UNIVERSITÄT
BERN

Adding Class Support and Global Methods to Polite Smalltalk

Bachelor Thesis

Thomas Steinmann
from
Bern, Switzerland

Faculty of Science
University of Bern

4. April 2016

Prof. Dr. Oscar Nierstrasz
Jan Kurš
Dr. Mircea Lungu

Software Composition Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Polite Smalltalk was created as part of a proposed study about the impact of Sentence Identifier Names on code readability and maintainability. It was based on a Smalltalk environment where it was parsed and compiled to Smalltalk code. Polite syntax was kept very minimal and did not support class or method definitions, even though Smalltalk classes and their methods could be used.

In order to create code that is both more readable and very similar to natural language, the goal of this thesis is to extend the grammar and implementation of Polite Smalltalk with class and method definitions, as well as the necessary tools to provide a workspace for writing and executing a complete Polite program. In order to handle these new requirements, the original Polite's compiler architecture must be improved.

The validation of our work shows that several non-trivial programs, each highlighting one aspect of the new functionality, can be written and executed yielding the expected results. Furthermore, arbitrary code can be parsed and evaluated in the Polite Playground with expected features and results.

Acknowledgements

I would like to thank *Professor Oscar Nierstrasz* and the *Software Composition Group* for making this thesis possible. Especially my tutor, *Jan Kurš* who trusted me with his project and taught me a lot during this thesis.

And also the people who discussed with me, inspired me, read my script and supported my ideas. Thank all of you for your effort.

Contents

1 Introduction **5**

2 Related Work **7**
2.1 Pharo Smalltalk 7
2.2 Parsing Expression Grammars 7
2.3 PetitParser 8
2.4 Abstract Syntax Trees and Visitors 8
2.5 Sentence Identifiers 8

3 Initial State **9**

4 Thesis Objectives **11**
4.1 Class and Method Definition 11
4.2 Global Methods and Messages 12
4.3 Compiler Infrastructure 12
4.4 Polite Vision 13

5 General Approach **14**
5.1 Challenges 14
5.1.1 Syntax Extension 14
5.1.2 Global Methods and Messages 14
5.1.3 Compiler Architecture 15
5.2 Solutions 15
5.2.1 Syntax Extension 15
5.2.2 Global Methods and Messages 16
5.2.3 Compiler Architecture 16

6 Implementation **18**
6.1 Syntax Analysis Step 18
6.2 Compilation Step 20
6.2.1 The Depolitor 21
6.2.2 The PSGlobalMessageSearchVisitor 21

6.2.3 The PSCompiler 22
6.3 Execution Step 23
6.4 Approaching Global Methods 23

7 Validation **25**
7.1 Polite Vision 25
7.2 Recursive Factorial 26
7.3 LO Game 26
7.4 The Polite Playground 26

8 Conclusion and Future Work **27**
8.1 Conclusion 27
8.2 Future Work 28

9 Bibliography **28**

A Anleitung zu wissenschaftlichen Arbeiten **31**
A.1 Installation 31
A.2 Using Polite inside Pharo 32
A.3 Syntax 32
A.4 Conventions 33

B Validation Code **35**
B.1 Polite Vision 35
B.2 Recursive Global Methods 37
B.3 LO Game 38

1 Introduction

It is generally agreed that method and class names play a considerable role in the readability of programs. Some programmers even argue that comments can be omitted entirely if every method or class name accurately describes its functionality, which is considered good practice in Smalltalk.

Robert Martin describes the importance of code readability for productivity as follows: "The ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code." [10]. If identifier names are that important to better understand code and understanding code is as important to writing it as Martin proposes, it follows logically that by improving the quality of identifier names the overall readability of code can be improved significantly.

Polite Smalltalk was introduced as a scripting language in 2013 by Jan Kurš and Mircea Lungu in order to study the impact of Sentence Identifier Names[1] on readability and maintainability.[9] It was designed with a syntax very similar to regular Smalltalk but with a few alterations in order to create a language that could be read just like an English sentence, using sentence identifiers and relying on very few control symbols such as brackets, commas *etc.* Let us call this property "politeness". It is expected that higher politeness leads to higher readability and therefore to faster writing and higher quality of code but this still has to be proven experimentally and is not the object of this thesis.

This thesis aims to extend Polite Smalltalk with class and method definitions, as

[1] Sentence Identifiers are allowing identifier names to consist of several words separated by character spaces rather than limiting them to one word using camelCasing or under_scoring.

Figure 1.1: The vision of a language as readable as everyday English

well as global methods and variables for greater politeness. In order to achieve such a behavior, a dedicated compiler is required. With these changes, Polite becomes a fully object oriented programming language, parsed and compiled into viable Smalltalk classes that can be executed independently from the original Polite code.

These improvements allow for code as polite as shown in Figure 1.1 and after this iteration Polite is ready for use in studies such as the one it was created for [9].

2
Related Work

This chapter introduces studies and technologies that were used in or had an impact on this thesis.

2.1 Pharo Smalltalk

Pharo Smalltalk is one of the most popular implementations of Smalltalk [5]. It provides an IDE as well as being its own programming language. Smalltalk code is represented in the system browser which works similar to a folder based file organization system.

2.2 Parsing Expression Grammars

Parsing expression grammars (PEG) [2] offer a formalism for describing grammars. A main difference between PEG and BNF, the predominant formalism for discussing grammars, is that PEG is not ambiguous when it comes to the choice operator. The second option in a PEG choice statement is only considered if the first one fails. This means that, contrary to the situation in BNF, a rule `rule ← a / a b` would never evaluate to `a b` because the first choice `a` would be accepted before ever checking if the second was viable as well.

The ← symbol describes an assignment to a rule and Table 2.1 shows the relevant PEG operators for the grammar rules used in this thesis.

Operator	Description
$'abc'$	Literal string
$e?$	Optional expression
$e*$	Zero-or-more repetitions of e
$e_1\ e_2$	Sequence of expressions
e_1 / e_2	Prioritized choice

Table 2.1: Operators for constructing parsing expressions

2.3 PetitParser

PetitParser [7, 11] is a parser framework for Pharo Smalltalk and combines several existing parsing approaches in a single framework. PetitParser provides parsers for widely used code features such as words, letters, numbers etc. but also the necessary combinators in order to form more complicated expressions. Furthermore PetitParser supports layout sensitive parsing which can be used to create indentation sensitive grammars using the INDENT and DEDENT parsers [4, 8].

2.4 Abstract Syntax Trees and Visitors

An Abstract Syntax tree (AST) [6] represents the structure of source code in a tree structure. Semantically meaningful constructs are formed and stored as nodes in the tree. Tree structures can then be altered by using visitors [3] that add functionality without changing the implementation of the tree structure.

2.5 Sentence Identifiers

In 2013 Jan Kurš and Mircea Lungu proposed a study to examine the correlation between identifier names and readability [9]. It focuses on the impact of so called *sentence identifiers*, a syntactic construct that allows whitespace in identifiers and therefore more natural class, method and variable names in programs. In order to conduct such a study the scripting language Polite Smalltalk was created. Polite Smalltalk should be as close to natural language as possible, allowing sentence identifiers as well as using as few control symbols as possible, making Polite code almost as readable as an everyday English sentence. This proposed study built the basis for Polite and is the motivation for its implementation.

3
Initial State

Before the start of this thesis the prior iteration of Polite already allowed statements in Polite syntax. It differed from regular Smalltalk syntax only by accepting sentence identifiers and therefore requiring messages to be separated from their receivers by a comma. Listing 3.1 shows an example of what was possible with Polite.

```
|array|
array := Array, with: 3 with: 30 + 2.
^ array, as comma string.
```

Listing 3.1: An example of what Polite was capable of

In addition, Polite syntax allowed every line of code that could be written inside a regular Smalltalk method. A string of Polite code would be parsed, resulting in an abstract syntax tree which could then be depolitized[1] and compiled into a single Smalltalk method. This method could then be evaluated in order to obtain the result of the computation. Listing 3.2 shows the Smalltalk method generated from the Polite code in Listing 3.1. It is represented as a String and can be used as an input for a Smalltalk compiler which allows it to be executed like ordinary Smalltalk code.

[1]Converting sentence identifiers into camel case

```
foo
    |array|
    array := Array with: 3 with: 30 + 2.
    ^ array asCommaString.
```

Listing 3.2: Generated method

What Polite did not yet allow however, was defining user created classes and methods. These definitions are however an essential feature for writing polite code.

4
Thesis Objectives

The objectives of this thesis were set in terms of use cases. Namely: *Define class and class methods*, *instantiate class* and *call class method*. The most challenging of these was allowing user defined classes and class methods since the other use cases directly depended on this one and were mostly solved already while handling class definition. Global methods and messages were added to the thesis objectives later on because they proved useful and were an interesting feature to implement. During the thesis we also recognized that the structure of the Compiler was not designed to handle our requirements and therefore needed some refactoring and a couple of expansions. Since the second and third use case were closely related to the first, they will be covered together from here on, and three main aspects: *class and method definition, global methods and messages* and *Compiler architecture* will remain the focus of the thesis.

4.1 Class and Method Definition

The main objective of this thesis is to identify a viable syntax for class and method definitions and expand the previous iteration of Polite with this new syntax in order to allow class and method definitions as well as provide a way of representing them during translation and in the generated source code.

4.2 Global Methods and Messages

A method defined outside of a class definition is called a global method. When a global method is called it is referred to as a global message. In prior iterations these methods worked coincidentally rather than on purpose. Nonetheless they helped achieve more polite code and therefore inspired the design of global methods as a permanent feature.

With global methods, helper methods can be created in order to write more polite code. Listing 4.1 shows the definition of a global helper method. Instead of writing a standard if-else statement (Listing 4.2) a much cleaner method can be written with the help of the new helper method (Listing 4.3). This is especially noticeable when reading the code with natural language in mind.

```
if: condition then: (true block) else: (false block)
    ^ condition,
    if true: [true block, value]
    if false: [false block, value].
```

Listing 4.1: Definition of the global `if: then: else:` helper method

```
drive off: (an enemy) and save: (the lady)
    (my hero, wins against: an enemy)
    if true: [my hero, saves: the lady]
    if false: [my hero, faints]
```

Listing 4.2: Regular if statement

```
drive off: (an enemy) and save: (the lady)
    if: (my hero, wins against: an enemy)
    then: [my hero, saves: the lady]
    else: [my hero, faints]
```

Listing 4.3: Polite if statement

4.3 Compiler Infrastructure

It became obvious that Polite required a dedicated compiler infrastructure. Its responsibility is handling the translation from Polite source code to viable Smalltalk classes and all the necessary steps during this process. The interaction between Polite and Smalltalk classes and methods is also an important requirement. Smalltalk classes and their methods must be fully usable in Polite syntax.

4.4 Polite Vision

Our vision of how Polite syntax should work and look can be compressed in a short code example shown in Figure 1.1 in the Introduction. When control symbols such as braces and commas are ignored, each part of the code can be read as a meaningful English sentence. While not exactly syntactically correct and certainly not of highly rhetorical quality, it is still easily understandable for the intended purpose. This vision represents the objectives of this thesis perfectly, since this is the sort of code that Polite should accept. It should be noted that the means to increase politeness were discussed and decided on with the original creators of Polite, but this thesis focuses only on the implementation of these means and not their effectiveness.

5
General Approach

This chapter will provide a high level overview of the challenges posed by the implementation of full class support as well as global methods and revising the compiler architecture in Polite. It will also explain how these problems were addressed with the new Polite Compiler architecture.

5.1 Challenges

In order to achieve the objectives specified in Section 4 there are several obstacles to overcome which will be discussed here.

5.1.1 Syntax Extension

The first challenge is the inclusion of class and method definitions in the Polite syntax. A grammar for the other features of the language already exists from prior iterations but these definitions are not part of it. Before implementing these definitions, a syntax must be chosen.

5.1.2 Global Methods and Messages

The major problem with the concept of global methods is that they are not part of Smalltalk, the target language of the translation. In order to translate global methods to Smalltalk, global methods must be transformed to some Smalltalk compatible structure.

Like Smalltalk, Polite knows three kinds of messages. Binary messages which cannot be global, unary messages which consist of only the message itself, and keyword messages which contain the message and one or several parameters.

Global unary messages provide a unique challenge since they cannot be distinguished from variables or keywords grammatically because all of those consist of only an identifier. This means that in the code snippet `my hero, level up` the identifier `my hero` could either be a variable or a message sent to no specified receiver, which would make it a global message. By default they are all considered variables or keywords. The decision of whether an identifier refers to a keyword, a variable or a global unary method must therefore be made during the translation and the representation in the generated code has to be changed accordingly.

5.1.3 Compiler Architecture

For these all tasks mentioned above, a compiler architecture is required which is capable of generating Smalltalk code from Polite input and handling every necessary step during the translation as well as guaranteeing that Smalltalk classes and methods can be used in Polite. The architecture from previous iterations does not suffice and must be adapted appropriately.

5.2 Solutions

This section will present the chosen approaches to all challenges mentioned in Section 5.1.

5.2.1 Syntax Extension

Before class definitions can be implemented in Polite, an elegant class definition syntax must be determined. In contrast to Pharo, Polite does not use a class browser to navigate code. Instead it relies on a more traditional approach with the Polite Playground, a single window where the entire code is displayed in one place.

Because of these different approaches, the Polite class definition must also be specified differently in order to better fit into this one-window approach. The chosen layout is also similar to the languages mentioned above. One distinction is that Polite is indentation sensitive. This means that rather than using brackets to form blocks, every statement in a block is indented one more column than the element it belongs to.

The method syntax can be kept from Smalltalk with the additional general Polite rules which state that in Polite, sentence identifiers are permitted, the entire code is indentation sensitive and as mentioned before, commas are used to separate messages from their receivers.

After specifying the class syntax theoretically, implementing class and method definitions in Polite syntax requires new rules in both the grammar and parser which handle the transformation of source code to array form and from array form to ASTs respectively. The environment for these rules is given by the PSGrammar and PSParser classes, which can be expanded for this purpose.

5.2.2 Global Methods and Messages

There is no predefined way for implementing global methods. Since Smalltalk does not know the concept of global methods and messages, these elements must be transformed to a structure that can be understood by Smalltalk. The chosen approach is to create a Smalltalk class that handles all global requests. Global methods can be stored as class methods of this global class, and global messages, lacking a dedicated receiver, can implicitly be sent to it in order to call them.

Because global unary messages cannot be recognized syntactically, they are initially handled the same way variables are. They are stored in variable nodes in the AST by the grammar and parser. In order to distinguish global unary messages in the generated Smalltalk code, these variable nodes must be reviewed and represented accordingly in a later stage of the translation process. Conceptually, each variable node is thereby checked against the variable scope. In case a variable with the same name as the one of the variable node is in scope, the variable name is considered to represent that variable. The variable node is only interpreted as a global method if no variable with the given name is in scope. This naturally follows the intuition that the local scope is always prioritized before a wider one since global methods are always defined globally and therefore have the lowest priority.

Global keyword messages on the other hand can be recognized syntactically and are implemented as a grammar extension. They are still represented as variable nodes in the AST however, in order to unify the handling of all global messages. This way both global unary and keyword messages are represented as variable nodes and have to be detected at a later point.

5.2.3 Compiler Architecture

The compiler architecture from prior iterations is being reworked and can now be divided into three groups, each containing classes used in one step of the translation. The grammar and parser classes are used for parsing the Polite source code and creating an AST. The depolitor, global message search visitor, and compiler are used to gradually alter the AST until it can be compiled to Smalltalk classes. Meanwhile the Polite Smalltalk class acts as the interface of the implementation and is responsible for evaluating the generated Smalltalk code. Figure 5.1 represents the translation process in three steps.

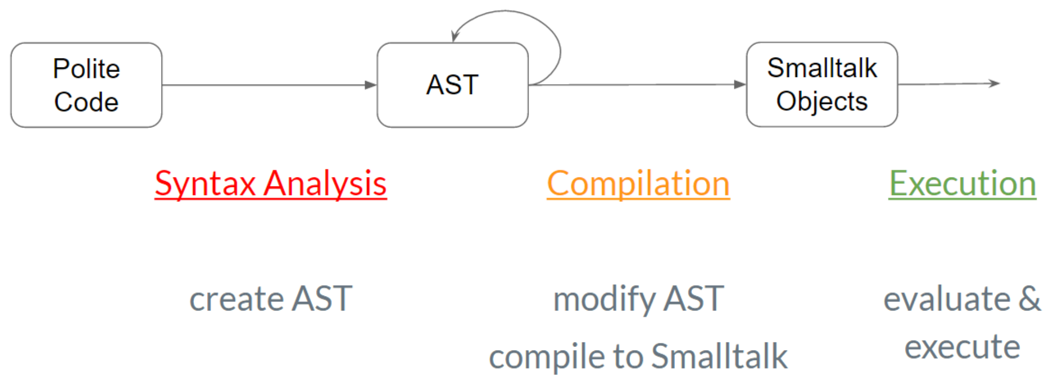

Figure 5.1: Polite Translation Process

The interaction between Polite and Smalltalk classes and methods is addressed by depolitization which means transforming sentence identifiers to Smalltalk identifiers. This way a Smalltalk identifier can be referenced via a sentence identifier in the Polite code.

6
Implementation

This chapter describes the current state of the Polite implementation, looking at every step of the translation process chronologically. This translation process covers the entire process of translating a program in Polite syntax to viable compiled Smalltalk classes and executing them. It consists of the following three steps which are indicated in Figure 5.1: The Syntax Analysis step where Polite code is parsed into an AST, the Compilation step where several visitors are applied to alter and ultimately compile the AST into Smalltalk classes, and finally the execution step where the generated code is evaluated in order to compute and return a result value. Special treatment of global methods and messages will be discussed at the end of each subsection.

6.1 Syntax Analysis Step

In this step the Polite source code is parsed by the PSGrammar and PSParser. Using standard parsing techniques they produce an AST that can easily be altered and interpreted by using various visitors.

When the PSGrammar, in which the syntax rules are specified, consumes the input string it creates complex nested arrays consisting of syntax tokens. The structure of these arrays represent the composition of the grammar rules and are therefore not optimized for further computation concerning semantics. That is why the PSParser, a subclass of PSGrammar, decodes these arrays into an AST, creating nodes for meaningful elements e.g. classes, methods, arrays *etc*. The ASTs structure represents the meaning of its elements in the program and is therefore far more adequate to represent a program during

translation.

As an example, look at Listing 6.1 which shows a message sent to `Polite Hero`, the class defined in Figure 1.1, and its AST representation created by the PSParser (Figure 6.1).

```
Polite Hero, new, wins against: some enemy
```

Listing 6.1: A simple message send

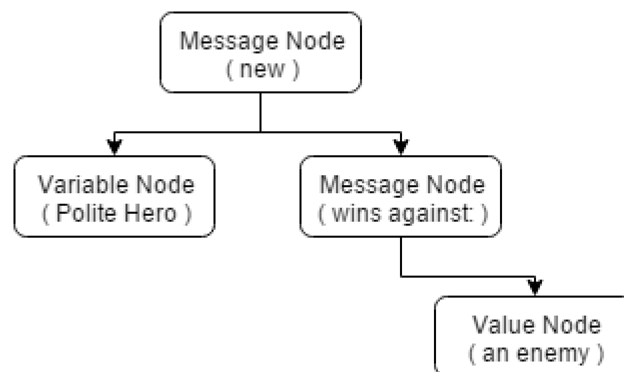

Figure 6.1: Resulting AST

Polite uses PetitParser for its PSParser and PSGrammar implementations. These classes have already existed in the previous iteration and were extended to incorporate class and method definitions as shown in Listings 6.2 to 6.4 for class, method and global keyword method definitions respectively.

```
classDefinition       ←    classDeclaration
                             INDENT
                                 instanceVariables
                                 classBody
                             DEDENT

classDeclaration      ←    identifierName comma
                             subclassToken identifierName

classBody             ←    method*

subclassToken         ←    'subclass:'
```

Listing 6.2: Grammar rules for class definitions

```
method              ←   methodDeclaration
                          INDENT
                              methodSequence
                          DEDENT

methodDeclaration   ←   keywordSentenceMethod /
                          unarySentenceMethod /
                          binaryMethod

methodSequence      ←   Temporaries
                          periodToken*
                          Statements
```

Listing 6.3: Grammar rules for indentation sensitive method definitions

```
globalMethod        ←   method
```

Listing 6.4: Grammar rules for global keyword method definitions

Global *methods*, both unary and keyword, can be defined like any other method but outside of any class definition and therefore not indented.

Global *unary messages* are handled exactly the same as variables are up to this point. They are parsed and stored in the AST as variable nodes since they cannot be distinguished otherwise.

In contrast, global *keyword* messages are defined in the PSGrammar. This means they can be recognized as such in the syntax analysis step. They cannot be represented as a regular message node however, for lack of a receiver. When a global keyword message is recognized, it is also represented as a variable node in the AST. This serves to unify the handling of global keyword and unary messages as they are now both represented as a variable node for the time being.

6.2 Compilation Step

The visitor pattern enables adding new functionality to a set of classes without changing their code directly. This is mostly used in tree data structures like the AST created by the PSParser. The visitor starts at a given root node and traverses the children of that root node, performing an action depending on the type of the child class. This action may alter the data stored in the node, replace it entirely, and/or traverse a list of children itself.

In the compilation step the AST obtained from the parser is traversed three times by different visitors. The first two visitors alter the structure and content of the AST in order

for the third, the compiler, to compile it to viable Smalltalk code.

6.2.1 The Depolitor

The Depolitor traverses the AST from the program node, the root node of the entire tree. It transforms every identifier name to camel case, which includes removing whitespace and capitalising the first letter after each whitespace. Depolitization is needed in order to transform Polite's sentence identifiers to Smalltalk compatible identifiers. This is needed for any identifier to be used in the Smalltalk environment but also enables referencing Smalltalk classes and methods in Polite code.

6.2.2 The PSGlobalMessageSearchVisitor

The message search visitor was newly introduced in this thesis and is responsible for marking global messages for the compiler. It does so by wrapping the variable node, which was created by the PSParser and represents the global message, with a global message node.

The search visitor is initialized with a dictionary of global variables stored in PS-Global, as well as Smalltalk keywords such as *self, super, true and false*. This dictionary represents the identifiers that are never interpreted as global methods.

While traversing the AST from a given root node, the visitors dictionary is adjusted whenever a variable is introduced. This includes variable declaration, class and method definitions *etc*. In this case, a new instance of the visitor is created and the variable is added to the dictionary obtained from the old search visitor. The new visitor is then applied to the children of the current node.

All of this is done in preparation for the eventual variable node which, as explained in Chapter 5, can represent either a variable, a keyword, or a global unary message. The variable node's name is now checked against the visitor's dictionary. If the variable name is in the dictionary, the variable node represents either a variable or a Smalltalk keyword. Either way there is no further computation required since the default interpretation for these is appropriate. If the variable name is not in the dictionary however, it cannot be a variable or keyword and therefore must be a global message. In this case a global message node is wrapped around the variable node in order to mark it as a global message for the compiler. The variable node still represents the name of the message but the information of where to send it is represented by the global message node.

In the example given in Listing 6.1 the parameter `some enemy` is neither defined as a variable globally nor in the class definition. Therefore it has to be a global message. This is detected by the search visitor and the variable node is wrapped inside a global message node. The resulting AST after both the Depolitor and the search visitor have been applied, as well as its Smalltalk representation, printed besides the nodes, are shown in Figure 6.2.

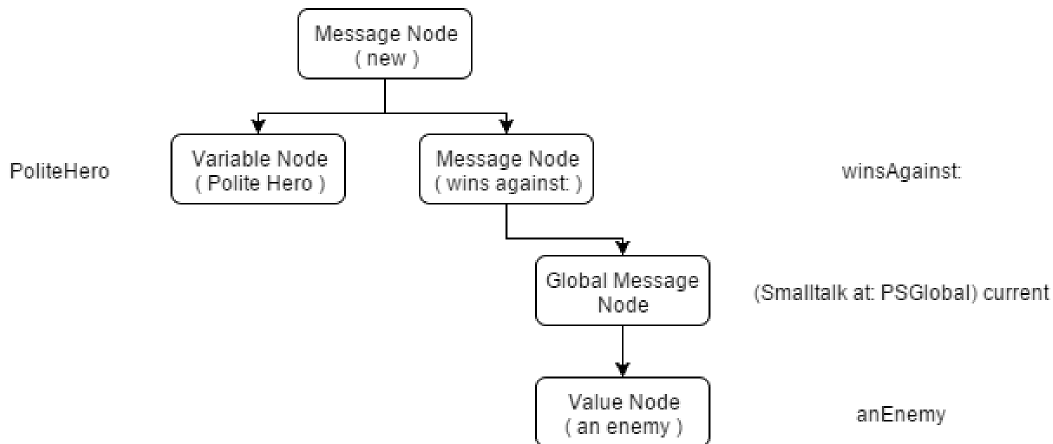

Figure 6.2: The global message is wrapped by a Global Message Node

6.2.3 The PSCompiler

The Polite Compiler was created in a refactoring effort by separating the compiling responsibilities from the PoliteSmalltalk class which works as the interface to the entire application. The compiler was implemented as a visitor to better adapt to the increased complexity of the AST due to the newly integrated elements.

When the compiler is initialized, it creates the `PoliteSmalltalk-UserCreated` package where all user data is stored. In this package, a new PSGlobal class is created and instantiated. The `PSGlobal` class has a class side method called `current` which retrieves its latest instance. It is this instance that stores all globally available data for a translation. Whenever a new Polite compiler is created, or a new translation is initiated, the entire package is reset and populated with the new user created classes.

When applied to the program node, the root node of the AST, the Polite Compiler first stores all global variables to the PSGlobal object and then visits all definitions of the program. The definitions are represented by subtrees and each represent either a class definition or a global method definition. When a class definition is visited, the compiler immediately subclasses a new class appropriately. The class methods are then compiled to the new class. A global method on the other hand is compiled as a class method of PSGlobal. After all definitions have been compiled, the main sequence is compiled as the psMainMethod of PSGlobal. This method is the first to be evaluated in the Execution step.

The compiler only visits the program node, its definitions and main sequence. From there, the BIConfigurableFormatter is employed in order to assemble methods, statements *etc.* The configurable formatter is a visitor used throughout Pharo in order to collect the representation of classes and objects. After implementing visiting methods for the newly introduced program, class and global message nodes, the formatter can compute

representations for every AST node used in Polite. The formatter is employed by the PSCompiler whenever the representation of a subtree is needed in order to compile it. It traverses the subtree, adding each node's representation to a stream which is returned as a string at the end of the computation. This string consists of valid Smalltalk code for the subtree that was visited, and is called the formatted code.

This is where the global message node gets interesting. When the formatter visits such a global message node, it returns the representation shown in Listing 6.5.

```
(Smalltalk at: #PSGlobal) current
```

Listing 6.5: Representation of a global message node

Listing 6.6 displays the formatted code retrieved from the example AST in Figure 6.2 by the formatter. The formatted code is in Smalltalk syntax and clearly represents the message `anEnemy` sent to the current `PSGlobal` object.

```
PoliteHero new winsAgainst: (Smalltalk at: #PSGlobal) current
    anEnemy
```

Listing 6.6: Formatted code of a global message send

6.3 Execution Step

In the execution step the main method of the PSGlobal object is executed. It contains the main sequence of the Polite program and possibly references Polite and/or Smalltalk classes, global methods etc. The result of the main method is returned as the result of the entire translation and can be displayed in the playground or other environments.

In the compilation step, global messages were represented in the formatted code as `(Smalltalk at: #PSGlobal) current` followed by the message name. When evaluated, this code retrieves the current PSGlobal object and sends the global message to it. This completes the transformation of global methods to regular Smalltalk methods stored in and executed from a special global object.

6.4 Approaching Global Methods

Two approaches for recognizing a variable node representing a global message were attempted. The first one tried to check if a method was implemented, and therefore stored in the PSGlobal object, at run time. While this solution worked in practice, the generated code was filled with these checks against the method dictionary and the generated code was near unreadable due to code pollution.

For the same example used before in this chapter (Listing 6.1) the generated code shown in Listing 6.7 is already very polluted. For each variable node, representing any kind of identifier, an if-else block is created, which checks if PSGlobal includes a method selector by that name.

```
(( PSGlobal includesSelector: #PoliteHero )
    ifTrue: [ PSGlobal current PoliteHero ]
    ifFalse: [ PoliteHero ]) new
    winsAgainst:
        (( PSGlobal includesSelector: #anEnemy )
            ifTrue: [ PSGlobal current anEnemy ]
            ifFalse: [ anEnemy ])
```

Listing 6.7: Generated code using the run time approach

For this reason the run time approach was abandoned in favour of a cleaner, more elegant scope tracking approach. This solution employs the GlobalMethodSearchVisitor described in Section 6.2.2. The visitor's purpose is to mark which variable nodes represent in fact a global method, so that the Polite compiler can retrieve its representation accordingly later on.

Listing 6.8 shows the generated code for the same example using the global message search visitor approach. While the parameter `some enemy` could be either a variable or a global message in the polite code (Listing 6.1), `someEnemy` is clearly a message sent to PSGlobal in the generated code (Listing 6.8). Note that if `some enemy` was a variable instead of a global message, the generated code would be even simpler, as shown in Listing 6.9. Compared to the previous approach shown in Listing 6.7, this is a great improvement in code quality.

```
PoliteHero new
    winsAgainst: (Smalltalk at: #PSGlobal) current someEnemy
```

Listing 6.8: Generated code using the search visitor approach

```
PoliteHero new winsAgainst: someEnemy
```

Listing 6.9: Generated code assuming `someEnemy` is not a global message

7 Validation

This chapter will provide three examples of Polite code in order to show that Polite Smalltalk holds what it promises. The first example is an implementation of the vision for Polite which was used for most examples in this thesis, the second one shows two recursive implementations of a global factorial function and the last one is the well known LOGame from Pharo by Example, translated into Polite syntax. The complete source code for these examples are presented in the appendix and can be evaluated using the Polite Playground.

7.1 Polite Vision

This example represents the hero story that is used in almost every example of this thesis. It consists of a `Polite Character` class and its two subclasses `Polite Lady` and `Polite Hero`. It also features two global helper methods `if: then:` and `if: then: else:` which are used to write more polite code. These five make up the definitions of the program while the rest of the statements make up its body.

The example shows that inheritance as well as global methods work as expected. Class methods as well as global methods are recognized correctly and yield the expected results.

7.2 Recursive Factorial

The second example focuses on global methods and how they interact. It shows a unary and a keyword message based implementation of a recursive factorial implementation. In the former (Listing B.2), the parameters are kept global while the later (Listing B.3) only needs a start value and does not need any temporary variables besides this parameter. This example showcases both global methods and global variables and how they can be used to quickly write some small code snippets rather than having to create a separate class.

7.3 LO Game

The LO Game's source code was taken from the Book *Pharo by Example* [1], an external source, and translated to Polite syntax line by line in order to show that Polite works for arbitrary programs. The only addition is a global `delete all` method for deleting all instances of the Game. It also represents a somewhat more complicated program.

7.4 The Polite Playground

The PolitePlayground received some architectural upgrades in order to support the well known inspectIt and printIt functionalities for partial evaluation. The structure of the new playground was greatly inspired by the Pharo Playground. The shortcut for doIt was kept from prior iterations and still parses the entire input and displays the output in the second window of the playground. Highlighting has also been improved and is now highly customisable in the `initialize method` of the `PSHighlighter`. Highlighting helps with reading code as well as finding errors when writing.

For validation purposes the playground proves that all examples used in this thesis behave as intended and described. Note that in some occasions, examples are excerpts and do not show the entire code required in order to work properly. All examples have been tested and validated with the help of the playground.

8
Conclusion and Future Work

This chapter will analyse how far the thesis has come and what is yet to achieve. It will draw conclusions for both the result of the thesis as well as personal growth during this thesis and propose steps to further improve Polite Smalltalk.

8.1 Conclusion

The main goal, defining and implementing class and method definitions for Polite was a complete success. Users can define their own classes, instantiate them and send messages to them just like any other class in the system.

Global methods and messages also work as intended. Future studies will tell if they in fact increase readability but for now the additional functionality is well received.

The compiler infrastructure is more organised than it was in previous iterations. This hopefully leads to easier expansion and future development. Especially applying additional visitors in order to further manipulate the AST is simplified significantly. Also, Smalltalk classes can be used in Polite just like Polite classes as long as their names can be recreated by depolitization.

Nevertheless, a few compromises had to be made in the grammar. For instance the structure of class definitions does not allow for class side methods yet.

Personally this thesis has been a rewarding and illuminating experience. As my first serious contact with Smalltalk and so far deepest involvement with ASTs and visitors, I learned much more about these topics and environments than any lecture could have taught me. Working on a comparatively large project also provided me with

some valuable insights about cleanliness and the process of learning to understand this environment.

8.2 Future Work

Most excitingly, as was the declared objective of the thesis, Polite is ready to be used in the study on sentence identifiers. This study was the main motivation for Polite and now it can be put to the test.

The lack of class side methods does not pose a problem for most small scale applications. Filling this hole could however be an important step for Polite in order to become a more complete language. Class side methods could be included in the class definition and would most likely be implemented by adding several grammar and parser rules as well as adapting the class node and the compiler's class visiting method. The issue with this is that it would most likely need additional keywords, which is generally disliked in the Smalltalk community.

Furthermore the interaction between Polite and its generated code is not yet ideal. The debugger for example runs on the generated Smalltalk code which can be misleading for the developer who only ever works with the corresponding Polite code. Mapping the Polite source code with the corresponding generated Smalltalk code in order to represent the source in the debugger could increase abstraction and hide unnecessary details from Polite users.

Code base management could also be improved. Right now there is no way to save either Polite or generated code beyond the next translation. Exporting both to a file or even inside the image would be a rather simple improvement.

Bibliography

[1] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[2] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[3] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[4] Attieh Sadeghi Givi. Layout sensitive parsing in the PetitParser framework. Bachelor's thesis, University of Bern, October 2013.

[5] Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[6] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques — A Practical Guide*. Springer, 2008.

[7] Jan Kurš, Guillaume Larcheveque, Lukas Renggli, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, September 2013.

[8] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[9] Mircea Lungu and Jan Kurš. On planning an evaluation of the impact of identifier names on the readability and maintainability of programs. In *USER'13: Proceedings of the 2nd Workshop on User evaluations for Software Engineering Researchers*, pages 13 – 15, 2013.

[10] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[11] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, pages 1–4, Malaga, Spain, June 2010.

Anleitung zu wissenschaftlichen Arbeiten

Polite Smalltalk is fully object oriented program language that is parsed and compiled into independent Smalltalk classes in order to be executed in the Pharo Smalltalk environment. In this chapter however, the technical aspect of this translation process is omitted in order to provide a fast and simple entry point to working with Polite.

A.1 Installation

Polite currently runs on Pharo Smalltalk using the Moose 6 image. In order to download these, visit the following site and download the Pharo virtual machine (VM) for your operating system:

```
http://pharo.org/download
```

Unpack the archive and move it to some secure location. Open the following link and download the `moose 6.0 image` or newer.

```
http://www.moosetechnology.org/#install
```

Unpack this zip into your Pharo folder and open the image with Pharo from there.

Open a `Playground` by clicking on the background and selecting it from the world menu. Copy the code shown in Listing A.1 and do it (Alt/Cmd + d) in order to download Polite.

```
Gofer new smalltalkhubUser: 'JanKurs' project: '
   PoliteSmalltalk';
    configurationOf: #PoliteSmalltalk; load.
(Smalltalk at: #ConfigurationOfPoliteSmalltalk)
    perform: #loadPoliteSmalltalk.
```

Listing A.1: Command for downloading Polite

A.2 Using Polite inside Pharo

In order to get to the Polite Playground click on the background and choose `Polite Playground`. This is where you write your code and evaluate your program. The entire code inside the playground is translated and executed by pressing the *accept* button in the top right corner or by pressing alt/cmd + d. The result of the computation is shown in the bottom window of the playground. You can also evaluate code partially by highlighting it and pressing alt/cmd + p to print the result beside the highlighted code or alt/cmd + i to inspect it in an inspector window.

In case you want to look at the Smalltalk code your program generates, you can find it by opening a system browser[1] and navigating to `PoliteSmalltalk-UserCreated` where all your classes are stored. Global methods are stored in the PSGlobal object.

Notice that all classes as well as the PSGlobal object are removed every time you accept your code (alt/cmd + d). This means that in order to work on several projects at once, you either need to keep several images or copy your code out of the playground and store it somewhere else.

A.3 Syntax

Polite syntax is kept very similar to regular Smalltalk syntax. In fact, all Pharo Smalltalk classes can be used in Polite as long as the naming conventions described in section A.4 are observed. Allowing sentence identifiers and global methods as well as relying on the Polite Playground instead of a system browser made some syntax changes inevitable however.

The general rules for Polite Syntax are as follows:

- Each message is separated from its receiver by using a comma. This is essential for allowing sentence identifiers as they could not be differentiated otherwise.

[1] click on the background of the Pharo window and choose system browser

- Polite is also indentation sensitive. This means that every statement of a method's body has to be indented once by using a tab. The same applies for classes, so a statement of a class method would be indented twice.

Listing A.2 shows a class definition as well as a global method definition which together include all important syntactically relevant differences.

```
Polite Character, subclass: Polite Hero                        ①
    drive off: (an enemy) and save: (the lady)                 ②
        if: (self, wins against: an enemy)                     ③
        then: [self, saves: the lady]

if: condition then: (block true)                               ④
    ˆ condition,
    if true: [block true, value]
```

Listing A.2: Syntax of class and method definitions

Method definition syntax is basically identical to the corresponding Smalltalk syntax ②. Consider that the general rules above still apply, so sentence identifiers, comma separation and indentation still have to be taken into account.

Polite knows a special case of method that is defined outside of any class definition and called without a specific receiver. This is called a global method ④. Global methods are used in order to write more polite code as well as really short code snippets in smaller projects. Defining a global method is as easy as not indenting it. The method then does not belong to any class body and therefore is an independent definition, a global method. In order to call a global method, also called sending a global message, simply write down the method's name, omitting its receiver ③. A global message is syntactically identical to a variable call, so pay attention not to have variables with the same name as global methods as they will be called instead.

The Polite class syntax had to be adapted for not using the system browser ②. It therefore looks more similar to its method syntax with instance variables being defined in between '|' characters and separated with a comma. The declaration was kept from Smalltalk however, with the exception that the class name is not preceded by a '#' symbol. Following the indentation rule, every line of code that follows the class declaration and is indented one more than the class declaration is considered part of the class body.

A.4 Conventions

There are two binding conventions in Polite that have to be followed in order to generate valid code. First, class names have to be capitalized. This convention follows directly

from Smalltalk's requirement to have capitalized class names. Because Polite is translated to Smalltalk this rule still has to be observed.

The second convention concerns classes and methods defined in Smalltalk. All of these can be used in Polite as long as the identifiers used can be depolitized[2] correctly to result in the proper Smalltalk identifiers. This means the original Smalltalk identifiers work perfectly fine. It is however more polite[3] to use the Polite convention of separating words with space characters and not necessarily capitalise every word.

[2]computation to create camel cased identifiers from sentence identifiers by removing whitespace and capitalizing each word but the first

[3]politeness is used to measure the cleanness of Polite code

B
Validation Code

B.1 Polite Vision

```
Object, subclass: Polite Character
    | name, STR |

    initialize
        STR := 1

    strength
        ^ STR.

    increase strength
        STR := STR + 1.

    wins against: a character
        ^ self, strength > a character, strength

Polite Character, subclass: Polite Lady
    | freedom  |

    initialize
        freedom := false
```

```
    is freed
        ^ freedom := true.

    is free
        ^ freedom

    expresses her gratitude
        ^ if: (self, is free)
        then: ['the lady is ever thankful to our valiant hero
    ']
        else: ['the lady has not been freed yet']

Polite Character, subclass: Polite Hero
    | health  |

    initialize
        super, initialize.
        health := 5

    drive off: (an enemy) and save: (a lady)
        if: (self, wins against: an enemy)
        then: [a lady, is freed]

    exercises
        self, increase strength.
        health := health + 1

if: condition then: (true block)
    ^ condition,
    if true: [true block, value]

if: condition then: (true block) else: (false block)
    ^ condition,
    if true: [true block, value]
    if false: [false block, value]

| my hero, the bandits, the lovely lady |

my hero := Polite Hero, new.
the bandits := Polite Character, new.
the lovely lady := Polite Lady, new.
```

```
my hero, drive off: (the bandits) and save: (the lovely lady)
    .
the lovely lady, expresses her gratitude.

my hero, exercises.
my hero, drive off: (the bandits) and save: (the lovely lady)
    .
the lovely lady, expresses her gratitude.
```

Listing B.1: The code of the Polite vision

B.2 Recursive Global Methods

```
a global method
    ˆ (index > 0), if true: [
        value := value * index.
        index := index - 1.
        a global method
    ] if false: [
        value
    ].

| index, value |

index := 6.
value := 1.

a global method.
```

Listing B.2: Recursive global unary message

```
a global method: value
    ˆ (value > 0), if true: [
        ˆ value * a global method: value-1
    ] if false: [
        ˆ 1
    ].

a global method: 6.
```

Listing B.3: Recursive global keyword message

B.3 LO Game

```
Simple switch morph, subclass: LO cell
    | mouseAction |

    initialize
        super, initialize.
        self, label: ''.
        self, border width: 2.
        bounds := 0@0, corner: 32@32.
        off Color := Color, pale yellow.
        on Color := Color, pale blue, darker.
        self, use square corners.
        self, turnOff

    mouse action: a Block
        ˆ mouse action := a Block

    mouse up: an event
        mouse action, value

    mouse move: an event
        ˆ self.

Bordered Morph, subclass: LO Game
    | cells |

    initialize
        | sample cell, width, height, n |
        super, initialize.
        n := self, cells per side.
        sample cell := LO Cell, new.
        width := sample cell, width.
        height := sample cell, height.
        self, bounds: (5@5, extent: ((width*n) @(height*n)) +
    (2 *self, border width)).
        cells := Matrix, new: n tabulate: [ :i :j | self, new
    cell at: i at:j ].

    cells per side
        ˆ 10

    new cell at: i at: j
```

```
        | c, origin |
        c := LO Cell, new.
        origin := self, inner bounds, origin.
        self, add morph: c.
        c, position: (((i-1) * c, width)@((j-1)*c, height) +
    origin).
        c, mouse action: [self, toggle neighbours of cell at:
     i at: j].
        ^ c

    toggle neighbours of cell at: i at: j
        (i > 1), if true:
            [ (cells, at: (i-1) at: j ), toggle state ].
        (i < self, cells per side), if true:
            [ ( cells, at: (i+1) at: j ), toggle state ].
        (j > 1), if true:
            [ (cells, at: i at: (j-1)), toggle state ].
        (j < self, cells per side), if true:
            [ (cells, at: i at: (j+1)), toggle state ].

delete all
    LO game, all instances, do: [:each | each, delete]

LO Game, new, open in world.
```

Listing B.4: The LO Game from *Pharo by Example*