A critical analysis of string APIs: The case of Pharo

Damien Pollet, Stéphane Ducasse

PII: S0167-6423(17)30243-5
DOI: https://doi.org/10.1016/j.scico.2017.11.005
Reference: SCICO 2169

To appear in: *Science of Computer Programming*

Received date: 15 January 2017
Revised date: 6 November 2017
Accepted date: 9 November 2017

Please cite this article in press as: D. Pollet, S. Ducasse, A critical analysis of string APIs: The case of Pharo, *Sci. Comput. Program.* (2017), https://doi.org/10.1016/j.scico.2017.11.005

A Critical Analysis of String APIs: the Case of Pharo

Damien Pollet[a], Stéphane Ducasse[a]

[a]*RMoD — Inria & Université Lille, France*

Abstract

Most programming languages, besides C, provide a native abstraction for character strings, but string APIs vary widely in size, expressiveness, and subjective convenience across languages. In Pharo, while at first glance the API of the String class seems rich, it often feels cumbersome in practice; to improve its usability, we faced the challenge of assessing its design. However, we found hardly any guideline about design forces and how they structure the design space, and no comprehensive analysis of the expected string operations and their different variations. In this article, we first analyse the Pharo 4 String library, then contrast it with its Haskell, Java, Python, Ruby, and Rust counterparts. We harvest criteria to describe a string API, and reflect on features and design tensions. This analysis should help language designers in understanding the design space of strings, and will serve as a basis for a future redesign of the string library in Pharo.

Keywords: Strings, API, Library, Design, Style

1. Introduction

While strings are among the basic types available in most programming languages, we are not aware of design guidelines, nor of a systematic, structured analysis of the string API design space in the literature. Instead, features tend to accrete through ad-hoc extension mechanisms, without the desirable coherence. However, the set of characteristics that good APIs exhibit is generally accepted [1]; a good API:

- is easy to learn and memorize,
- leads to reusable code,
- is hard to misuse,
- is easy to extend,
- is complete.

To evolve an understandable API, the maintainer should assess it against these goals. Note that while orthogonality, regularity and consistency are omitted, they arise from the ease to learn and extend the existing set of operations. In the case of strings, however, these characteristics are particularly hard to reach, due to the following design constraints.

For a single data type, strings tend to have a large API: in Ruby, the String class provides more than 100 methods, in Java more than 60, and Python's str around 40. In Pharo[1], the String class alone understands 319 distinct messages, not counting inherited methods. While a large API is not always a problem *per se*, it shows that strings have many use cases, from concatenation and printing to search-and-replace, parsing, natural or domain-specific languages. Unfortunately, strings are often abused to eschew proper modeling of structured data, resulting in inadequate serialized representations which encourage a procedural code style[2]. This problem is further compounded by overlapping design tensions:

Mutability: Strings as values, or as mutable sequences.

Abstraction: Access high-level contents (words, lines, patterns), as opposed to representation (indices in a sequence of characters, or even bytes and encodings).

Orthogonality: Combining variations of abstract operations; for instance, substituting one/several/all occurrences corresponding to an index/character/sequence/pattern, in a case-sensitive/insensitive way.

In previous work, empirical studies focused on detecting non-obvious usability issues with APIs [2–4]; for practical advice on how to design better APIs, other works cite guideline inventories built from experience [5, 6]. Joshua Bloch's talk [7] lists a number of interesting rules of thumb, but it does not really bridge the gap between abstract methodological advice (e.g. *API design is an art, not a science*) and well-known best practices (e.g. *Avoid long parameter lists*). Besides the examples set by particular implementations in existing languages like Ruby, Python, or Icon [8], and to the best of our knowledge, we are not aware of string-specific analyses of existing APIs or libraries and their structuring principles.

Email addresses: damien.pollet@inria.fr (Damien Pollet), stephane.ducasse@inria.fr (Stéphane Ducasse)

[1]Numbers from Pharo 4, but the situation in Pharo 3 is very similar.

[2]Much like with Anemic Domain Models, except the string API is complex: http://www.martinfowler.com/bliki/AnemicDomainModel.html