Polite Programmers, Use Spaces in Identifiers When Needed

Mircea F. Lungu
Software Architecture Group, University of Groningen
m.f.lungu@rug.nl

Jan Kurš
Software Composition Group, University of Bern
kurs@inf.unibe.ch

Abstract

JavaProgrammersUseCamelCaseToSeparateWordsInIdentifiers.
Pythonistas_and_others_use_underscore_in_their_identifiers.
Polite programmers can use spaces in their identifiers if needed.

Keywords programming languages, design, usability, Smalltalk

1. Introduction

Polite Smalltalk is a small evolutionary mutation of the Smalltalk programming language that aims to encourage developers to think more about their programs as prose. The main mechanism by which Polite does this is what we define here as *sentence case identifiers* — a naming convention that allows spaces in identifier names. Unlike projection editors which can easily allow spaces in their identifiers, Polite programs are stored as text.

Although spaces in identifiers have been used before in DSLs (e.g. Applescript, the Cucumber testing framework, Inform 7) or COBOL, the feature is unusual for a general purpose object-oriented text-based programming language. We suspect that the main reason for this is historical: spaces allow the scanner to easily tokenize the text in a traditional compiler backend architecture. We are interested more in easying the job of the programmer instead.

We believe that sentence case identifiers could impact code readability and we have two hypotheses why this could be:

- A syntax like that of Polite might encourage developers to think more about their programs as prose. This might make them work just a little bit harder towards writing more understandable code.
- A system with sentence case identifiers might be perceived as more user friendly by beginners.

We believe that it is worth investigating these hypotheses, given the impact that even small increases in code readability could have on software development: it is well known that developers spend the vast majority of their time reading rather than writing source code.

Some readers will argue that this is not a very serious problem, since programmers do not use identifiers as long as we illustrated in the abstract. For those readers, we present three method names that can be found in three popular open source programs written in Java, one of the most popular programming languages at the moment:


```
// nakedobjects-4.0.0
whenCallEnsureThatContextOverloadedShouldThrowIll
egalThreadStateExceptionUsingSuppliedMessage

// aspectj-1.6.9
getPointcutParserSupportingSpecifiedPrimitivesAnd
UsingSpecifiedClassLoaderForResolution

// maven-3.0
disabledtestResolveCorrectDependenciesWhenDiffer
entDependenciesOnNewestVersionReplaced
```

2. Illustrating the Polite Syntax

To experiment with sentence case identifiers, we started from the grammar of Smalltalk and we modified all the rules that involve identifiers to allow spaces in identifier names. Once this was done, we discovered several other grammar modifications which we considerd desirable. We named the resulting language Polite Smalltalk or for short, Polite. Others in the past have also started from Smalltalk and provided deltas to augment its syntax (Borning and O'Shea 1987).

In this section we discuss those modifications to the original Smalltalk language that were performed to obtain Polite.

2.1 Sentence Case Identifiers

In Smalltalk, to send to an object a unary message, one simply separates the two with a space, e.g.

```
politeHero rechargeEnergy.
```

Keyword messages are similar but they are separated by their arguments with a column, e.g.

```
politeHero fightWith: anEnemy.
```

If we want to allow spaces in the name of identifiers, we cannot use space to separate the object and the message. In Polite we introduce a comma to separate the name of an agent from the message that is receives, as one would also do in natural language (e.g. *"Alfred, get the Batwing ready"*). All the characters in an identifier including the spaces are relevant for its identity. The previous code snippets thus become:

```
polite hero, recharge energy.
polite hero, fight with: an enemy.
```

For the seasoned Smalltalkers, using a comma in this way might be anathema since the operator is traditionally used to concatenate strings and other collections. However, such a sentiment is misplaced since comma is just a simple message implemented in the Collection class. In Polite, the '+' operator takes over the responsibilities of ',' and is thus used to concatenate two collections.

2.2 Polite Programs

The Smalltalk grammar does not provide productions for programs or classes, since the programmer grows a program by compiling a method at a time in the Smalltalk IDE.

To be able to write programs independent of the Smalltalk IDE we introduce a new grammar rule for a program: a sequence of class and global method definitions, followed by code to be executed. The following code snippet presents a simple program with one class definition and a few lines of code:

```
1  Polite Class, subclass: 'Polite Hero'.
2    | energy level, name |
3
4    recharge energy
5      energy level := 100.
6
7    is dead
8      ^ energy level <= 0
9
10   fight with: an enemy
11     while neither: [self, is dead]
12       nor: [an enemy, is dead]
13        do: [self, throw a punch at: the enemy.
14             the enemy, throw a punch at: self.]
15
16 "The main program"
17 | polite hero |
18 polite hero := Polite Hero, new.
19 polite hero, recharge energy.
```

In the code snippet we see several other features of the language:

- A class declaration (line 1) has the syntax of a message sent to the superclass for creating the subclass.
- Indentation based scoping is used for defining the body of a class and the body of a method.
- Class, program, and method local variable declarations (lines 2 and 17) are enumerated between pipes and must be separated by ',' (line 2)

2.3 First Class Functions

Polite allows the definition of first class functions as opposed to Smalltalk where a method must always be part of a class. These functions are global to the program.

One of the benefits of this, is the possibility of implementing new control structures like the one used in lines 10–14 from the example program:

```
while neither: [self, is dead]
          nor: [an enemy, is dead]
           do: [self, fight with: an enemy]
```

The possibility of defining such ad-hoc control structures, that other languages like Grace or Scala also provide, can be used to improve the readability of program text. This is especially beneficial for several of the conditional structures of Smalltalk which were shown in a study by Stefik to be unintuitive for newcomers to the language (Stefik and Siebert 2013). To illustrate the difference in readability, compare the previous code with the equivalent traditional Smalltalk:

```
((self isDead not) and: [anEnemy isDead not])
        whileTrue: [self fightWith: anEnemy]
```

3. Implementation

Polite is implemented on top of Pharo Smalltalk. The Polite programs are compiled to Smalltalk. The language interpreter is built using PetitParser — a top-down context-sensitive parser combinator framework that uses the parsing expression grammar formalism (Kurš et al. 2014).

The current implementation of Polite can be found online and downloaded from the Zenodo data repository (Kurš et al. 2016). The repository contains an image which includes the code for the language implementation, a suite of more than 450 unit tests, and a sytax-highlighting code editor that is pictured in Figure 1.

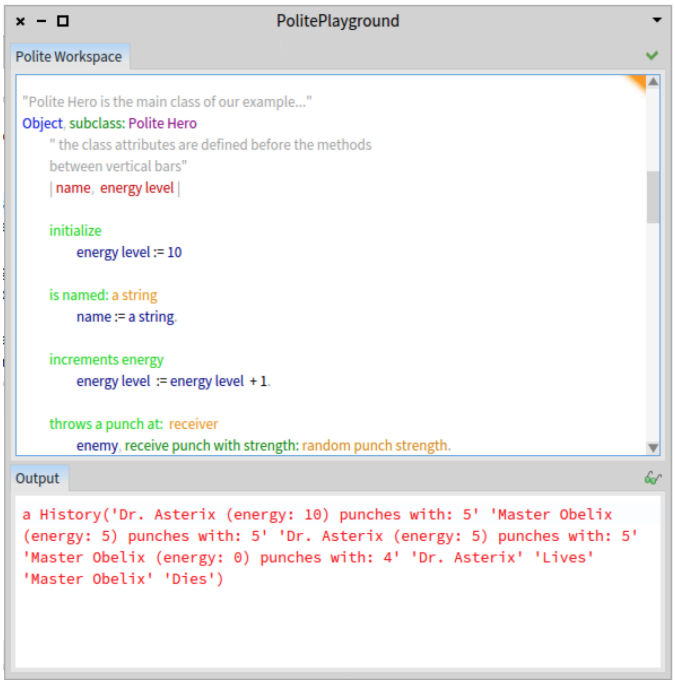

Figure 1. The Polite Playground provides a syntax highlighting code editor for Polite

Acknowledgments

We would like to thank Oscar Nierstrasz and Tijs van der Storm for feedback on earlier versions of this paper.

References

A. Borning and T. O'Shea. *Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language*, pages 1–10. Springer Berlin Heidelberg, 1987. doi: 10.1007/3-540-47891-4_1.

J. Kurš, M. Lungu, and O. Nierstrasz. Top-down parsing with parsing contexts. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.

J. Kurš, M. Lungu, O. Nierstrasz, and T. Steinmann. Polite smalltalk - an implementation, Sept. 2016. URL `https://doi.org/10.5281/zenodo.61578`.

A. Stefik and S. Siebert. An empirical investigation into programming language syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, Nov. 2013. ISSN 1946-6226. doi: 10.1145/2534973.