Characterizing Pharo Code: A Technical Report

Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil

Characterizing Pharo Code: A Technical Report

Oleksandr Zaitsev[1,2], Stéphane Ducasse[1], and Nicolas Anquetil[1]

[1]Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
[2]Arolla SAS
{*oleksandr.zaitsev, stephane.ducasse, nicolas.anquetil*}*@inria.fr*

Abstract

Pharo (https://pharo.org/) is a modern dynamically-typed reflective pure object-oriented language. It is inspired from Smalltalk. Its unconventional syntax mimics natural language: arguments are not grouped around parentheses at the end but within the message, making expressions looks like sentences. In addition, all control flow operations are expressed as messages and the programmer can freely define new ones and as such define Domain Specific Languages for his task. In this technical report we discuss the statistical properties of source code that people write using Pharo programming language. We present the methodology and tooling for analysing source code selected from the projects of Pharo ecosystem. By analysing 50 projects, consisting of 824 packages, 13,935 classes, and 151,717 methods, we answer questions such as *"what is a typical method length?"* or *"what percent of source code are literals?"*.

1 Introduction

Pharo (https://pharo.org/) is a modern dynamically-typed reflective pure object-oriented language [Black et al., 2009]. It is inspired from Smalltalk[Goldberg and Robson, 1989]. Its unconventional syntax mimics natural language [Kay, 1993]: arguments are not grouped around parentheses at the end but within the message, making expressions looks like sentences. In addition, all control flow operations are expressed as messages and the programmer can freely define new ones and as such define Domain Specific Languages for his task [Bergel, 2016, Bunge, 2009].

Sometimes in our research, we need to make certain claims about source code. For example, *"most methods are short"* or *"most literals are numbers"*. In this report, we provide numbers to support such claims. Did you know that 50% of methods in Pharo have no more than 3 lines of code? Or that 25% of classes have just 3 methods? This report provides some answers, a methodology and tooling to build more experiences.

With the help of some of active contributors of the Pharo open-source community, we have selected 50 projects (such as Iceberg, Roassal, etc.) that are representative of the Pharo ecosystem.

In this report, we propose a methodology to obtain and preprocess source code of projects for statistical analysis. In addition, we have implemented several open-source tools to support such analyses. You can find the list of those tools at the end of this report in Section 8.

Using such methodology and tools, we have collected the source code of the 50 selected projects and used it to answer some important statistical questions. We document the results of our exploration.

The rest of this report is structured in the following way:

Section 2. Peculiar Syntax of Pharo
We start by briefly introducing Pharo programming language and explaining why its syntax is very different from most other languages. If we want to explain why Pharo programmers write code in a certain way, it is important to understand the peculiarity of Pharo's syntax.

Section 3. Code-Related Questions
In this section, we present the questions that we will be answering with our analysis. We group our questions into several categories, each of which will be referenced in a separate section.

Section 4. Collecting the Data
Here we describe the process of data collection. We provide a full list of projects that were chosen for our study, together with instructions on how to load the versions of those projects that we have analysed and reproduce our results. Then we explain the process of tokenization and subtokenization that were part of our data preprocessing.

Section 5. Answering Questions
Once we have collected the data, we use it to answer the questions that were presented in the previous sections. We discuss every group of questions in a separate subsection.

Section 6. Interesting Findings
In this section, we briefly summarize some interesting discoveries that we made with our study.

Section 7. Final Words
We conclude our report and present some additional questions that can be answered in the future work.

Section 8. Tools
In this section, you will find the list of tools that we have developed while working on this report. For each tool we provide the URL of its open-source repository.

2 The Peculiar Syntax of Pharo

Pharo is a dynamic, reflective, pure object-oriented programming language in the tradition of Smalltalk [Goldberg, 1984, Black et al., 2009] and a powerful development environment, focused on simplicity and immediate feedback. The following points are the main reasons why we chose Pharo as a language for case studies in this research:

Object-oriented and dynamically-typed. Pharo is a pure object-oriented programming language: even core elements such as Booleans, or methods are plain objects. In addition, Pharo is dynamically-typed: there is no need to specify static types during the definition of classes.

Favors polymorphism. Developers who use Pharo are encouraged to override existing methods [Ingalls, 1981]. The Pharo's culture among developers is to build polymorphic objects to avoid explicit type check [Demeyer et al., 2002]. The core library provides many hooks and templates to support polymorphic behavior [Gamma et al., 1993, Gamma et al., 1995, Black et al., 2009].

Inplace arguments in method name. Arguments are not grouped at the end of a function call as in C like language but introduced in between inside the function/method names.

Here is an example. The following java expression:

```
bob.send(email,emma);
```

is equivalently expressed in Pharo as follows:

```
bob send: email to: emma.
```

In Pharo, invocation arguments are not grouped at the end surrounded by parentheses but placed along the message. Here the Pharo message is send:to: and it is composed of two parts: send: and to: and the arguments email and emma are positioned just after each message parts.

Consider the following two lines of code. First line is written in Java:

```
Date date = new Date(2019, 6, 1);
```

And here is the same expression expressed in Pharo:

```
date := Date year: 2019 month: 6 day: 1.
```

`year: month: day:` is a single method name that takes three arguments. Notice how the Smalltalk way of dealing with arguments encourages programmers to choose longer and more descriptive method names that describe each argument and make the whole line easier to understand.

Syntax for kids mimicking English. Pharo has the same syntax as Smalltalk[Goldberg and Robson, 1989, Black et al., 2009]. Smalltalk's syntax was designed for kids [Kay, 1972, Kay, 1977, Kay, 1993]. The entire syntax of Pharo fits on a postcard. Smalltalk was designed to resemble English statements. Period is used as a separator instead of semi-colons.

Simpler or different abstractions. In addition, Pharo proposed alternate and simplified iterators. To illustrate this, consider that we need to perform some action 10 times. In Java or C, one would use a `for` loop:

```
for (int i = 0; i < 10; ++i) {
    \\ action
}
```

But in Pharo we say to the number that it should repeat the action:

```
10 timesRepeat: [ "action" ].
```

Disguised as sentence. Using spaces instead of dots between a method's invocation and its receiver, as well as ending statements with dots instead of semicolons, makes Pharo code surprisingly similar to human language and easy to read:

```
1 second asDelay wait.
clock value: Time now.
42 factorial decimalDigitLength.
```

Short methods. Methods in Pharo are very short. As you will see in the following sections, 50% of methods in our dataset have no more than 3 lines of code. Such method length favors superclass code reuse.

Language constructs are identifiers. All the language control flow operators (if, while, loops) are not part of the syntax but are simple methods. They are executed in reaction to messages sent to number, Booleans, collections or lexical closures. Nearly every computation in Pharo is expressed as a message send (return and assignment are not messages).

A condition. Conditions are messages sends to Booleans. The following snippet performs the lexical closure argument (delimited by [and]) only if o isVShrinking returns a true value.

```
o isVShrinking ifTrue: [ o privateExtent: self outerBounds extent ]
```

Every class can be extended by defining new methods and adapted to the needs of programmer. In addition, developers have the freedom to extend the core libraries with any method they need. In particular extra control flow operators (as it is the case for iterators). It means that most of the code is composed out of identifiers that are written by developers.

The following snippet illustrates the use of the iterator named do:separatedBy: that performs a loop and in addition performs an extra action only in between the processed elements.

```
flags
  do: [ :each | self flagSynopsis: each ]
  separatedBy: [ formatter space ]
```

DSL. Pharo developers often use this possibility of extension to define embedded DSL [Bergel, 2016, Bunge, 2009]. The following script describes a Roassal interactive visualisation.

```
v := RTView new.
shape := RTBox new color: Color blue trans.
elements := shape elementsOn: (1 to: 50).
v addAll: elements.
elements @ RTPopup.

RTEdgeBuilder new
  view: v;
  objects: (1 to: 50);
  connectFrom: [ :i | i // 3 ].
RTTreeLayout on: elements.
```

Community. Pharo has a small and highly responsive community that help us by providing feedback on the suggested names, selecting projects for the analysis, and discussing naming conventions.

3 Code-Related Questions

In this section, we discuss the five groups of questions that we want to answer in our study. We did not try to come up with an exhaustive list of questions, but simply explored the properties of source code that are most interesting for us. With our questions we mainly focus on the structural (*what belongs where?*) and semantic (*what does this word mean?*) aspects of code.

1. **How big are the projects?**

   To start our analysis, we need to understand the size of the projects that we are working with. To do that, for every projects in our dataset, we report its

   (a) Number of packages

   (b) Number of classes

   (c) Number of methods

(d) Number of lines of code

(e) Number of tokens

This allows us to understand how relevant and statistically significant those projects are for our analysis. And it's also interesting to compare those projects to each other.

2. **How many tests are available?**

   To get a quick insight on how well are the projects tested, for each project we also report its:

   (a) Number of test packages

   (b) Number of test classes

   (c) Number of test methods

   (d) Proportion of test/non-test methods

3. **How much is too many?**

   We continue by exploring the typical sizes of packages, classes, and methods in the projects and identifying the ones that are too large. We study the statistical distribution of the following qualities:

   (a) Number of classes per package

   (b) Number of methods per class

   (c) Number of lines of code per method

   (d) Number of tokens per method

   This allows us to identify their typical ranges and find packages, classes, and methods that are too large and probably should be refactored.

4. **What is the code made of?**

   (a) Distribution across different types of tokens (literals, identifier names, etc.)

   (b) Distribution of literals across string, number, character, etc.

   (c) Distribution of identifier names across class names, method names, etc.

5. **What is the "englishness" of the code?**

   (a) What is the size of vocabulary used to construct identifier names? (number of unique words)

   (b) How does it compare to the literary English texts? (greater or smaller vocabulary?)

   (c) How many of those are recognised as valid English words?

4 Collecting the Data

To select projects from the Pharo ecosystem, we took into account several criteria:

1. **Domain variety** - the projects that we have selected for this study cover a wide range of domains, including Web development, Network management, UI frameworks, Graphics, Data Visualization, Scientific Computing, Version Control Systems, Data Collections, Text Processing, File Management, etc.
2. **The use of the project** - some projects are used a lot while others have a small user base.
3. **Size** - some projects such as Moose or Seaside are large and composed of multiple subprojects, while some others are composed of a couple of packages.
4. **Activity** - some projects are under active development while some others are stable and with limited contributions.

For this report, we have collected source code from 50 projects written in Pharo programming language. In total, our dataset contains 824 packages, 13,935 classes, and 151,717 methods. For each project, we selected packages whose name starts with a certain prefix, associated with this project, and loaded all classes and methods from these packages. For example, package names of project *Roassal3* start with prefix *"Roassal3"* and packages of *PolyMath* have prefix *"Math"* in their names. 34 of the collected projects are integrated into Pharo 7 environment. We call them *internal projects*. You can see the list of the internal projects used for this study together with their associated prefixes in Table 1.

	Project	Prefixes
1	System	System-
2	Commander	Commander-
3	Debugger	Debugger
4	Epicea	Epicea
5	Fuel	Fuel-
6	Graphics	Graphics-
7	Kernel	Kernel
8	Keymapping	Keymapping-
9	Metacello	Metacello
10	Monticello	Monticello
11	Network	Network-
12	Calypso	Calypso-
13	Collections	Collections-
14	Morphic	Morphic-
15	AST	AST-
16	Athens	Athens-
17	OpalCompiler	OpalCompiler-
18	OSWindow	OSWindow-
19	Ombu	Ombu
20	Polymorph	Polymorph-
21	Refactoring	Refactoring-
22	Refactoring2	Refactoring2-
23	Reflectivity	Reflectivity
24	Regex	Regex-
25	Renraku	Renraku
26	Rubric	Rubric
27	STON	STON-
28	SUnit	SUnit-
29	Settings	Settings-
30	Shift	Shift-
31	Text	Text-
32	Tool	Tool-
33	UnifiedFFI	UnifiedFFI
34	Zinc	Zinc-

Table 1: 34 projects integrated into Pharo environment that we have collected into our dataset.

Additionally, we have loaded 16 projects from external GitHub repositories. We refer to them as *external projects*. Table 2 lists the external projects together with their prefixes. Table 3 describes

also the URL addresses of their repositories, and SHA of commits specifying the exact version of each project that we have loaded.

	Project	Prefixes
1	DrTests	DrTests
2	Mustache	Mustache
3	PetitParser	Petit
4	Pillar	Pillar
5	Seaside3	Seaside JQuery Javascript
6	Spec2	Spec2
7	PolyMath	Math
8	TelescopeCytoscape	Telescope
9	Voyage	Voyage
10	Bloc	Bloc
11	DataFrame	DataFrame
12	Roassal2	Roassal2
13	Roassal3	Roassal3
14	Moose	Fame Famix Moose
15	GToolkit	GToolkit
16	Iceberg	Iceberg

Table 2: 16 external projects that we have collected into our dataset. We selected all packages whose names start with a given prefix.

	Project	Repository	Commit
1	DrTests	https://github.com/juliendelplanque/DrTests	b31fd5a
2	Mustache	https://github.com/noha/mustache	728feda
3	PetitParser	https://github.com/moosetechnology/PetitParser	5cf9331
4	Pillar	https://github.com/pillar-markup/pillar	65dbece
5	Seaside3	https://github.com/SeasideSt/Seaside	4ba832d
6	Spec2	https://github.com/pharo-spec/Spec	c091e45
7	PolyMath	https://github.com/PolyMathOrg/PolyMath	644e8f9
8	TelescopeCytoscape	https://github.com/TelescopeSt/TelescopeCytoscape	e614f4d
9	Voyage	https://github.com/pharo-nosql/voyage	cb4362d
10	Bloc	https://github.com/pharo-graphics/Bloc	d6ff9ae
11	DataFrame	https://github.com/PolyMathOrg/DataFrame	2facb1c
12	Roassal2	https://github.com/ObjectProfile/Roassal2	c1da861
13	Roassal3	https://github.com/ObjectProfile/Roassal3	f963667
14	Moose	https://github.com/moosetechnology/Moose	839b7f9
15	GToolkit	https://github.com/feenkcom/gtoolkit	6a937d8
16	Iceberg	https://github.com/pharo-vcs/iceberg	4c7c57d

Table 3: 16 external projects that we have collected into our dataset.

The first 14 external projects were loaded into Pharo 7. GToolkit and Iceberg were loaded into Pharo 8 because their latest latest versions are only compatible with the latest version of Pharo.

4.1 Tokens, Subtokens, and Words: Five Representations of Source Code

Our dataset is essentially the collection of methods. We store the metadata of each method, such as package and class name, project to which it belongs, protocol, selector (method name), and the number of lines of code.

As for the source code of each method, we tokenize and clean it, produce 5 different representations, and store them in separate tables:

- *"sources.csv"* – raw source code of a method stored as a string. We replaced all whitespace characters with a single space which simplified the code and allowed us to use tabulation and newline characters as separators in a CSV file.
- *"tokens.csv"* – in this table, source code of each table is represented as a sequence of tokens, separated by spaces. In Section 4.1.1 we discuss the process of tokenization in more details.
- *"subtokens.csv"* – we split each token into separate words and symbols (we call them **subtokens**) by camel case and converted them to lowercase. In this table, the source code of each method is represented as a sequence of subtokens, separated by spaces. We discuss the detailed process of extracting and filtering subtokens in Section 4.1.2.
- *"words.csv"* – in this table, the source code of each method is represented as a sequence of alphabetic subtokens (we call them **words**). Subtokens are taken from table *subtokens.csv* which means that words from comments and strings are not included. In other word, each method in this table is represented as a sequence of words from identifier names.
- *"all_words.csv"* – in this table, we represent source code of each method as a sequence of all words, including the words from comments, strings, symbols, and literal arrays.

Additionally, we save the following metadata for each method in table *"methods.csv"*: unique identifier that is used as a foreign key by other tables, project name, package, class, protocol, selector (name) of a method, number of lines of code, and a boolean value specifying whether the class of the method is a subclass of TestCase. We stored the number of lines of code separately because this information is lost from *"sources.csv"* table when we replace all whitespace characters with spaces.

Note that tokens table stores the type of each token which allows us to know which token is an instance variable, literal, selector, class name, etc. This will be used in Section 5.4 to study the distribution of tokens across their types.

4.1.1 Extracting Tokens

While it is not too hard to split English sentences into words (we can split them on spaces and punctuation), tokenizing source code is not a trivial task. Whitespace characters are not mandatory around certain kinds of tokens (such as delimiters or operators), numbers have many different representations, and everything enclosed into single or double quotes is a string literal or comment and should be considered as one token. We used the abstract syntax tree (AST) of each method provided by Pharo environment and collected its leaves as tokens of source code into our dataset. We also used the type of each leaf node to specify the token types.

4.1.2 Extracting and Filtering Subtokens

Additionaly, we split each token using camel case into the so-called "subtokens" - words and symbols that are used to construct identifier names using the camelCaseNotation. This way, the single token

collect:thenSelect:, corresponding to the message send, was split into 5 subtokens: collect, :, then, select, :. Notice that we have separated colon as a separate subtoken.

We have also replaced all numbers with a single `<num>` token and all strings with a single `<str>` token. This reduced the number of unique tokens by about a half.

5 Answering Questions

Now that we have collected the data, we can use it to answer questions defined in Section 3.

5.1 How big are the projects?

First of all, we want to understand how big are the projects in our dataset and how they are structured. To that end, we report structural object-oriented measures such as the number of classes or number of methods. In Table 4, you can see the full list of projects that we use in this study, with number of packages, classes, methods, lines of code, and tokens of source code reported for each project. Notice that we do not count metaclasses separately. This means that Object and Object class count as one class. As far as we are concerned, both instance and class-side methods belong to class Object.

5.2 How many tests do we have?

It is interesting to know how well are the projects tested. Getting the code coverage is a bit complicated, but we can easily count the number of test methods. In Table 5, we report the number of test packages, test classes, and test methods for each project. We also report the proportion of test methods to the total number of methods in the project. You can see that DataFrame has the highest proportion of 55.8%, which means that most of its methods are tests. This is not a perfect metric, but it can give us a hint that the remaining 44.2% of methods are well covered with tests.

It is not always clear what do we consider as a test method, class, or especially package. Here is the heuristic that we used:

Test package – a package whose name ends with -Test or -Tests or contains a -Tests- substring.

Test class – subclass of TestCase.

Test method – a method inside test class whose name starts with test.

5.3 How much is too many?

There is no such thing as a single threshold that would tell you when your class has too many methods or when a method has too many lines of code. Still, most of us would agree that a method with 100 lines is probably not a good one. Appropriate length for a method can not be defined as a number of lines of code or tokens: a good method should do one thing, and only one thing [Martin, 2009]. However, in practice it is very hard to automatically detect methods that are doing more than one thing and very easy to count the lines of code. It is useful to detect source code entities (methods, classes, etc.) that are much larger than most of the others, and ask developers to inspect those entities manually.

Project	Packages	Classes	Methods	Lines of Code	Tokens
GToolkit	87	1,897	16,644	113,399	331,586
Moose	98	1,208	10,987	66,750	218,788
Bloc	21	1,027	10,227	58,702	209,382
Roassal2	19	838	8,831	90,630	256,473
Morphic	18	367	8,358	48,061	192,230
Seaside3	51	860	7,611	78,448	134,266
Spec2	21	655	5,872	23,895	80,572
Kernel	9	254	5,750	36,173	144,539
Iceberg	17	664	5,749	25,030	87,043
Calypso	51	749	5,264	19,548	70,741
Pillar	40	408	4,555	22,379	75,623
Collections	16	231	4,442	27,880	117,588
System	40	286	3,510	19,987	63,999
PolyMath	59	311	3,326	24,290	109,754
PetitParser	36	202	3,309	22,690	84,186
Metacello	26	211	3,258	27,466	87,493
Tool	17	190	3,142	17,074	62,099
Monticello	13	213	2,871	15,940	60,372
Roassal3	14	245	2,685	18,287	78,576
Zinc	10	184	2,607	15,147	59,807
Refactoring	9	221	2,503	15,832	55,422
OpalCompiler	3	170	2,364	13,773	54,592
Rubric	4	102	2,334	12,777	50,195
TelescopeCytoscape	12	245	2,145	9,721	31,355
OSWindow	5	221	2,098	8,223	23,721
Polymorph	2	87	2,070	11,466	37,899
Graphics	8	75	1,889	16,723	68,927
Athens	7	182	1,574	10,367	28,111
AST	3	93	1,505	8,267	33,992
UnifiedFFI	3	148	1,292	6,033	19,244
Reflectivity	5	120	1,272	7,762	33,493
Text	6	71	1,221	7,661	31,379
Epicea	4	127	1,213	5,469	18,201
Fuel	6	172	1,029	5,192	19,294
Refactoring2	3	128	990	8,095	26,371
Network	6	56	912	6,151	22,155
DataFrame	6	27	912	7,309	23,829
SUnit	8	71	842	4,455	16,281
Voyage	10	133	812	3,289	11,022
Keymapping	6	65	542	2,026	7,499
Renraku	3	109	538	2,515	7,788
Regex	3	33	510	3,297	8,528
STON	3	54	437	2,435	10,907
Commander	9	45	342	1,171	3,724
DrTests	10	47	336	1,329	4,211
Debugger	4	36	315	1,459	4,616
Ombu	2	36	289	1,368	4,503
Shift	5	35	232	1,029	4,288
Mustache	3	23	151	661	2,645
Settings	3	3	50	971	1,860
Total	824	13,935	151,717	958,602	3,191,169

Table 4: Structural object-oriented measures of projects in our dataset. Projects are sorted by the number of methods.

Project	**Test Packages**	**Test Classes**	**All Methods**	**Test Methods**	%
DataFrame	3	11	912	509	55.8%
PetitParser	15	52	3,309	1,410	42.6%
Regex	1	3	510	193	37.8%
STON	1	10	437	125	28.6%
Pillar	18	147	4,555	1,282	28.1%
OpalCompiler	1	36	2,363	660	27.9%
Mustache	1	1	151	39	25.8%
PolyMath	23	96	3,326	804	24.2%
Kernel	4	86	5,748	1,354	23.6%
Reflectivity	2	14	1,272	283	22.2%
Refactoring2	1	43	990	208	21.0%
Fuel	1	29	1,029	193	18.8%
Collections	3	61	4,442	804	18.1%
Zinc	3	43	2,607	414	15.9%
DrTests	7	10	336	52	15.5%
Moose	28	236	10,987	1,664	15.1%
Calypso	17	151	5,264	704	13.4%
Ombu	1	9	288	38	13.2%
Epicea	2	17	1,213	150	12.4%
Seaside3	17	139	7,610	928	12.2%
Roassal2	3	176	8,831	1,063	12.0%
SUnit	2	18	842	101	12.0%
AST	1	17	1,505	180	12.0%
Refactoring	4	51	2,503	299	11.9%
Shift	2	5	232	27	11.6%
Network	1	16	912	93	10.2%
Graphics	1	12	1,889	176	9.3%
Monticello	2	45	2,871	256	8.9%
Spec2	5	137	5,872	522	8.9%
Iceberg	4	95	5,749	494	8.6%
Voyage	4	24	812	67	8.3%
UnifiedFFI	1	19	1,290	103	8.0%
Keymapping	1	10	542	43	7.9%
System	10	47	3,510	265	7.5%
Renraku	1	19	538	39	7.2%
TelescopeCytoscape	4	31	2,145	135	6.3%
Text	2	10	1,221	64	5.2%
Metacello	2	34	3,258	165	5.1%
Roassal3	1	27	2,685	130	4.8%
Debugger	1	3	315	14	4.4%
Commander	1	3	342	10	2.9%
Tool	3	12	3,142	91	2.9%
Rubric	1	2	2,334	58	2.5%
Bloc	3	31	10,221	151	1.5%
Morphic	1	16	8,358	52	0.6%
Athens	1	2	1,574	5	0.3%
OSWindow	1	3	2,098	4	0.2%
GToolkit	2	8	16,640	27	0.2%
Polymorph	0	0	2,070	0	0.0%
Settings	0	0	50	0	0.0%

Table 5: This table shows the number of test packages, classes, and methods found in each of the 50 projects in our dataset, as well as the total number of methods and the proportion of test methods to the total number of methods in each project. Projects are sorted by the proportion of test methods

5.3.1 Number of classes per package

First we want to understand how many classes are there inside packages. In Table 6 you will find some statistics describing the distribution of the number of classes per package. Those are the min, max, mean, median, and the other two quartiles (25% and 75%).

In the first row, we report the numbers for all packages. In second and third rows however, we present test and non-test packages separately. In fact, the concept of a test package does not exist in Pharo programming language. However, it is a highly encouraged convention that is widely used in the community: all tests are stored in the separate package whose name ends with suffix -Tests. To cover most variations of package names, we mark a package as test package if:

1. Package name ends with -Test or -Tests
2. Package name contains the substring -Tests-

	count	min	25%	median	mean	75%	max
All packages	824	1	2	7	18.4	17	469
Test packages	214	1	1	4	11.8	12	162
Non-test packages	610	1	3	8	20.7	19	469

Table 6: Statistics describing the distribution of the number of classes per package. First row reports the numbers for all packages, second and third rows describe test and non-test packages separately

You can see that all three distributions (rows) are right-skewed: they are shifted to the right, meaning that most packages have very few classes in them, but there are some exceptionally large packages that are packed with hundreds of classes. Those large packages are the minority, but they shift the mean value. We will see the same form of distribution in the following sections where we look at the number of methods per class as well as the number of lines of code and the number of tokens per method.

It is important to note that right-shifted distributions are not normal. Intuitively, we think of mean (average) as a "typical" value. But in right-shifted distributions, mean, median, and mode are not the same. Take a look at the numbers in Table 6: most packages have less classes than mean. Besides, mean value of such distributions is very unstable. You can change it dramatically by adding or removing a very large package (class, method, etc.) to the dataset. Median on the other hand is a much more stable measure of central tendency. Therefore, we use the median as a better indicator than mean. We will use it systematically for the number of classes per package or number of lines of code per method.

You might be wondering what are those large packages that are home to hundreds of classes. In Table 7, we present the list of the top 10 largest packages in our dataset (both test and non-test).

	Project	Package	# classes
1	Bloc	Bloc	469
2	Roassal2	Roassal2	403
3	Iceberg	Iceberg-TipUI	287
4	Moose	Famix-Traits	238
5	Seaside3	Seaside-Core	174
6	GToolkit	GToolkit-Coder	170
7	Roassal2	Roassal2-Tests	162
8	GToolkit	GToolkit-Inspector	157
9	Iceberg	Iceberg	150
10	Spec2	Spec2-Core	148

Table 7: Top 10 packages by the number of classes

5.3.2 Number of classes inside test and non-test packages

By their nature, test classes are very different from other classes. In this section, we once again take a look at the number of classes inside packages, but this time we study test packages and non-test packages separately.

5.3.3 Number of methods per class

When it comes to exploring the distribution of the number of methods per class, there are two major considerations that have a strong effect on the statistics:

1. In Pharo (and in our dataset), every class is instance of another class, called a metaclass. This class can be considered in a first approximation as the static part of the Pharo class [Black et al., 2009]. For software analyses, tools often merge the class and metaclass while distinguishing the scope of the merged properties. In this project, we are merging a class with its metaclass. When counting the number of methods per class, we consider Object and Object class to be the same class. This means that when we count the number of methods of class Object, we include the methods of Object class.

2. Many packages in our dataset contain extension methods for classes outside those packages [Bergel et al., 2005, Polito et al., 2017]. For example, Moose adds several extension methods to class String, but we do not want to count String as the class of Moose (in fact, 27 out of 50 projects in our dataset extend String). Therefore, **we do not count extension methods** (4.5% of all methods). We detect all extension methods in our dataset as the ones whose protocol start with an asterisk symbol * and remove them from our study (but only in this section).

We have counted some statistics that describe the distribution of the number of methods per class as shown in Table 8.

	count	min	25%	median	mean	75%	max
All classes	12,520	1	3	6	11.3	12	1,341
Test classes	2,056	1	2	5	10.2	10	230
Non-test classes	10,761	1	3	6	11.2	12	1,341

Table 8: Statistics describing the distribution of the number of methods per class. We present the numbers for all classes, as well as the separate numbers for test and non-test classes. Column count shows the number of classes in each group.

In Table 9 you can see the list of the top 10 largest classes in our dataset.

	Project	Class	# methods
1	GToolkit	GtGleamGL	1341
2	Morphic	Morph	720
3	Polymorph	UITheme	634
4	Bloc	BlElement	291
5	Spec2	SpRubTextFieldMorph	241
6	Rubric	RubAbstractTextArea	240
7	Rubric	RubTextEditor	233
8	DataFrame	DataFrameTest	230
9	Seaside3	WAHtmlCanvas	226
10	Roassal2	RTRoassalExample	223

Table 9: Top 10 classes by the number of methods. This includes both test and non-test classes.

5.3.4 Number of lines of code per method

The number of lines of code (LoC) is a very popular metric in software engineering. In this section, we will study the typical number of LoC in Pharo methods [Chidamber and Kemerer, 1994, Balmas et al., 2009].

There are many ways to count the number of lines of code. In this report, we use the implementation provided by the CompiledMethod.linesOfCode. It is based on the following key principles:

1. First line of a method which contains its name and arguments is counted as a line of code.
2. Comments are included and all lines of a multiline comment are counted as lines of code.
3. Empty lines are not counted. However, if a line contains whitespace characters, it is not considered empty.
4. Each line of a statement (terminated by dot) that is split into several lines is counted separately.

Take a look at the following implementation of the LinkedList.remove:ifAbsent: method. This method has 11 lines of code: those are lines number 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, and 14. Lines number 4, 9, and 13 are not counted.

```
1  LinkedList >> remove: aLinkOrObject ifAbsent: aBlock
2    "Remove aLink from the receiver.
3    If it is not there, answer the result of evaluating aBlock."
4
```

```
5     | link |
6     link := self
7       linkOf: aLinkOrObject
8       ifAbsent: [↑aBlock value].
9
10    self
11      removeLink: link
12      ifAbsent: [↑aBlock value].
13
14    ↑aLinkOrObject
```

If you look at Table 10, you will see how the number of lines of code per method is distributed in our dataset. The length of a method stretches from 1 to as long as 15,609 lines. The mean is 6.3, but it is strongly affected by the extremely large methods. Median, which is a more stable central tendency metric, tells us that 50% of methods in our dataset have no more than 3 lines of code. As you will see in the rest of this section, those numbers are impacted by data methods.

	count	min	25%	median	mean	75%	max
All methods	151,717	1	2	3	6.3	6	15,609
Test methods	16,448	2	4	7	9.3	12	320
Non-test methods	135,269	1	2	3	6	6	15,609

Table 10: Statistics describing the distribution of the number of lines of code per method. The three rows of this table present separate statistics for all methods, only test methods, and only non-test methods. The count column shows how many methods are in each group.

15,609 lines of code in a single method is a lot. Table 11 lists the 10 largest methods in our dataset in terms of number of lines of code.

	Project	Class	Method	# lines
1	Seaside3	JQUiDeploymentLibrary	jQueryUiJs	15609
2	Seaside3	JQUiDevelopmentLibrary	jQueryUiJs	15609
3	GToolkit	GtCSExamplesData	unsplashJsonTextPictures3500Cropped	10221
4	GToolkit	GtCSExamplesData	unsplashJsonTextPicturesCropped	10044
5	Seaside3	JQDevelopmentLibrary	jQueryJs	8330
6	Moose	VerveineJTestResource	mse	5007
7	Roassal2	RTOSM	buildingsAndStreamKampungMelayu	2880
8	Roassal2	RTHTML5Exporter	roassalJSContent	1816
9	Athens	VGTigerDemo	tigerPoints	1701
10	PetitParser	PPXmlResource	xsdXsd	1449

Table 11: Top 10 longest methods by the number of lines of code. This table includes both tests and not tests. Just by looking at the method names in this table, you can tell that many of them are data methods.

Many of the largest methods are what we call *"data methods"*. They do nothing, but simply return long sequences of numbers, arrays, or strings. Take the longest method in our dataset for example. JQUiDeploymentLibrary.jQueryUiJs has 15,609 lines of code, all of which are a single string - the JavaScript source code of the jQuery UI - v1.12.1 library.

Data methods like that one are the noise in our dataset. They are very different from other methods (one might argue that data methods should not be considered methods at all) and usually much larger. They can not be considered representative of Pharo's source code and they significantly shift the statistics. So in the next step of our analysis, we detect and remove data methods.

Detecting data methods is not an easy task, because there is no clear definition of what they are and no easy way to draw a line between a normal method and a data method. We managed to detect most of the largest data methods using the following heuristic: *we consider methods that contain only literals, delimiters, comments, and a return operator to be data methods.* We have detected 6,976 such methods in our dataset (4.6% of all methods). Table 12 presents the distribution of the number of lines of code without data methods.

	count	**min**	**25%**	**median**	**mean**	**75%**	**max**
All methods	144,741	1	2	3	5.7	7	2,880
Test methods	16,448	2	4	7	9.3	12	320
Non-test methods	128,293	1	2	3	5.3	6	2,880

Table 12: Statistics describing the distribution of the number of lines of code per method after data methods were removed from analysis. The three rows of this table present separate statistics for all methods, only test methods, and only non-test methods. The count column shows how many methods are in each group.

Median is a stable metric of central tendency - those values were not affected by removing 4.6% of extreme values. Mean, on the other hand, has dropped from 6.3 to 5.7 for all methods and from 6 to 5.3 for non-test methods. Notice that test methods were not affected at all. That is because not a single test method from our dataset was marked as the data method by our heuristic. As one might expect, the max values changed a lot. Take a look at the top 10 longest methods in our dataset after we have removed the data methods. In Table 13, 9 out of 10 largest methods from Table 11 were filtered out as data methods.

	Project	**Class**	**Method**	**# lines**
1	Roassal2	RTOSM	buildingsAndStreamKampungMelayu	2880
2	Roassal2	RTAnimatedScatterPlotExample	exampleEvolutionOfGraphET2	396
3	PolyMath	PMStatisticsBugs	testProbabilityDensity	320
4	Metacello	MetacelloConfigurationResource	setUpConfigurationOfProjectToolBox	285
5	Moose	FamixGenerator	defineRelations	267
6	System	KeyboardKey	initializeKeyTable	231
7	Moose	FamixGenerator	defineTraits	227
8	Roassal2	RTMetricMap	example05	210
9	Roassal2	RTAnimatedScatterPlotExample	exampleCountriesAndPublications	205
10	Regex	RegexHelp	usage	205

Table 13: Top 10 longest methods by the number of lines of code. Data methods (detected by the heuristic described in this section) are excluded from study.

Some of the methods that are left (including RTOSM.buildingsAndStreamKampungMelayu which is the longest method in this table) are still data methods. But they are not easy to filter out because they construct objects before returning them and look very much like normal methods -

they have message sends, assignments, etc. The next largest method, RTAnimatedScatterPlotExample.exampleEvolutionOfGraphET2 has 396 lines of code, which is a lot, but much less than 15,609 lines of the top 2 methods from Table 11.

We also want to explore the internal projects separately from the external ones. Source code integrated into Pharo passes certain quality checks and adheres to the same conventions. External projects on the other hand are more independent, they exist in different domains, in some of which long methods are more acceptable than in others. Indeed, out of 10 longest methods presented in Table 13, only 3 methods are from internal projects: methods number 4, 6, and 10.

In Table 14 we describe the distribution of the number of lines of code for the methods that were integrated into the Pharo environment. You can see that all three quartiles: 25%, 50% (median), and 75% remain the same. However, the mean values are slightly decreased, and max values decrease a lot.

	count	min	25%	median	mean	75%	max
All methods	65,474	1	2	3	5.8	7	285
Test methods	7,171	2	4	7	9.1	12	179
Non-test methods	58,303	1	2	3	5.4	6	285

Table 14: Statistics describing the distribution of the number of lines of code per method. In this table, we count only methods from internal projects (the ones that are integrated into Pharo). We also do not count data methods. We report separate numbers for all methods, only test methods, and only non-test methods. The count column shows the total number of methods in each group.

Table 15 presents the top 10 largest non-data methods in the internal projects from our dataset.

	Project	Class	Method	# lines
1	Metacello	MetacelloConfigurationResource	setUpConfigurationOfProjectToolBox	285
2	System	KeyboardKey	initializeKeyTable	231
3	Regex	RegexHelp	usage	205
4	Regex	RegexHelp	syntax	196
5	Kernel	IntegerTest	testPositiveIntegerPrinting	179
6	Kernel	IntegerTest	testNegativeIntegerPrinting	176
7	Graphics	JPEGReadWriter	idctBlockInt:qt:	151
8	Metacello	MetacelloReferenceConfig	baseline10:	143
9	Metacello	MetacelloMCVersionValidator	validateBaselineVersionSpec:	141
10	Metacello	MetacelloProjectSpec	configMethodBodyOn:indent:fromShortCut:	140

Table 15: Top 10 longest methods by the number of lines of code. Data methods and methods from external projects are excluded from study.

5.3.5 Number of tokens per method

As you could see in the previous section, the large number of lines of code does not mean that the method is complex. The method can simply return a very long string. On the other hand, line breaks in source code are optional, which means that the same method can be split into many lines or written in a single line. In this section we take a look at the number of tokens in the source code of methods. Table 16 will give you an idea on how many tokens do methods in Pharo typically have.

	count	min	25%	median	mean	75%	max
All methods	151,717	2	4	8	21	24	1,639
Test methods	16,448	3	16	32	41.9	55	1,173
Non-test methods	135,269	2	4	7	18.5	19	1,639

Table 16: Statistics describing the distribution of the number of tokens per method. The three rows of this table present separate statistics for all methods, only test methods, and only non-test methods. And in the count column you can see how many methods are there in each group.

Notice that the maximum number of tokens is lower than the maximum number of lines of code from any of the Tables 10, 12, and 14. (even though one line can have many tokens). This is due to the fact that the longest methods contain very long literals, some of which stretch over multiple lines, even though their number of tokens is relatively low. For example, a method that returns a string containing the code of a JQuery library has many lines of code but only two tokens: the return operator and the string.

Let's take a look at the largest methods in terms of the number of tokens. You can see them in Table 17. We already know that method with many lines of code do not always have many tokens. By comparing the last two columns of this table, we can also see that the methods with many tokens do not necessarily have many lines.

	Project	Class	Package	# tokens	# lines
1	Roassal2	RTExperimentalExample	sankeyData	1639	123
2	Roassal2	RTCPSequential	rawPalette	1212	3
3	Moose	FamixGenerator	defineTraits	1208	227
4	Kernel	IntegerTest	testPositiveIntegerPrinting	1173	179
5	Kernel	IntegerTest	testNegativeIntegerPrinting	1155	176
6	Graphics	JPEGReadWriter	idctBlockInt:qt:	944	151
7	PolyMath	PMStatisticsBugs	testProbabilityDensity	936	320
8	System	KeyboardKey	initializeUnixVirtualKeyTable	821	109
9	Roassal3	TSAthensRenderer	buildRingPath:	788	135
10	Athens	AthensCubicBezier	recursiveSubDiv:level:	758	121

Table 17: Top 10 longest methods by the number of tokens.

5.4 What is the code made of?

In this section, we study the nature of tokens that are the building blocks of source code. We want to understand how many of those tokens are literals, identifier names, or delimiters, how many of them are reserved keywords, how many literals are numbers, strings, characters, etc. Those numbers describe the way programmers use the language.

5.4.1 Distribution across different types of tokens

First of all, we want to understand what is the proportion of different types of tokens such as literals, identifiers, delimiters, etc. In this section, we split tokens of Pharo programming language into several groups, provide definition for each group and then count the number of tokens from each group that appear in our dataset.

Keywords – we define keywords in Pharo as a combination of **reserved identifiers**: `nil`, `true`, `false`, `self`, `super`, `thisContext` and **restricted selectors**: `ifTrue:`, `ifFalse:`, `to:do:`, `==`, `basicAt:`, `basicSize`, `ifTrue:ifFalse:`, `ifFalse:ifTrue:`, `to:by:do:`, `basicAt:put:`, `basicNew:` [Goldberg and Robson, 1983].

Identifiers – we define identifiers as a combination of **class names**, **method names**, **global variable names**, **instance variable names**, **class variable names**, **temporary variable names**, and **argument names**. We did not count reserved identifiers (`nil`, `true`, `false`, `self`, `super`, `thisContext`) and restricted selectors as identifier names.

Literals – there are five types of literals in Pharo: **numbers**, **strings**, **characters**, **symbols**, and **literal arrays**.

Delimiters – those are parentheses `()`, square brackets `[]`, curly braces {}, pipes `|` and a dot character `.` which terminates statements. We counted opening and closing parentheses, brackets, and braces as two separate tokens. Also, we made no difference between the dot at the end of each statement and dots between the elements of arrays, because every element in a non-literal array is also a statement. As for the pipes, we counted both those that surround the declaration of temporary variables, and the ones that are part of blocks. We did not count single quotes `''` around strings and parentheses of literal arrays `#(()())`. Because even though these are delimiters, they are part of the literals. For the same reason, we did not count double quotes `""` surrounding the comments.

Keywords are also identifiers. In Pharo, it is especially hard to draw a line between "keywords" and other identifiers. This is due partly to the fact that true and false are objects of classes True and False, + and - are methods implemented for numerical objects, and control statements which are keywords in most other languages, such as if or for, are implemented as methods in Pharo as ifTrue:, ifFalse:, or do:. Nevertheless, we put keywords into a separate group because those identifiers are much more common than the other ones and are very unlikely to be changed by the programmer. So it is useful to know how many tokens in the source code are reserved keywords such as true, false, self, or nil, and how many are identifier names created by developers.

In Table 18 you can see the distribution of tokens across the 4 groups defined above.

Type	#	%	characters	%
Keywords	281,717	9.0	1,290,149	4.5
Identifiers	1,630,165	51.9	13,766,097	48.0
Literals	223,554	7.1	12,601,322	44.0
Delimiters	1,005,472	32.0	1,005,472	3.5
Total	3,140,908	100.0	28,663,040	100.0

Table 18: Distribution of Pharo source tokens across their kinds on token and character levels.

5.4.2 Comparing those numbers to Java

Similar analysis has been performed by Deissenboeck et al. [Deißenböck and Pizska, 2005]. They have studied the Eclipse Java code base and counted source level tokens differentiating them according to their kind. The result of this analysis can be seen in Table 19.

Type	#	%	characters	%
Keywords	1,321,005	11.2	6,367,677	12.7
Delimiters	5,477,822	46.6	5,477,822	11.0
Operators	701,607	6.0	889,370	1.8
Identifiers	3,886,684	33.0	35,723,272	71.5
Literals	378,057	3.2	1,520,366	3.0
Total	11,765,175	100.0	49,978,507	100.0

Table 19: Distribution of Java source tokens across their kinds on token and character levels (according to [Deißenböck and Pizska, 2005]).

Although it is very interesting to compare those distributions, we must warn readers to be extra careful when combining the numbers from different studies. We made many decisions that affected the way how we preprocessed the data, tokenized source code, or defined the groups of tokens. Even slightest changes to this process (for example, removing comments, separating colon from the method name, counting opening and closing parentheses as one or two tokens) can have large effect on the statistics. Therefore, when you compare the numbers from Table 18 and Table 19, remember that they come from different studies and there are large margins of error.

5.4.3 Distribution of Keywords

Now we take a closer look at the distribution of specific types of keywords. We start with the reserved identifiers. As you can see in Table 20, most commonly used keyword is self. It takes 6.8% of source code on the level of tokens and 3% of code on character level. Following it are super, and false that have relatively similar counts and take up only 0.3% of source code on token level. 0.2% of tokens are nil and true respectively. Finally, thisContext is a very rare keyword, which was used only 339 times in the methods from our dataset. This is normal because it reifies the execution stack and it is known by developers to be costly and it is also rare to have to manipulate execution stack.

Type	#	%	characters	%
self	214,162	6.8	856,648	3.0
super	8,113	0.3	40,565	0.1
false	8,047	0.3	40,235	0.1
nil	7,634	0.2	22,902	0.1
true	7,487	0.2	29,948	0.1
thisContext	339	0.0	3,729	0.0
Total	245,782	7.8	994,027	3.5

Table 20: Distribution of reserved identifiers in Pharo source tokens across their kinds on token and character levels.

Now we look at the distribution of restricted selectors. Table 21 shows that the most common one of those selectors is ifTrue: - it appears in our methods 15,671 times and takes 0.5% of all tokens. It is interesting that ifFalse: appears only 5,398 times, which makes it approximately 3 times less frequent. Similarly, ifTrue:ifFalse: was used 7,531 and ifFalse:ifTrue: - only 295 times, which makes the positive statement 26 times more common. It is worth to understand that the most frequent restricted message selectors (ifTrue: and related) would not be counted as such in other languages because in other languages such operations are not implemented as methods but as language constructs hardcoded in the language syntax.

Type	#	%	characters	%
ifTrue:	15,671	0.5	109,697	0.4
ifTrue:ifFalse:	7,531	0.2	112,965	0.4
ifFalse:	5,398	0.2	43,184	0.2
==	4,623	0.1	9,246	0.0
to:do:	1,769	0.1	10,614	0.0
ifFalse:ifTrue:	295	0.0	4,425	0.0
to:by:do:	182	0.0	1,638	0.0
basicSize	148	0.0	1,332	0.0
basicAt:	144	0.0	1,152	0.0
basicAt:put:	101	0.0	1,212	0.0
basicNew:	73	0.0	657	0.0
Total	35,935	1.1	296,122	1.0

Table 21: Distribution of restricted selectors in Pharo source tokens across their kinds on token and character levels.

5.4.4 Distribution of Identifier Names

Now that we have explored the distribution of keywords, which are the special type of identifiers, we can look at those identifier names that are usually chosen by developers. In Table 22 you can see how those names are distributed among different source entities to which they belong. Remember that those numbers tell us how many times method names or class names were used in the source code of methods that we have collected into our dataset. These are not the total number of methods or classes in the system.

Type	#	%	characters	%
Method names	858,126	27.3	7,538,361	26.3
Temporary variable names	301,505	9.6	2,043,432	7.1
Argument names	227,999	7.3	1,573,026	5.5
Instance variable names	123,123	3.9	1,065,204	3.7
Class names	107,215	3.4	1,383,064	4.8
Class variable names	8,737	0.3	132,844	0.5
Global variable names	3,460	0.1	30,166	0.1
Total	1,630,165	51.9	13,766,097	48.0

Table 22: Distribution of identifier names in Pharo source tokens across their kinds on token and character levels.

It is interesting to note that method names take 27.3% of source code on token level and 26.3% of code on character level. Which means that about a quarter of source code that programmers write are method names (message sends).

5.4.5 Distribution of Literals

In this section, we study how literals are distributed among the five types of literals defined in Pharo:

1. Numbers: 42, -0.5, 2.4e7, etc.
2. Strings: 'Hello world!', 'Lorem ipsum dolor sit amet', etc.

3. Symbols: #Hello, #with:collect:, etc.

4. Characters: $a, $b, $c, etc.

5. Literal arrays: #(), #(1 2 3), #(hello true #(1 2)), #((2 4)(3 9)), etc.

Notice that a literal array may contain many numbers, strings, other arrays, or any of the literals mentioned above. Still, it is counted as a single literal (in the same way as string is counted as one literal and not as a collection of character literals).

In Table 25 you can see how literals in our dataset are distributed among those types. 47% of all literals (3.4% of tokens) are numbers, 29% of literals are strings, 16% are symbols, 6% are literal arrays, and the remaining 2% are characters.

Type	**#**	**%**	**characters**	**%**
Numbers	105,283	3.4	195,149	0.7
Strings	64,541	2.1	6,441,659	22.5
Symbols	36,255	1.2	385,321	1.3
Literal arrays	12,133	0.4	5,573,910	19.4
Characters	5,342	0.2	5,283	0.0
Total	223,554	7.1	12,601,322	44.0

Table 23: Distribution of literals in Pharo source tokens across their kinds on token and character levels. The second column shows the total number of literals of each kind, third column tells us what percent of all tokens are those literals, fourth column contains the total number of characters taken by all literals of each kind, and the last fifth column contains the percent of characters in source code that are ocupied by those literals.

5.4.6 Distribution of Delimiters

Finally, we explore the distribution of delimiters. You can see them in Table 24. Notice that 15.2% of tokens in source code are dots which are used to terminate statements. Both opening and closing square brackets take 3.5% of tokens (7% together), opening and closing parentheses take 3.2% each. 3.1% of tokens are pipes. As for the opening and closing curly braces (used to create dynamic arrays), they take only 0.2% of tokens.

Type	**#**	**%**	**characters**	**%**
.	478,480	15.2	478,480	1.7
[	108,708	3.5	108,708	0.4
]	108,708	3.5	108,708	0.4
)	99,832	3.2	99,832	0.3
(	99,831	3.2	99,831	0.3
|	98,565	3.1	98,565	0.3
{	5,674	0.2	5,674	0.0
}	5,674	0.2	5,674	0.0
Total	1,005,472	32.0	1,005,472	3.5

Table 24: Distribution of delimiters in Pharo source tokens across their kinds on token and character levels.

Now remember how at the beginning of this section we said how much the statistics are affected by small decisions that are made during preprocessing. Imagine how all the percentages in the previous sections would be shifted if we removed dots and parentheses from this study.

5.5 What is the "englishness" of the code?

By convention, identifier names are constructed by concatening several English words (sometimes also numbers and special symbols). Intuitively, we understand that developers use less vivid language in their identifier names and comments than the language used in War and Peace or in Shakespeare's plays. We also know that not all words that are used by developers are valid English words. In our code, we tend to invent new words or modify the existing ones, usually by shortening or abbreviating them. In this section, we want to provide numerical evidence for those two statements. We want to answer two questions:

1. What is the total number of unique words used in source code? How does it compare to the number of unique words used in literary English texts?
2. How many of the words used in source code can be recognized as valid English words?

In the top part of Table 25 you can see the total number of words, the number of unique words, and unique stems that appear in source code of Pharo (remember that word is defined as a sequence of alphabetic characters after identifiers were split by camel case). In the bottom part of the table you will find the number of words that were recognised as valid English words[1] along with the number of unique English words and unique English stems that appear in code. We compare those numbers to the texts written in literary English language collected in two large corpora:

The Brown Corpus (The Standard Corpus of Present-Day Edited American English) - the first computer-readable general corpus of texts prepared for linguistic research on modern English. It contains 500 samples of English texts printed in the United States during the year 1961 [Kucera and Francis, 1979] The version provided by NLTK includes all 500 texts.

Gutenberg Dataset - a collection of 3,036 English books written by 142 authors. Those books were manually cleaned to remove metadata, license information, and transcribers' notes [Lahiri, 2014]. We use the subset of the Gutenberg Dataset provided by NLTK, which contains 18 books from 12 different authors.

	Pharo	**Pharo***	**Brown**	**Gutenberg**
Words	2,882,414	5,044,507	981,716	2,135,400
Unique words	8,211	115,620	40,234	41,487
Unique stems	5,693	103,728	25,147	25,562
English words	2,567,081	4,032,600	699,011	1,389,877
Unique English words	5,480	15,155	33,555	29,979
Unique English stems	3,375	8,461	19,116	16,130

Table 25: First two columns of this table describe words that appear in the Pharo source code. Column Pharo describes words from identifier names, and column Pharo* - identifier names, comments, array and string literals. The last two columns describe words that appear in Brown and Gutenberg corpora of English texts.

[1] We used pyenchant (https://pypi.org/project/pyenchant/) provided by NLTK (https://www.nltk.org/) Python library to check if a given word exists in English language or not.

Despite having more words in total, Pharo corpus has much less unique words than both natural English corpora. Both Brown and Gutenberg corpora use around 40,000 unique words and identifier names of Pharo use only 8,211 unique words. This is surprising considering the fact that developers can use any sequence of characters to construct identifier names, for example, array, arr, ar, arrray, ascwcs, all those words, even the ones with typos can be used as part of a variable name. Programmers can use the entire English vocabulary, unlimited vocabulary of new words that don't exist in any language but can be invented by a programmer (take arr, ii, or stemmer for example), and another unlimited vocabulary of lexical mistakes, typos and simply meaningless names. Authors of English books on the other hand are limited to using correct and valid English (even though they also known to come up with Newspeaks and Jabberwockies). The published prose is edited and proofread. So one might expect developers to use much more words (some real, most - not) than English books. And yet, this is not the case. The 151,717 methods in our dataset contain almost 3 million words. This code was written by hundreds of different developers. And all of them managed to use only 8,211 unique words or 5,693 unique stems.

Whats even more interesting is that 5,480 of those words are recognised as valid English words. Notice that software that we used for identifying English words has recognised just 72% of unique words from Brown and Guttenberg corpora (of actual printed English texts). Which means that its knowledge of English is very limited. And yet, this tool recognised around 67% of unique letter sequences from Pharo's source code as valid English words. This brings us two conclusions:

1. The source code that people write in Pharo has good identifier names. It favors plain and understandable English words.

2. Identifier names are created using very simplistic, limited, and highly repetitive vocabulary.

6 Interesting Findings

Here we summarize some interesting facts about the source code that we have discovered during our analysis.

- Three largest packages in our dataset are Bloc, Roassal2, and Iceberg-TipUI. They have 469, 403, and 287 classes respectively. However, these are the extreme cases, most packages are much smaller. In fact, 50% of packages in our dataset have no more than 7 classes.

- Out of 13,935 classes in our dataset, 25% have just 3 methods and 50% of classes have no more than 6 methods.

- Based on the 151,717 methods from the internal projects in our dataset (excluding data methods), the average number of lines of code in Pharo methods is 5.8. However, this number is not representative because only 30% of methods have 6 or more lines. The distribution is right-skewed, so it is better to look at the median, which tells us that 50% of methods in our dataset have no more than 3 lines of code.

- DataFrame has the highest proportion of test methods. Almost 56% of its methods are tests.

- About a quarter of source code is taken by message sends. In source code of methods, message sends (method names) take 27.3% of tokens and 26.3% of characters.

- On character level, 22.5% of source code are string literals and 19.4% are literal arrays. Together literals take 44% of characters in source code, but only 7.1% of tokens.

- Positive statements are much more common than negative ones. ifTrue: is used 3 times more often than ifFalse:. Similarly, ifTrue:ifFalse: is 26 times more common than ifFalse:ifTrue:.
- All source code in our dataset was written by hundreds of different people using only 8,211 unique words or 5,693 unique stems (this includes words such as i, j, ordered, nil, etc.). Compare this to over 40,000 unique words used in roughly the same amount of printed English prose.
- 5,480 of those words were recognised as valid English words.

7 Final Words

This report only scraps the surface of interesting questions that can be answered by analysing the statistical properties of source code. Here are some pointers for potential future work:

- We have collected data into CSV tables. However, for more in-depth analysis that might include information about inheritance hierarchies, tests, examples, and traits, the tabular representation of data could be too restrictive. We recommend trying to store packages, classes, methods, and all other entities as Pharo objects.
- It would be interesting to include information about the superclass of each class and the parents of each superclass. This would allow us to know which classes are related through inheritance hierarchies, which classes are tests, exceptions, or collections.
- Just as the vocabulary used in source code is different from the "natural" human English used in literature, grammatical forms are also different. It would be interesting to explore the distribution of words across singular and plural forms, parts of speech (nouns, verbs, adjectives, etc.), tenses (past, present, future).

8 Tools

Here is the list of open-source tools that we have developed while working on this technical report:

- The repository of this project contains our dataset, as well as the notebooks with the code that we used to analyse this data: https://gitlab.inria.fr/RMODPapers/2019-sourcecodedata.
- SourceCodeDataCollector is the tool that we have built for collecting the code from both internal and external projects: https://github.com/olekscode/SourceCodeDataCollector
- PharoCodeTokenizer is an AST-based tool that can split the source code of any given method into tokens, subtokens, or words: https://github.com/olekscode/PharoCodeTokenizer
- IdentifierNameSplitter is a simple regex-based tool for splitting identifier names by camel case: https://github.com/olekscode/IdentifierNameSplitter

9 Acknowledgements

We want to thank Arolla for financing our research. We are also grateful to Cyril Ferlicot, Julien Delplanque, and Guillaume Larcheveque for helping us select the projects for this study.

References

[Balmas et al., 2009] Balmas, F., Bergel, A., Denier, S., Ducasse, S., Laval, J., Mordal-Manet, K., Abdeen, H., and Bellingard, F. (2009). Software metric for java and c++ practices (squale deliverable 1.1). Technical report, INRIA Lille Nord Europe.

[Bergel, 2016] Bergel, A. (2016). *Agile Visualization*. LULU Press.

[Bergel et al., 2005] Bergel, A., Ducasse, S., and Nierstrasz, O. (2005). Classbox/J: Controlling the scope of change in Java. In *Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA. ACM Press.

[Black et al., 2009] Black, A. P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., and Denker, M. (2009). *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland.

[Bunge, 2009] Bunge, P. (2009). Scripting browsers with Glamour. Master's thesis, University of Bern.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Deißenböck and Pizska, 2005] Deißenböck, F. and Pizska, M. (2005). Concise and consistent naming. In *International Workshop on Program Comprehension (IWPC 2005)*, pages 97–106.

[Demeyer et al., 2002] Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[Gamma et al., 1993] Gamma, E., Helm, R., Vlissides, J., and Johnson, R. E. (1993). Design patterns: Abstraction and reuse of object-oriented design. In Nierstrasz, O., editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany. Springer-Verlag.

[Goldberg, 1984] Goldberg, A. (1984). *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass.

[Goldberg and Robson, 1989] Goldberg, A. and Robson, D. (1989). *Smalltalk-80: The Language*. Addison Wesley.

[Ingalls, 1981] Ingalls, D. H. (1981). Design principles behind Smalltalk. *Byte*, 6(8):286–298.

[Kay, 1972] Kay, A. C. (1972). A personal computer for children of all ages. In *Proceedings of the ACM National Conference*. ACM Press.

[Kay, 1977] Kay, A. C. (1977). Microelectronics and the personal computer. *Scientific American*, 3(237):230–240.

[Kay, 1993] Kay, A. C. (1993). The early history of Smalltalk. In *ACM SIGPLAN Notices*, volume 28, pages 69–95. ACM Press.

[Kucera and Francis, 1979] Kucera, H. and Francis, W. (1979). A standard corpus of present-day edited american english, for use with digital computers (revised and amplified from 1967 version).

[Lahiri, 2014] Lahiri, S. (2014). Complexity of word collocation networks: A preliminary structural analysis. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 96–105, Gothenburg, Sweden. Association for Computational Linguistics.

[Martin, 2009] Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[Polito et al., 2017] Polito, G., Ducasse, S., Fabresse, L., and Teruel, C. (2017). Scoped extension methods in dynamically-typed languages. *The Art, Science, and Engineering of Programming*, 2(1).