variables in Java is presented, along with its advantages and limitations. Then, the mechanisms of using Java annotations are evaluated. Finally, we analyse the need and properties of using an external preprocessor software, along with its perceived advantages.

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 Variant Factory design pattern will be presented [5] alongside the data structure catalogue system implemented in PLeTs v3. We then briefly expose the basic concepts of Software Connectors [10], 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 (which, as far as we could find, was first used in CORBA [6]).

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.

Acronyms

BNF	Backus-Naur Form. 9
CBSE	Component-Based Software Engineering. 5
CORBA	Common Object Request Broker Architecture. 1
SPL	Software Product Line. 5, 12
SPLE	Software Product Line Engineering. 5

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. 5

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 [4]. 6, 9, 10, 12

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 [4]. 6–10

Contents

Chapter 1

Compilation Directives

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 quality 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).

Though variability mechanisms can be considered to have a "binding time" (such as compile, link, run, load, etc.), this chapter considers only the variability mechanisms for Compile-Time variation. This focus is due to our current focus being on *static* SPL, which are SPL where variation is handled during the configuration of a product, rather than while the product is running.

Compilation directives are well-known in C++, often identified as "#ifdef". While C# made a similar mechanism available to us (#if), Java is lacking in an in-built preprocessor, and therefore has no mechanism with the exact same purpose as the C++ compilation directives. This chapter presents a short introduction to compilers and the concept of compilation directives and preprocessors, as well as the reasons why these are used in PLeTs and how to use Static Final variables in Java to reach a similar result.

How to read this chapter

This chapter is very theory-heavy, and therefore may be a "heavier" read for people less acquainted with the concepts presented. Furthermore, the depth at which they are presented is not meant to fully clarify their inner workings, as there are entire books dedicated to that [1] [2].

Section 1.1 covers the basic concepts of compiler technology, relating the notions explained to their use in variability management and implementing compiler directives. It is recommended that this section be read more than once, as understanding the concepts explained at the end of the section may clarify some of the concepts presented at the beginning, and vice-versa.

Section 1.2 more specifically analyses the means of using compilation directives in the Java language. This section covers the in-built mechanisms that Java has available and why they are insufficient to our purposes. Understanding this section is important for the appropriate use of directives, and crucial to the developers in charge of the Java Preprocessor project.

The Java grammars and regular expressions are purely pedagogical, and in no way represent the real syntax of a Java compiler. For the official specification of the Java language, see [3]. Knowledge of the Java specification is *not* required for this project, and is here for referencing purposes only.

1.1 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.

1.1.1 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 1, taken from [9].

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 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 2, which is a (very) simplified example of some of the regular expressions implemented in any Java compiler.

If we follow the rules set out in Listing 2 to analyse the code in Listing 1, we will find that the first item in the stream conforms to the rule "ACCESS_MODIFIER" (public), the second conforms to "SPACE" (the empty

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

Listing 1: HelloWorld.java from [9]

$\langle ACCESS\_MODIFIER \rangle$ ::= public | private | protected

$\langle CLASS \rangle$ ::= class

$\langle IDENTIFIER \rangle$ ::= [A-Za-z_$][A-Za-z_$0-9]*

$\langle LEFT\_BRACKET \rangle$ ::= {

$\langle SPACE \rangle$ ::= [\r\n\t]+

Listing 2: Reduced example of possible regular expressions for Java tokens.

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 [7] 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.

1.1.2 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 1.

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 2. 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 2 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 3 shows a possible rule to represent this concept.

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

Listing 3: 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.

1.1.3 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 4 presents a simplified grammar based on the code from Listing 1 and the regular expressions from Listing 2. 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.

$\langle \mathit{class} \rangle$ ::= $\langle \mathit{ACCESS\_MODIFIER} \rangle$ $\langle \mathit{CLASS} \rangle$ $\langle \mathit{IDENTIFIER} \rangle$ $\langle \mathit{LEFT\_BRACKET} \rangle$ $\langle \mathit{class\_inside} \rangle$ $\langle \mathit{RIGHT\_BRACKET} \rangle$

$\langle \mathit{class\_inside} \rangle$ ::= $\langle \mathit{comments} \rangle$ $\langle \mathit{declaration} \rangle$

$\langle \mathit{comments} \rangle$ ::= ϵ | ...

$\langle \mathit{declaration} \rangle$ ::= $\langle \mathit{method} \rangle$ | $\langle \mathit{attribute} \rangle$ | $\langle \mathit{declaration} \rangle$ $\langle \mathit{declaration} \rangle$ | ϵ

Listing 4: 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.

$\langle \mathit{conditional} \rangle$ ::= $\langle \mathit{CONDITIONAL\_CLAUSE} \rangle$ $\langle \mathit{IDENTIFIER} \rangle$ $\langle \mathit{code} \rangle$ $\langle \mathit{END\_CONDITIONAL\_CLAUSE} \rangle$

$\langle \mathit{code} \rangle$ ::= ...

Listing 5: Example of non-terminal conditional rule.

The rules in Listing 5 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 3 and

5, we could create a code that may be evaluated by the "conditional" rule like the one in Listing 6.

```
#define boolean PRINT = true;

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

Listing 6: Adapted from HelloWorld.java from [9]

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.

1.1.4 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 5 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 6, 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 6 would look like this:

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

Listing 7: Adapted from HelloWorld.java from [9]

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).

1.2 Compilation Directives in Java

1.2.1 Static Final Attributes

One way to get around the problem of Java not having a preprocessor is by abusing the optimization features that Java does 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.

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 be false or not, but since a Static Final attribute can never change its value, the optimizer will be able to tell. Listings 8 and 9 show an example of how this can be used.

```
public class Preprocessor
{
 public static final boolean FULLACCESS = false;
  ...
}
```

Listing 8: Preprocessor.java, encapsulating variability points

A Preprocessor class is declared in Listing 8 which contains all of the variability point constants for free access from the rest of the project's packages. The packages may then use these attributes in normal conditional loops to transform them into conditional compilation loops. In Listing 9, the method getEnd1 is conditionally compiled so that if the selected product has full access, a reference to end1Element is returned, whereas if the selected product does not have full access, only a copy is returned. 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. While it works in the example above, it does work in the example presented in Listing 10.

In this example, an entire constructor should be conditionally compiled so that this particular overload is only available if the configured product has FULLACCESS. 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 an IDE like NetBeans or Eclipse would be fooled into thinking this overload is available, when in fact it is not. The ideal way of solving this

```java
import Preprocessor;

public class UmlAssociation extends UmlElement implements
 ↪  Serializable
{
...
  private UmlElement end1Element;
...
  public UmlElement getEnd1()
  {
    if(Preprocessor.FULLACCESS)
    {
      return this.end1Element;
    }
    else
    {
      return DeepCopy.copy(this.end1Element);
    }
  }
...
}
```

Listing 9: Variability point being resolved by the Preprocessor.java attributes, taken from [8]

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. To be able to do this, we fall back into the idea of making our own Java preprocessor, with capabilities similar to the C++ and C# preprocessors.

1.2.2 Preprocessing with Annotations

1.2.3 Java Preprocessor Project

```java
import Preprocessor;

public class UmlAssociation extends UmlElement implements
 ↪  Serializable
{
...
  //INCORRECT JAVA SYNTAX, DOES NOT COMPILE!!!!!
  if(FULLACCESS)
  {
    public UmlAssociation()
    {
      BuildIt(null, null, null, null, null, null, null,
 ↪  null, null);
    }
  }
...
}
```

Listing 10: Limitations of Static Final attributes, taken from [8]

Chapter 2

Architectural and Design Patterns in PLeTs v4

2.1 Variant Factory

2.2 Software Connectors

2.2.1 Linkage Connectors

2.2.2 Arbitrator Connectors

2.3 PLeTs v4 ArchPatt - “Builder”

2.3.1 Distributed Objects Architectural Style

Chapter 3

PLeTs v4 Modules

3.1 Input Processing

3.1.1 XML Reader

3.2 Test Sequence Generation

3.2.1 Depth-First Search

3.2.2 Breadth-First Search

3.3 Test Case Generator

3.3.1 Performance Test Case Generator

3.4 Test Script Generation

3.4.1 JMeter Script Generator

3.4.2 Oracle Application Testing Suite (OATS) Script Generator

3.5 Test Executor

3.5.1 JMeter Test Executor

3.5.2 Oracle Application Testing Suite (OATS) Test Executor

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[3] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[4] 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.

[5] 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.

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

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

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

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

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