A First Analysis of String APIs: the Case of Pharo

Damien Pollet, Stéphane Ducasse

A First Analysis of String APIs: the Case of Pharo

Damien Pollet Stéphane Ducasse
RMoD — Inria & Université Lille 1
damien.pollet@inria.fr

Abstract

Most programming languages natively provide an abstraction of character strings. However, it is difficult to assess the design or the API of a string library. There is no comprehensive analysis of the needed operations and their different variations. There are no real guidelines about the different forces in presence and how they structure the design space of string manipulation. In this article, we harvest and structure a set of criteria to describe a string API. We propose an analysis of the Pharo 4 String library as a first experience on the topic.

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 [4]; a good API:

- is easy to learn and memorize,
- leads to reuseable 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 [12, 13, 11]; for practical advice on how to design better APIs, these works cite guideline inventories built from experience [2, 6]. 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.

Section 2 shows the problems we face using the current Pharo 4 string library. In Sections 3 and 4, we identify idioms and smells among the methods provided by Pharo's String class. Section 5 examines the relevant parts of the ANSI Smalltalk standard. Finally, we survey string API features in Section 6, before discussing and concluding the paper.

IWST'15, July 15–16, 2015, Brescia, Italy.

ACM 978-1-4503-3857-8/15/07…$15.00.
http://dx.doi.org/10.1145/2811237.2811298

[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

2. Pharo: Symptoms of Organic API Growth

As an open-source programming environment whose development branched off from Squeak, Pharo inherits many design decisions from the original Smalltalk-80 library. However, since the 1980's, that library has grown, and its technical constraints have evolved. In particular, since Squeak historically focused more on creative and didactic experimentation than software engineering and industrial use, the library that has evolved organically more than it was deliberately curated towards a simple and coherent design.

Even though we restrict the scope of the analysis to the String class, we face several challenges to identify recurring structures and idioms among its methods, and to understand and classify the underlying design decisions.

Large number of responsibilities. As explained in Section 1, strings propose a wide, complex range of features. For example, Pharo's String defines a dozen class variables for character and encoding properties.

Large number of methods. The current Pharo String class alone has 319 methods, excluding inherited methods. However, Pharo supports open-classes: a package can define *extension methods* on classes that belong to another package [3, 5]; we therefore exclude extension methods, since they are not part of the core behavior of strings. Still, this leaves 180 methods defined in the package of String. That large number of methods makes it difficult to explore the code, check for redundancies, or ensure completeness of idioms.

Using the code browser, the developer can group the methods of a class into protocols. However, since a method can only belong to one protocol, the resulting classification is not always helpful to the user. For example, it is difficult to know at first sight if a method is related to character case, because there is no dedicated protocol; instead, the case conversion methods are all part of a larger *converting* protocol which bundles conversions to non-string types, representation or encoding conversions, extracting or adding prefixes.

Multiple intertwined behaviors. Strings provide a complex set of operations for which it is difficult to identify a simple taxonomy. Consider the interaction between features: a single operation can be applied to one or multiple elements or the whole string, and can use or return an index, an element, a subset or a subsequence of elements:

Operations: insertion, removal, substitution, concatenation or splitting

Scope: element, pattern occurrence, anchored subsequence

Positions: explicit indices, intervals, matching queries

Occurrences: first, last, all, starting from a given one

In Pharo we can replace all occurrences of one character by another one using the replaceAll:with: inherited from SequenceableCollection, or all occurrences of one character by a subsequence (copyReplaceAll:with:). Like these two messages, some operations will copy the receiver, and some other will change it in place. This highlights that strings are really mutable collections of characters, rather than pieces of text, and that changing the size of the string requires to copy it. Finally, replacing only one occurrence is yet another cumbersome message (using replaceFrom:to:with:startingAt:).

```
'aaca' replaceAll: $a with: $b                              → 'bbcb'
'aaca' copyReplaceAll: 'a' with: 'bz'                       → 'bzbzcbz'
'aaca' replaceFrom: 2 to: 3 with: 'bxyz' startingAt: 2      → 'axya'
```

Lack of coherence and completeness. Besides its inherent complexity, intertwining of behaviors means that, despite the large number of methods, there is still no guarantee that all useful combinations are provided. Some features are surprisingly absent or unexploited from the basic String class. For instance, string splitting and regular expressions, which are core features in Ruby or Python, have long been third-party extensions. They were only recently integrated, so some methods like lines, substrings:, or findTokens: still rely on ad-hoc implementations. This reveals refactoring opportunities towards better composition of independent parts.

Confusingly, some similarly named methods do not accept the same arguments, while some use different wording for similar behavior. For instance, findTokens: and replaceAll:with: accept single characters, but their relatives findTokens:keep: and copyReplaceAll:with: require collection; conversely, compare the predicates isAllDigits but onlyLetters, or asUppercase and asLowercase but withFirstCharacterDownshifted.

Impact of immutability. In some languages such as Java and Python, strings are immutable objects, and their API is designed accordingly. In Smalltalk, strings historically belong in the collections hierarchy, and therefore are mutable.

In practice, many methods produce a modified copy of their receiver to avoid modifying it in place, but either there is no immediate way to know, or the distinction is made by explicit naming. For instance, replaceAll:with: works in-place, while copyReplaceAll:with: does not change its receiver. Moreover, the VisualWorks implementation supports object immutability, which poses the question of how well the historic API works in the presence of immutable strings.

Duplicated or irrelevant code. A few methods exhibit code duplication that should be factored out. For instance, withBlanksCondensed and withSeparatorsCompacted both deal with repeated whitespace, and findTokens: and findTokens:keep: closely duplicate their search algorithm.

Similarly, some methods have no senders in the base image, or provide ad-hoc behavior of dubious utility. For instance, the method comment of findWordStart:startingAt: mentions "HyperCard style searching" and implements a particular pattern match that is subsumed by a simple regular expression.

3. Recurring Patterns

We list here the most prominent patterns or idioms we found among the analyzed methods. Although these patterns are not followed systematically, many of them are actually known idioms that apply to general Smalltalk code, and are clearly related to the ones described by Kent Beck [2]. This list is meant more as a support for discussion than a series of precepts to follow.

Layers of convenience. One of the clearest instances in this study is the group of methods for trimming (Figure 1). Trimming a string is removing unwanted characters (usually whitespace) from one or both of its extremities.

The library provides a single canonical implementation that requires two predicates to identify characters to trim at each end of the string. A first layer of convenience methods eliminates the need for two explicit predicates, either by passing the same one for both ends, or by passing one that disables trimming at one end (trimBoth:, trimLeft:, and trimRight:). A second layer of convenience methods passes the default predicate that trims whitespace (trimLeft, trimBoth, and trimRight). Finally, two additional methods provide concise verbs for the most common case: whitespace, both ends (trim and trimmed, which are synonymous despite the naming).

Convenience methods can also change the result type; the following list shows a few examples of convenience predicates wrapping indexing methods.

Trimming ends trim, trimmed, trimLeft:right:, trimBoth, trimBoth:, trimLeft, trimleft:, trimRight, trimRight:

Index of character indexOf:, indexOf:startingAt:, indexOf:startingAt:ifAbsent:

Index of substring findString:, findString:startingAt:, findString:startingAt:caseSensitive:, and related predicates includesSubstring:, includesSubstring:caseSensitive:

Macro expansion expandMacros, expandMacrosWith: etc., expandMacrosWithArguments:

Sort order compare:, compare:caseSensitive:, compare:with:collated:, and predicates sameAs:, caseInsensitiveLessOrEqual:, and caseSensitiveLessOrEqual:

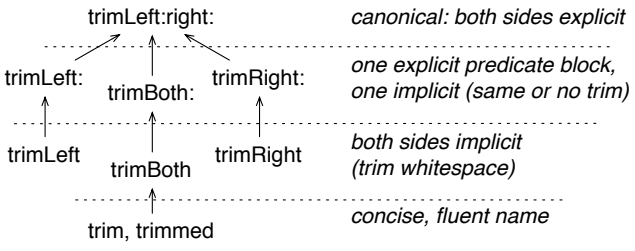

Figure 1. Chains of convenience methods delegating to a single canonical behavior: trimming at one or both ends.

Spelling correction correctAgainst:, correctAgainst:continuedFrom:, correctAgainstDictionary:continuedFrom:, correctAgainstEnumerator:continuedFrom:

Lines lines, lineCount, lineNumber:, lineCorrespondingToIndex:, linesDo:, lineIndicesDo:

Missed opportunity substrings does not delegate to substrings:

This idiom allows concise code when there is a convention or an appropriate default, without giving up control in other cases. However, its induced complexity depends on the argument combinations necessary; it then becomes difficult to check all related methods for consistency and completeness.

We propose to broaden and clarify the use of this idiom wherever possible, as it is an indicator of how flexible the canonical methods are, and promotes well-factored convenience methods. There are several missed opportunities for applying this idiom in String: for instance copyFrom:to: could have copyFrom: (up to the end) and copyTo: (from the start) convenience methods.

Pluggable sentinel case. When iterating over a collection, it is common for the canonical method to expect a block to evaluate for degenerate cases. This leads to methods that are more akin to control flow, and that let the caller define domain computation in a more general and flexible way.

Methods that follow this idiom typically include either ifNone: or ifAbsent: in their selector. For context, in a typical Pharo image as a whole, there are 47 instances of the ifNone: pattern, and 266 instances of ifAbsent:.

Index lookup indexOf:startingAt:ifAbsent:, indexOfSubCollection:startingAt:ifAbsent:

We promote this idiom in all cases where there isn't a clear-cut choice of how to react to degenerate cases. Indeed, forcing either a sentinel value, a Null Object [14], or an exception on user code forces it to check the result value or catch the exception, then branch to handle special cases. Instead, by hiding the check, the pluggable sentinel case enables a more confident, direct coding style. Of course, it is always possible to fall back to either a sentinel, null, or exception, via convenience methods.

Sentinel index value. When they fail, many index lookup methods return an out-of-bounds index; methods like copyFrom:to: handle these sentinel values gracefully. However, indices resulting from a lookup have two possible conflicting interpretations: either *place of the last match* or *last place examined*. In the former case, a failed lookup should return zero (since Smalltalk indices are one-based); in the latter case, one past the last valid index signifies that the whole string has been examined. Unfortunately, both versions coexist:

```
'abc' findString: 'x' startingAt: 1                    → 0
'abc' findAnySubStr: #('x' 'y') startingAt: 1          → 4
```

We thus prefer the pluggable sentinel, leaving the choice to user code, possibly via convenience methods.

Zero index findSubstring:in:startingAt:matchTable:, findLastOccurrenceOfString:startingAt:, findWordStart:startingAt:, indexOf:startingAt:, indexOfFirstUppercaseCharacter, indexOfWideCharacterFrom:to:, lastSpacePosition, indexOfSubCollection:

Past the end findAnySubStr:startingAt:, findCloseParenthesisFor:, findDelimiters:startingAt:

Iteration or collection. Some methods generate a number of separate results, accumulating and returning them as a collection. This results in allocating and building an intermediate collection, which is often unnecessary since the calling code needs to iterate them immediately. A more general approach is to factor out the iteration as a separate method, and to accumulate the results as a special case only. A nice example is the group of line-related methods that rely on lineIndicesDo:; some even flatten the result to a single value rather than a collection.

Collection lines, allRangesOfSubstring:, findTokens:, findTokens:keep:, findTokens:escapedBy:, substrings, substrings:

Iteration linesDo:, lineIndicesDo:

In our opinion, this idiom reveals a wider problem with Smalltalk's iteration methods in general, which do not decouple the iteration per se from the choice of result to build — in fact, collections define a few optimized methods like select:thenCollect: to avoid allocating an intermediate collection. There are many different approaches dealing with abstraction and composeability in the domain of iteration: push or pull values, internal or external iteration, generators, and more recently transducers [10, 9].

Conversion or manipulation. String provides 24 methods whose selector follows the as*Something* naming idiom, indicating a change of representation of the value. Conversely, past participle selectors, e.g. negated for numbers, denote a transformation of the value itself, therefore simply returning another value of the same type. However, this is not strictly followed, leading to naming inconsistencies such as asUppercase vs. capitalized.

Type conversions asByteArray, asByteString, asDate, asDateAndTime, asDuration, asInteger, asOctetString, asSignedInteger, asString, asStringOrText, asSymbol, asTime, asUnsignedInteger, asWideString

Value transformation or escapement asCamelCase, asComment, asFourCode, asHTMLString, asHex, asLegalSelector, asLowercase, asPluralBasedOn:, asUncommentedCode, asUppercase

Past participles read more fluidly, but they do not always make sense, e.g. commented suggests adding a comment to the receiver, instead of converting it to one. Conversely, adopting as*Something* naming in all cases would be at the price of some contorted English (asCapitalized instead of capitalized).

4. Inconsistencies and Smells

Here we report on the strange things we found.

Redundant specializations. Some methods express a very similar intent, but with slightly differing parameters, constraints, or results. When possible, user code should be rewritten in terms of a more general approach; for example, many of the pattern-finding methods could be expressed as regular expression matching.

Substring lookup findAnySubStr:startingAt: and findDelimiters:startingAt: are synonymous if their first argument is a collection of single-character delimiters; the difference is that the former also accepts string delimiters.

Character lookup indexOfFirstUppercaseCharacter is redundant with SequenceableCollection»findFirst: with very little performance benefit.

Ad-hoc behavior. Ad-hoc methods simply provide convenience behavior that is both specific and little used. Often, the *redundant specialization* also applies.

Numeric suffix numericSuffix has only one sender in the base Pharo image; conversely, it is the only user of stemAndNumericSuffix and endsWithDigit; similarly, endsWithAColon has only one sender.

Finding text findLastOccurrenceOfString:startingAt: has only one sender, related to code loading; findWordStart:startingAt: has no senders.

Find tokens findTokens:escapedBy: has no senders besides tests; findTokens:includes: has only one sender, related to email address detection; findTokens:keep: only has two senders.

Replace tokens copyReplaceTokens:with: has no senders and is convenience for copyReplaceAll:with:asTokens:; redundant with regular expression replacement.

Miscellaneous lineCorrespondingToIndex

Mispackaged or misclassified methods. There are a couple methods that do not really belong to String:

- asHex concatenates the literal notation for each character (*e.g.,* 16r6F) without any separation, producing an ambiguous result; it could be redefined using flatCollect:.
- indexOfSubCollection: should be defined in SequenceableCollection; also, it is eventually implemented in terms of findString:, which handles case, so it is not a simple subsequence lookup.

Many ad-hoc or dubious-looking methods with few senders seem to come from the completion engine; the multiple versions and forks of this package have a history of maintenance problems, and it seems that methods that should have been extensions have been included in the core packages.

Misleading names. Some conversion-like methods are actually encoding or escaping methods: they return another

string whose contents match the receiver's, albeit in a different representation (uppercase, lowercase, escaped for comments, as HTML...).

Duplicated code. Substring testing methods beginsWithEmpty:caseSensitive: and occursInWithEmpty:caseSensitive: are clearly duplicated: they only differ by a comparison operator. They are also redundant with the generic beginsWith:, except for case-sensitivity. Moreover, the –WithEmpty: part of their selector is confusing; it suggests that argument is supposed to be empty, which makes no sense. Finally, their uses hint that were probably defined for the completion engine and should be packaged there.

5. The ANSI Smalltalk Standard

The ANSI standard defines some elements of the Smalltalk language [1]. It gives the definition *"String literals define objects that represent sequences of characters."* However, there are few guidelines helpful with designing a string API.

The ANSI standard defines the readableString protocol as conforming to the magnitude protocol (which supports the comparison of entities) and to the sequencedReadableCollection protocol, as shown in Figure 2 [1, section 5.7.10]. We present briefly the protocol sequencedReadableCollection.

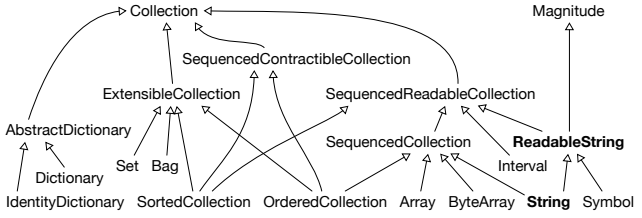

Figure 2. Inheritance of the ANSI Smalltalk protocols.

SequencedReadableCollection. The sequencedReadableCollection protocol conforms to the collection protocol; it provides behavior for reading an ordered collection of objects whose elements can be accessed using external integer keys between one and the number of elements in the collection. It specifies that the compiler should support the following messages — we add some of the argument names for clarity:

Concatenation: **,** tail (the *comma* binary message)

Equality: = other

Element access: at: index, at: index ifAbsent: block, first, last, before: element, after:, findFirst: block, findLast:

Subsequence access: from: startIndex to: stopIndex do: block

Transforming: reverse

Substitution: copyReplaceAll: elements with: replacingElements, copyReplaceFrom: startIndex to: stopIndex with: replacingElements, copyReplacing: targetElement withObject: replacingElement, copyReplaceFrom: startIndex to: stopIndex withObject: replacingElement

Index of element(s): indexOf: element, indexOf:ifAbsent:, indexOfSubCollection:startingAt:, indexOfSubCollection:startingAt:ifAbsent:

Copy: copyFrom: startIndex to: lastIndex, copyWith: element, copyWithout:

Iteration: do:, from:to:keysAndValuesDo:, keysAndValuesDo:, reverseDo:, with:do:

Many operations require explicit indices that have to be obtained first, making the API not very fluid in practice. Moreover, the naming is often obscure: for example, copyWith: copies the receiver, and *appends* its argument to it.

ReadableString. This protocol provides messages for string operations such as copying, comparing, replacing, converting, indexing, and matching. All objects that conform to the readableString protocol are comparable. The copying messages inherited from the sequencedReadableCollection protocol keep the same behavior. Here is the list of messages:

Concatenation: **,** (comma)

Comparing: <, <=, >, >=

Converting: asLowercase, asString, asSymbol, asUppercase

Substituing: copyReplaceAll:with:, copyReplaceFrom:to:with:, copyReplacing:withObject:, copyWith:

Subsequence access: subStrings: separatorCharacters

Testing: sameAs:

Analysis and ANSI Compliance. Indices are omnipresent, and very few names are specific to strings as opposed to collections, which makes the protocol feel shallow, low-level and implementation revealing. In particular, because the underlying design is stateful, the copyReplace* messages have to explicitly reveal that they do not modify their receiver through cumbersome names. In a better design, naming would encourage using safe operations over unsafe ones.

We believe that the value added by complying with the ANSI standard is shallow. Indeed, the standard has not been updated to account for evolutions such as immutability, and it does not help building a fluent, modern library. ANSI should not be followed for the design of a modern String library.

6. An Overview of Expected String Features

Different languages do not provide the exact same feature set[3], or the same level of convenience or generality. However, comparing various programming languages, we can identify the main behavioral aspects of strings. Note that these aspects overlap: for instance, transposing a string to upper-case involves substitution, and can be performed in place or return a new string; splitting requires locating separators and extracting parts as smaller strings, and is a form of parsing.

Extracting. Locating or extracting parts of a string can be supported by specifying either explicit indices, or by

[3] They can even rely on specific syntax, like Ruby's string interpolation.

matching contents with various levels of expressiveness: ad-hoc pattern, character ranges, regular expressions.

Splitting. Splitting strings into chunks is the basis of simple parsing and string manipulation techniques, like counting words or lines in text. To be useful, splitting often needs to account for representation idiosyncrasies like which characters count as word separators or the different carriage return conventions.

Merging. The reverse of splitting is merging several strings into one, either by concatenation of two strings, or by joining a collection of strings one after another, possibly with separators.

Substituting. The popularity of Perl was built on its powerful pattern-matching and substitution features. The difficulty with substitution is how the API conveys whether one, many, or all occurrences are replaced, and whether a sequence of elements or a single element is replaced.

Testing. Strings provide many predicates, most importantly determining emptiness, or inclusion of a particular substring, prefix or suffix. Other predicates range from representation concerns, like determining if all characters belong to the ASCII subset, or of a more ad-hoc nature, like checking if the string is all uppercase or parses as an identifier.

Iterating. Strings are often treated as collections of items. In Pharo a string is a collection of characters and as such it inherits all the high-level iterators defined in SequenceableCollection and subclasses. Similarly, Haskell's Data.String is quite terse (just 4 or so functions), but since strings are Lists, the whole panoply of higher-level functions on lists are available: foldr, map, etc.

Endogenous conversion. Strings can be transformed into other strings according to domain-specific rules: this covers encoding and escaping, case transpositions, pretty-printing, natural language inflexion, etc.

Exogenous conversion. Since strings serve as a human-readable representation or serialization format, they can be parsed back into non-string types such as numbers, URLs, or file paths.

Mutating vs copying. Strings may be considered as collections and provide methods to modify their contents in-place, as opposed to returning a new string with different contents from the original. Note that this point is orthogonal to the other ones, but influences the design of the whole library.

Mutating strings is dangerous, because strings are often used as value objects, and it is not clear at first sight if a method has side-effects or not. For example, in translateToUppercase, the imperative form hints that it is an in-place modification, but not in trim. Also, safe transformations often rely on their side-effect counterpart: for instance, the safe asUppercase sends translateToUppercase to a copy of its receiver.

In the case of strings, we believe methods with side effects should be clearly labeled as low-level or private, and their use discouraged; moreover, a clear and systematic naming convention indicating the mutable behavior of a method would be a real plus. Finally, future developments of the Pharo VM include the Spur object format, which supports immutable instances; this is an opportunity to make literal strings safe[4], and to reduce copying by sharing character data between strings.

7. Discussion and Perspectives

In this paper, we assess the design of character strings in Pharo. While strings are simple data structures, their interface is surprisingly large. Indeed, strings are not simple collections of elements; they can be seen both as explicit sequences of characters, and as simple but very expressive values from the domain of a language or syntax. In both cases, strings have to provide a spectrum of operations with many intertwined characteristics: abstraction or specialization, flexibility or convenience. We analyze the domain and the current implementation to identify recurring idioms and smells.

The idioms and smells we list here deal with code readability and reuseability at the level of messages and methods; they fall in the same scope as Kent Beck's list [2]. While the paper focuses on strings, the idioms we identify are not specific to strings, but to collections, iteration, or parameter passing; modulo differences in syntax and style usages, they apply to other libraries or object-oriented programming languages. To identify the idioms and smells, we rely mostly on code reading and the usual tools provided by the Smalltalk environment. This is necessary in the discovery stage, but it raised several questions:

- How to document groups of methods that participate in a given idiom? As we say in Section 2, method protocols are not suitable: they partition methods by feature or theme, but idioms are overlapping patterns of code factorization and object interaction.
- How to specify, check and enforce idioms in the code? This is related to architecture conformance techniques [7].

Our goal is not to provide definitive solutions or recommendations, but rather to provide a starting point for discussion around the complexity of the design space, and towards more understandable, reusable, and robust APIs.

References

[1] ANSI, New York. *American National Standard for Information Systems – Programming Languages – Smalltalk, ANSI/INCITS 319-1998*, 1998. http://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf.

[2] K. Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[4] While clever uses for mutable literals have been demonstrated in the past, we think it is a surprising feature and should not be enabled by default.

[3] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, Dec. 2005.

[4] J. Blanchette. The little manual of API design. http://www4.in.tum.de/~blanchet/api-design.pdf, June 2008.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.

[6] K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .Net Libraries*. Addison-Wesley Professional, first edition, 2005.

[7] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[8] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, Dec. 1996.

[9] R. Hickey. Clojure transducers. http://clojure.org/transducers.

[10] S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, Jan. 1996.

[11] M. Piccioni, C. A. Furia, and B. Meyer. An empirical study of API usability. In *IEEE/ACM Symposium on Empirical Software Engineering and Measurement*, 2013.

[12] J. Stylos, S. Clarke, and B. Myers. Comparing API design choices with usability studies: A case study and future directions. In P. Romero, J. Good, E. A. Chaparro, and S. Bryant, editors, *18th Workshop of the Psychology of Programming Interest Group*, pages 131–139. University of Sussex, Sept. 2006.

[13] J. Stylos and B. Myers. Mapping the space of API design decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 50–57, 2007.

[14] B. Woolf. Null object. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 5–18. Addison Wesley, 1998.

Appendix — Classifying the Pharo String API

Finding

Methods returning places in the string (indices, ranges).

findString:	findString:startingAt:
findString:startingAt:caseSensitive:	
findLastOccurrenceOfString:startingAt:	
allRangesOfSubString:	findAnySubStr:startingAt:
findCloseParenthesisFor:	findDelimiters:startingAt:
findWordStart:startingAt:	no senders
findIn:startingAt:matchTable:	auxiliary method
findSubstring:in:startingAt:matchTable:	auxiliary method
findSubstringViaPrimitive:in:startingAt:matchTable:	one sender
indexOf:	indexOf:startingAt: indexOf:startingAt:ifAbsent:
indexOfSubCollection:	mispackaged
indexOfSubCollection:startingAt:ifAbsent:	
indexOfFirstUppercaseCharacter	redundant, one sender
indexOfWideCharacterFrom:to:	
lastSpacePosition	
lastIndexOfPKSignature:	adhoc or mispackaged
skipAnySubStr:startingAt:	skipDelimiters:startingAt:

Extracting

Methods returning particular substrings.

wordBefore:	
findSelector	mispackaged, specific to code browser
findTokens:	
findTokens:escapedBy:	no senders (besides tests)
findTokens:includes:	one sender
findTokens:keep:	
lineCorrespondingToIndex:	
squeezeOutNumber	ugly parser, one sender
splitInteger	what is the use-case?
stemAndNumericSuffix	duplicates previous method

Splitting

Methods returning a collection of substrings.

lines	
subStrings:	
substrings	not a call to previous one, why?
findBetweenSubStrs:	
keywords	adhoc, assumes receiver is a selector

Enumerating

linesDo: lineIndicesDo: tabDelimitedFieldsDo:

Conversion to other objects

Many core classes such as time, date and duration that have a compact and meaningful textual description extend the class String to offer conversion from a string to their objects. Most of them could be packaged with the classes they refer to, but splitting a tiny core into even smaller pieces does not make a lot of sense, and there are legitimate circular dependencies

in the core: a string implementation cannot work without integers, for example. Therefore, most of these methods are part of the string API from the core language point of view:

asDate asNumber asString asSymbol
asTime asInteger asStringOrText
asDuration asSignedInteger asByteArray
asDateAndTime asTimeStamp

Some other methods are not as essential:

asFourCode romanNumber string stringhash

Conversion between strings

A different set of conversion operations occurs between strings themselves.

- typography and natural language: asLowercase, asUppercase, capitalized, asCamelCase, withFirstCharacterDownshifted, asPluralBasedOn:, translated, translatedIfCorresponds, translatedTo:
- content formatting: asHTMLString, asHex, asSmalltalkComment, asUncommentedSmalltalkCode,
- internal representation: asByteString, asWideString, asOctetString

Streaming

printOn: putOn: storeOn:

Comparing

caseInsensitiveLessOrEqual: caseSensitiveLessOrEqual:
compare:with:collated: compare:caseSensitive:
compare: sameAs:

Testing

endsWith: endsWithAnyOf:
startsWithDigit endsWithDigit endsWithAColon ad-hoc
hasContentsInExplorer should be an extension
includesSubstring:caseSensitive: includesSubstring:
includesUnifiedCharacter hasWideCharacterFrom:to:
isAllDigits isAllSeparators isAllAlphaNumerics
onlyLetters inconsistent name
isString isAsciiString isLiteral
isByteString isOctetString isLiteralSymbol
isWideString
beginsWithEmpty:caseSensitive: bad name, duplicate
occursInWithEmpty:caseSensitive: bad name, mispackaged

Querying

lineCount lineNumber:
lineNumberCorrespondingToIndex: leadingCharRunLengthAt:
initialIntegerOrNil numericSuffix indentationIfBlank:
numArgs selector-related
parseLiterals contents of a literal array syntax

Substituting

copyReplaceAll:with:asTokens: copyReplaceTokens:with:
expandMacros expandMacrosWithArguments:
expandMacrosWith: expandMacrosWith:with:
expandMacrosWith:with:with:
expandMacrosWith:with:with:with:
format:
replaceFrom:to:with:startingAt: primitive
translateWith: translateFrom:to:table:
translateToLowercase translateToUppercase

Correcting

correctAgainst: correctAgainst:continuedFrom:
correctAgainstDictionary:continuedFrom:
correctAgainstEnumerator:continuedFrom:

Operations

contractTo: truncateTo:
truncateWithElipsisTo:
encompassLine: encompassParagraph:
withNoLineLongerThan:
withSeparatorsCompacted withBlanksCondensed
withoutQuoting
withoutLeadingDigits withoutTrailingDigits
withoutPeriodSuffix withoutTrailingNewlines
padLeftTo: padLeftTo:with:
padRightTo: padRightTo:with:
padded:to:with: duplicates the two previous
surroundedBy: surroundedBySingleQuotes
trimLeft:right: trim trimmed
trimLeft trimBoth trimRight
trimLeft: trimBoth: trimRight:

Encoding

convertFromEncoding: convertFromWithConverter:
convertToEncoding: convertToWithConverter:
convertToSystemString encodeDoublingQuoteOn:
withLineEndings: withSqueakLineEndings
withUnixLineEndings withInternetLineEndings
withCRs convenience, used a lot

Matching

alike: howManyMatch: similarity metrics
charactersExactlyMatching: bad name: common prefix length
match: startingAt:match:startingAt:

Low-Level Internals

hash typeTable
byteSize byteAt: byteAt:put:
writeLeadingCharRunsOn:

Candidates for removal

While performing this analysis we identified some possibly obsolete methods.

asPathName asIdentifier: asLegalSelector
do:toFieldNumber:
indexOfFirstUppercaseCharacter