What do developers consider magic literals? A smalltalk perspective

Article *in* Information and Software Technology · May 2022

DOI: 10.1016/j.infsof.2022.106942

6 authors, including:

Nicolas Anquetil
University of Lille Nord de France
152 PUBLICATIONS **3,028** CITATIONS
SEE PROFILE

Stéphane Ducasse
National Institute for Research in Computer Science and Control
556 PUBLICATIONS **11,254** CITATIONS
SEE PROFILE

Christopher Fuhrman
École de Technologie Supérieure
42 PUBLICATIONS **204** CITATIONS
SEE PROFILE

Yann-Gaël Guéhéneuc
Concordia University Montreal
327 PUBLICATIONS **8,896** CITATIONS
SEE PROFILE

Contents

What Do Developers Consider Magic literals? A Smalltalk Perspective

N. Anquetil
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
France
nicolas.anquetil@inria.fr

J. Delplanque
Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL
France
julien.delplanque@inria.fr

S. Ducasse
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
France
stephane.ducasse@inria.fr

O. Zaitsev
Arolla
¨ Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
France
oleksandr.zaitsev@inria.fr

C. Fuhrman
christopher.fuhrman@etsmtl.ca
ÈTS Montreal
Canada

Abstract

Literals are constant values (numbers, strings, etc.) used in the source code. *Magic literals* are such values used without an explicit explanation of their meaning. Such undocumented values may hinder source-code comprehension, negatively impacting maintenance. According to R. Martin, avoiding magic literal is *probably one of the oldest rules in software development*. Yet they are still routinely found in source code. We, therefore, studied them to understand when that old rule could be ignored. **Experiments:** First, we perform a *quantitative* study of magic literals, to establish their prevalence. We analyzed seven real projects ranging from small (four classes) to large (7 700 classes). We report the literals' types (number, string, boolean, ...), their grammatical function (*e.g.,* argument in a call, operand in an expression, value assigned, ...), or the purpose of the code in which they appear (test methods, regular code). Second, we perform a *qualitative* study involving 26 programmers and about 24,000 literals to understand which literals they consider magic. Finally, we propose and evaluate *heuristics* for identifying literals that should be considered magic (or not). **Results:** We show that (1) magic literals still exist and are relatively frequent (close to 50% of the methods considered); (2) they are more frequent in test methods (in 80% of test methods); (3) to a large extent, they were considered acceptable (only 25% considered magic); and (4) they can be detected automatically with a good precision (above 80% precision in detecting *NotMagic* literals). We thus pave the way to future research on magic literals for example with tools that could help developers deciding if a literal is acceptable or not. Yann ►*Do we say anything about any possible negative impact of magic literals?*◄ Nic ►*on n'a pas de donnee specifique ladessus*◄ Stef ►*exact l'objectif du papier est comment on peut les identifier.*◄

Conference'17, July 2017, Washington, DC, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords Software analysis, Quality, Literals, Magic literals

1 Introduction

Stef ►*Todo: add literal programming ref and refs about identifiers as in ICPC papers in related works.*◄

Literals are constant values in source code, magic literals are constant values whose meaning might not be obvious to the reader of the code [2, 4]. They have long been considered a type of code smell as they may hinder source-code comprehension, negatively impacting maintenance. When the same literal is used in several places, there is also a risk that an evolution fails to modify all the instances of the literal, thus resulting in a bug.

Yet, despite being "probably one of the oldest rules in software development" [7], literals and magic literals are still easily found in source code. Smit *et al.,* [12] reported that magic literals were common in the systems they studied.

We therefore propose that if there is no solution, then there is no problem, or that if one cannot eradicate literals in the source, thus we must learn to live with them. For example, we envision tools that assist developers avoiding magic literals by (1) warning them when they introduce a magic literal and (2) refactoring magic literals semi-automatically. To do so, we need to better understand what literals are used, where, and what may make them acceptable (*NotMagic*) or not (*Magic*). This is difficult because the same value (ex. "1") may have different meanings in different contexts, and be *Magic* in one and *NotMagic* in another. For example, "2" in the statement "`length := 2 * pi * radius`" is not a magic literal. It is part of the well-known formula for the circumference of a circle. Thus, in this context, it is self explanatory and using a named constant (ex: "`DUE`") could, arguably, be considered detrimental to understanding.

In this paper, we report the results of several experiments to better understand literals:

- First, we study the manifestation of the "problem" with a *quantitative* study of literals. How often do they appear? In what kind of code (test methods or regular code)? What are their types (number, string, boolean, ...)? What are their grammatical function (*e.g.,* argument in a call, value assigned, ...)?
- Second, we study when a literal might be considered *Magic* or *NotMagic*, with a *qualitative* study involving 26 programmers and about 24,000 literals, from more than 3,500 methods in seven real projects, from small (a few classes) to large (several thousands of classes) and covering different domains.
- Third, we propose and evaluate *heuristics* for identifying literals that might be *Magic* or not. One goal is to have tools able to point at literals that should be replaced by named constants.

Results include:

- The confirmation that literals are frequent in source code. Almost 50% of the methods in the seven projects contained at least one literal. And they are more frequent in test methods with 80% of them containing at least one literal;
- Contrary to a common belief, most of these literals (close to 75%) were considered *NotMagic* and would therefore be acceptable in the source code;
- Our proposed heuristics to identify *NotMagic* (acceptable) literals have over 80% precision, meaning that if they accept a literal as *NotMagic*, there is only one chance in five that it should in fact be considered *Magic*.

The rest of the paper is organized as follows: Since the study presented is based on Pharo[1] projects and that Pharo has a syntax originating from Smalltalk, Section 2 gives an overview of this syntax and discusses its impact on the study. Section 3 provides a theoretical discussion on literals, different kinds, when and why they might be considered acceptable or not. Section 4 details the projects used for the *quantitative* study of literals, and gives and discusses the results of this study. Section 3.2 then goes to the *qualitative* analysis of literals, how we evaluated 24 000 of them to categorize them as *Magic* or *NotMagic*, and what the results are. In Section 5, we propose *heuristics* to detect literals that might be acceptable (*NotMagic*) and we evaluate these heuristics on the literals classified in the *qualitative* analysis. Section 6 discusses properties and limitations of the experiment. Section 7 discusses research related to the topic of this paper. Section 8 concludes the paper and discusses perspectives.

[1] pharo.org

```
1 CardGame » distributeCards
2   <menu: 'start'>
3   | cards |
4   cards := (1 to: 52) asOrderedCollection.
5   ^ (1 to: 4) collect: [ :i |  cards remove: i atRandom ]
```

Listing 1. Pharo code example

2 Pharo/Smalltalk Syntax in a Nutshell

We use Pharo systems and libraries to perform our qualitative and quantitative analyses of (magic) literals and show, in the rest of this paper, several examples using Pharo code. Thus, we recall Pharo syntax and use of literals.

2.1 General Syntax

Pharo syntax, following Smalltalk's tradition, is compact and fully fits on a postcard [2]. It is composed of literals (boolean values, numbers, strings, symbols, literal arrays, nil), variable definition, assignment, return, method definition, method annotation, lexical closures, and messages. For example, there are no control flow instructions. Loops and conditionals are expressed with messages (*e.g.,* "ifTrue:"[3] or "to:do:"[4]).

Listing 1, which defines method distributeCards in class CardGame, illustrates some of the syntactic element of Pharo syntax with several examples of literals:

- literals: numbers (1, 4, 52), strings ('start');
- variable definition: Line 3 – temporaries (| |);
- variable access: Line 5 – (cards, i);
- assignment: Line 4 – (:=);
- return: Line 5 – (^);
- method annotation: Line 2 – pragma (<menu:>);
- lexical closure: Line 5 – ([:i |...]);
- messages: Lines 4 and 5 – (to:, asOrderedCollection, collect: atRandom, and remove:).

Messages are an important part of Pharo because they are used to define method invocation, as in other object-oriented programming languages, but also all the control flow (see above). They can be:

- unary: no argument *e.g.,* "asOrderedCollection", "sin", or "atRandom". In Java syntax, they would correspond to "asOrderedCollection()", "sin()", or "atRandom()";
- binary: they are operators (often arithmetic) with one receiver and one argument *e.g.,* "+" in "1 + 2". They have no equivalent in Java since this one uses primitive types for arithmetic rather than messages;

[2] https://en.wikipedia.org/wiki/Pharo

[3] If the receiver (a boolean object) is true, executes the argument (a block, similar to a lambda)

[4] The receiver (a number) creates a sequence of numbers up to the first argument (another number) and the second argument (a block, similar to a lambda) is called for each numbers in the sequence

- keyword: for messages with arguments. The arguments are delimited by ":", *e.g.,* "remove:" would correspond to the message "remove(arg)" in Java syntax. When there are multiple arguments, there are multiple keywords followed by ":", *e.g.,* a method for dictionaries: "at: key put: value" would correspond to the Java syntax: "atPut(key, value)". The name of the Pharo method in this case is "at:put:value".

In Pharo, developers tend to write short methods: Ziatsev *et al.,* [16] reported that method length is only 5.7 lines in average and 3 in median. This could have an impact on our study since shorter methods would be easier to understand and would contain less literals.

2.2 Use of Literals

Pharo's syntax creates many situations in which literals are used. For example, because indexing of collections starts from 1, number 1 in the statement[5] "1 to: n" is typically not considered magic.

Literals are defined as follows: *"Literals describe certain constant objects, such as numbers and character strings. [...] The five types of literal constant are: (1) Numbers, (2) Individual characters, (3) Strings of characters, (4) Symbols, and (5) Arrays of other literal constants"* [5: p18–19]. This definition is similar to definitions used by other programming languages.

In the following, we use this definition of literals and extend it with true, false, and nil, which are not literals *per se* in Pharo but predefined objects contained in global variables.

In Pharo, *symbol*s are a special kind of unique *String* for which there is a single instance for each sequence of characters (symbols are Flyweight [1]). Symbols are often used as keys in associative data structures, such as dictionaries.

It must be noted that constant "variables" (typically public static final in Java) are often represented by a method returning a literal. For example PI could be represented by a method "pi ^3.14159". We will come back on this issue later.

3 Literals in Source Code Qualitatively

In the literature, authors may call *"magic number"* any kind of undocumented literal. Martin [6] highlights that "[t]he term "magic number" does not apply only to numbers. It applies to any token that has a value that is not self-describing." Thus, "magic numbers" could be integers, strings, characters, Boolean values, etc. To avoid confusion, we prefer the term "magic literals" and use "magic numbers" only when referring to magic literals that are numbers.

3.1 Structural Positioning/AST Role of Literals

Yann ►*I don't think that this sub-section should be there, we are not talking about detecting magic literal yet. Move later?*◄

[5]Actually, it is not a statement but a message sent

We study whether the place in the abstract syntax tree (AST) in which a literal is used determines whether it is a magic literal or not. We analyse the places in an AST where literals can occur and identify different AST places:

1. *Receiver.* The literal is the receiver of a message send. In Pharo, operators such as "+" or "<=" are (binary) messages (and the left operand is an argument of this message). For this study, we considered that receivers of such messages were Operands (see next).
2. *Operand:* Receiver or argument of a binary message.
3. *Argument:* Argument in a message send (except binary messages).
4. *Assignment:* Right hand side of an assignment.
5. *Return:* Returned value in a method.
6. *Sequence:* Corresponds to an expression statement. Mainly used as the last statement of blocks (similar to lambdas) for the return value of the block.
7. *Pragma.* Argument to a pragma (similar to annotations in Java).

3.2 Acceptable Literals

Martin [6] defines magic literal as "any token that has a value that is *not self-describing*" (our emphasis). The presence of literals in source code is not a code smell per se. In some contexts, it is legitimate or necessary to insert a literal value at a specific place in the code. We must identify these legitimate usages of literals. We present categories Yann ►*How were these categories obtained?*◄ of acceptable literal values in source code.

Literals Self-describing Their Semantics. Some literals directly refer to the data that they hold. For example, true, false, nil, #() (empty literal array), {} (empty curly braces array) and '' (empty string) are explicit literals. The two first refer directly to the Boolean values true and false; nil refers to the value held by an uninitialized variable, and the last ones refer to empty array and empty string. Those literals are acceptable because their semantics is obvious upon reading, i.e., known by any developer.

Using *string* literals in the source code might also be legitimate. Indeed, if its contents is for people's consumption, *i.e.,*, a sentence or part thereof in a natural language, then it is self-explanatory. Thus, such string literals are acceptable. However, other string literals are magic when their meaning is not clear when read, which is unacceptable.

Julien ►*Need to discuss: while these strings are self-explanatory, they decrease code modularity still*◄ Yann ►*Could also discuss internationalisation/localisation.*◄

By definition, using *symbols*, a special kind of string literals, is also legitimate. They are acceptable because they name/correspond to a unique object. They have a readable form, contrary to numbers, which are also unique, but do not convey semantic information besides their actual values.

```
CriticBrowser class»icon
  "Answer an icon for the receiver."
  ^ self iconNamed: #smallWarningIcon
```

Listing 2. Example of iconNamed: method call.

```
'abcdedfgh' copyFrom: 1 to: 5
```

Listing 3. Example of literals as arguments.

```
EventSensorConstants class »
      initializeEventTypeConstants
  "Types of events"
  EventTypeNone := 0.
  [...]
```

Listing 4. Example of literals assignments.

Listing 2 shows a method using the symbol iconNamed:, which is explicit and, therefore, not a magic literals.

Literals Related to Coding Conventions. Programming languages usually define some conventions that must be respected by developers. An example of such a convention is the indexing of collections. Java uses 0 as first index, Python uses -1 to refer to the last element of a list. Such conventions do not concern only indexing: because in C there is no Boolean type, it considers 0 as false and any other value is true. (In practice, macros are used to avoid using 0 or 1 literals in the source code.)

In Pharo, such conventions are contextual because they depend on the message sent: control flow operations are not part of the language syntax but are messages like any others. For example, copyFrom:to: Yann ►*I don't understand how this example is different from the next one*◄ sent to an instance of SequenceableCollection copies the collection from the first argument index to the last. However, when a collection must be copied from beginning to a given index, the literal 1 must be given as first argument, e.g., aCollection copyFrom: 1 to: n will copy the n first elements of aCollection as shown in Listing 3.

Such literals are acceptable because the programming languages explicitly expect developers to use them. Furthermore, these conventions are described in the languages specifications, and it is very unlikely that they change.

Literals Assigned to a Variable/Returned by a Method. Some literals appear assigned to a variable, *e.g.,* in Listing 4, or returned by a method, *e.g.,* in Listing 5.

In such cases, variable/method names must provide the semantics of the stored/returned literal. Both cases are acceptable but under different conditions. Variables can have their value changed at run-time, which make them ill-suited to store constant values. Methods returning a literal allow storing constant values and accessing them explicitly. Class

```
JPEGReadWriter class » typicalFileExtensions
  "Answer a collection of file extensions (lowercase)
      which files that I can read might commonly have"
  ^#('jpg' 'jpeg')
```

Listing 5. Example of method returning a literal directly.

```
FileDialogWindow class » example
  <example>
  (self onFileSystem: FileSystem disk) open
```

Listing 6. Example of <example> pragma usage.

```
Form » gtInspectorFormIn: composite
  <gtInspectorPresentationOrder: 90>
  ^ composite morph
    title: 'Morph';
    display: [ self asMorph ]
```

Listing 7. Example of <gtInspectorFormIn:> pragma usage with an argument.

methods, *i.e.,* static methods in Java, allow sharing such literal values easily across multiple classes. It is acceptable for a literal to be assigned to a variable or returned directly by a method because its semantics is provided by the variable/method names.

Literals in a Method Annotations In Pharo, methods can be annotated using *pragmas* [3]. A pragma is a static annotation that acts as a tag and supports a declarative registration mechanism to easily retrieve and execute all methods having a specific tag.

Stef ►*we could have only one example of pragma or at least replace the first one with one of FFI because at least this is smart.*◄

Listing 6 shows a method annotated as being an "example". This tag is used by the integrated development environment to let user find and run examples.

Pragmas can be parameterized with arguments, which can only be literals (in the sense Yann ►*Why different from our definition in this paper?*◄ of the definition given by Goldberg [5: p18–19]) because they are created during parsing and are static by nature. For example, Listing 7 shows the pragma gtInspectorPresentationOrder:, which supports inspector extension. It adds an additional information to the method (a number that orders the inspector view). Thus, any literal that is an argument to a pragma is legitimate because, by nature, it must be a literal.

Literals in Test and Example Methods. Literals may appear in test and example code [14, 15]. Following the *Literal Value* pattern [8], literals can sometimes be code smells in tests. If they are repeated, then they should be replaced with symbolic constant to reduce *test-code duplication.* If they

```
1 Behavior » instSpec
2  "Answer the instance specification part of the format
3  that defines what kind of object an instance of the
4  receiver is.  The formats are
5  0 = 0 sized objects (UndefinedObject True False \etal{})
6  1 = non−indexable objects with inst vars (Point \etal{})
7  [...]
8  24−31 = compiled methods (CompiledMethod)"
9
10  ^(self format bitShift: −16) bitAnd: 16r1F
```

Listing 8. Example of magic literals used for bit operations.

are self-describing or distinct values, then they could be improved Yann ►*How could they be improved?*◄ following this pattern.

Literals in test/example methods are otherwise acceptable because:

1. Tests or examples should stay as simple as possible. Calling methods to get some values or using class variables would make the code more complex.
2. A test class may contain many tests (methods) each using different literals. Refactoring all these literals would inflate the code.
3. Literals in tests/examples are used to build instances of "fake" objects only used locally and transiently.
4. Nic ►*This one seems much less pertinent*◄ Yann ►*Agree, and not directly related to tests/examples*◄ Literals arrays offer a compact way to create collections of different kinds using conversion methods e.g., #(2 3 5 2) asSet.

3.3 Unacceptable (Magic) Literals

Literals are said to be *magic* because their roles and meanings is not explicit. We define a *magic literal* as any literal that does not fall in one of the acceptable categories above, in Section 3.2. Objective reasons can be found to reject the usage of such magic literals from four perspectives on code quality: understandability, logic duplication, lack of documentation, missed opportunity for customisation.

Understandability. The usage of magic literals reduces code understandability and hinders program logic understanding.

Listing 8 shows a method with a detailed comment about its purpose and returned value: an integer encoding the format describing the kind of an object. However, this method is difficult to understand because of its magic literals. Line 8 is shifting the bits of the integer returned by the call to format 16 times to the right, which is used to discard the 16 least significant bits. Line 9 is applying the bit-wise "and" operation to the shifted integer and 16r1F literal.

The purpose of the bit-wise "and" is not explicit for anyone not familiar with base-16. Looking at the representation of this literal in base-2 (2r11111), the purpose of this operation

```
EncoderForV3 » genJumpLong: arg1
 (arg1 >=  −1024 and: [ arg1 < 1024 ])
  ifTrue: [ | tmp2 |
    tmp2 := stream.
    tmp2
      nextPut: 160 + (arg1 + 1024 bitShift:  −8);
      nextPut: (arg1 + 1024) \\ 256.
    ^ self ].
 ^ self
  outOfRangeError: 'distance'
  index: arg1
  range:  −1024
  to: 1023
```

Listing 9. Example of a mehthod with multiple occurrence of the magic literal 1024.

becomes clearer: it extracts the 5 least significant bits from the receiver of bitAnd:. Still, we do not know why it needs to extracts these bits.

Thus, using a base-2 representation could improve understandability. Another improvement would be to extract methods returning the values 16 and 16r1F as instSpecOffset and instSpecMask, respectively. (However, the performance cost of calling these methods should not be overlooked.)

Logic Duplication. When the same magic literal is repeated over and over, it makes the code more difficult to understand. Indeed, two difficulties arise from such duplication: (1) as with any duplication, a typo can happen in some occurrences, which create bugs and confusion for developers and (2) it is not clear if the (duplicated) values are coincidental or not.

For example, Listing 9 shows many occurrences of the magic literal "1024", which could refer to the same thing, *i.e.,* the maximal length of a jump in the bytecode: changing one would require changing all the others. More subtle, and not explicit, are the relationships between the magic literals "1024", "1023", and "256".

Lack of Documentation. Using magic literals reduces the quality of the documentation. Developers usually attempt to have code as self-explanatory as possible to reduce the need for documentation, in turn to reduce the need for maintaining this documentation.

The usage of magic literals decrease the self-explanatory quality of the source code because these literals have "hidden meanings" that can not be extracted from the literals alone. They require additional knowledge that, if not documented, is difficult to obtain. Listing 9 illustrates this need for documenting magic literals (and their relationships).

Modularity. Magic literals reduce the modularity of methods in which they occur. The occurrence of a magic literal freezes its value in the source code, preventing sub-classes or client code to change its value, Listings 10 and 11.

```
meaningOfLife
  ^ 42
```

Listing 10. Example of limited modularity.

```
meaningOfLife
  ^ meaningOfLife ifNil: [ meaningOfLife := 42 ]

meaningOfLife: anotherMeaning
  meaningOfLife := anotherMeaning
```

Listing 11. Example of improved modularity.

Modularity can be improved by replacing the magic literals by method calls:

1. Extracting the literal in a method that directly returns it. This allows sub-classes to change the value by overriding the method. However, it cannot be customized per instance.
2. Extracting the literal in a method that returns the value of an instance variable initialized with the literal by default. Initialization can be done lazily in the method. This allows users of instances to customize the values to fit their needs.

4 Literals in Source Code Quantitatively

Now that we have defined and studied magic literals qualitatively, we want to assess the prevalence of literals (magic or not) in the source code of different systems. For this study, we analysed seven projects of varying sizes and domains:

***Morphic*:** The graphical user-interface library of Pharo;
***Parser*:** A small parser/scanner library. We rarely include it in the discussions because of its size while expecting high proportions of character literals;
***Pharo*:** A Smalltalk-like programming language as well as a development environment. We consider *Morphic* separately from *Pharo*;
***Polymath*:** A mathematical library to represent mathematical abstractions such as Fourier transforms, Random number and statistical-distribution generators…;
***Roassal*:** A visualization engine to draw graphs or graphics of data and source code;
***Seaside*:** A server-side framework for developing and running Web applications;
***VMMaker*:** (also known as OpenSmalltalk-VM) A generator of the Smalltalk virtual machine (in C) from a description in a higher abstraction level language.

Pharo is a large system while *Parser* is a small one; the others are medium size, with *Roassal* and *Seaside* on the upper range, as shown by Table 1 (first two columns).

4.1 Descriptive Statistics

Table 1 provides descriptive statistics of the studied systems studied and the literals that they contain. Almost half (47.5%) of the methods contain literals, which confirms that literals are prevalent and magic literals could be an issue.

Two systems stand out: *VMMaker* (73% of methods with literals) and *Polymath* (61%). They have the highest proportions of methods with literals and the highest concentrations of literals in methods with literals. We explain this observation because *VMMaker* uses strings and integers for the generated C code while *Polymath* obviously deals with numbers.

Tests have much higher proportions of literals, 82.2% of methods with literals, and also higher densities of literals, 5.8 literals per methods instead of 4.0 overall (including tests themselves). This observation seems natural as tests must compare computed values to some golden standards.

Again, *VMMaker* and *Polymath* stand out as the systems with the highest densities of literals in test methods, 10.2 and 8.9, respectively. *Seaside* also comes out as a system with a high proportion of test methods containing literals, 92.9%.

Considering the proportions of literals found in tests, there are two groups. On the one hand, *Polymath*, *Pharo*, and *Seaside* have around half of their literals in tests methods, from 43.5% for *Pharo* to 67.4% for *Polymath*. They are also the systems with higher proportions of test methods, from 11.4% for *Seaside*, to 29.2% for *Polymath*.

On the other hand, *Morphic*, *Roassal*, and *VMMaker* have less than 6% of their literals in tests, from 3.3% for *VMMaker*, to 5.6% for *Roassal*. They are also the systems with less test methods, from 1.4% for *Morphic*, to 1.7% for *Roassal*.

However, the relation between the numbers of test methods and of test literals is not direct. For example, one third ($813/2,782 \simeq 29\%$) of *Polymath* methods are tests, but they contain two thirds ($6,709/9,947 \simeq 67\%$) of all the literals. Similarly, 15% of *Pharo* methods are tests ($14,198/94,872$), but they contain 44% of its literals ($62,906/144,454$).

4.2 Literal Types Distribution

Table 2 provides the numbers and percentages of each type of literals for the seven studied systems and overall.

Integer is the most frequent type except for *Seaside*. We believe this observation is due to the dual nature of integers as "domain oriented" and "implementation oriented", which make them ubiquitous. *Seaside* is the only system to have more *string* literals (56.1%) than *integer* literals (19.9%), because it uses string literals to describe HTML elements.

String and *Symbol* are the second and third most frequent types. Strings as often "domain oriented" and *symbols* "implementation oriented" Yann ►*Not sure we should keep this "domain" vs. "implementation"*◄. The prevalence of *symbols* over *strings* in *VMMaker*, 18.1% and 11.1% respectively, is possibly due to its nature as a system performing batch computations with little to no user input.

Table 1. Descriptive statistics on the seven studied systems (numbers of classes, methods, and literals; percentages of methods with literals, average numbers of literals per method that do have literals).

	Overall (including tests)					Tests only			
	#Class	#Method	Meth. with Lit.	#Literal	Lit./Meth.	#Method	Meth. w/ Lit.	#Literal	Lit./Meth.
Morphic	370	8 792	3 749 (42.6%)	10 362	2.8	123	89 (72.4%)	485	5.4
Parser	4	129	72 (55.8%)	236	3.3	0	0 (0.0%)	0	0.0
Pharo	7 725	94 872	41 433 (43.7%)	144 454	3.5	14 198	11 506 (81.0%)	62 906	5.5
Polymath	292	2 782	1 699 (61.1%)	9 947	5.9	813	754 (92.7%)	6 709	8.9
Roassal	882	9 952	4 457 (44.8%)	17 339	3.9	169	144 (85.2%)	978	6.8
Seaside	648	5 337	2 627 (49.2%)	8 194	3.1	607	564 (92.9%)	3 583	6.4
VMMaker	336	15 244	11 105 (72.8%)	66 860	6.0	232	218 (94.0%)	2 219	10.2
overall	10 257	137 108	65 142 (47.5%)	257 392	4.0	16 142	13 275 (82.2%)	76 880	5.8

Table 2. Distributions of the types of literals for each studied system (number of literals per type followed by their percentages over all literals for that system).

	all		*Morphic*		*Parser*		*Pharo*		*Polymath*		*Roassal*		*Seaside*		*VMMaker*	
	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
Int.	111 418	43.3%	4 385	42.3%	45	19.1%	52 486	36.3%	6 864	69.0%	8 738	50.4%	1 629	19.9%	37 271	55.7%
Strng	54 801	21.3%	827	8.0%	49	20.8%	38 030	26.3%	414	4.2%	3 442	19.9%	4 598	56.1%	7 441	11.1%
Symb.	41 733	16.2%	2 314	22.3%	53	22.5%	24 381	16.9%	120	1.2%	2 113	12.2%	650	7.9%	12 102	18.1%
Bool.	21 133	8.2%	1 316	12.7%	22	9.3%	11 530	8.0%	138	1.4%	560	3.2%	481	5.9%	7 086	10.6%
Arr.	10 552	4.1%	310	3.0%	6	2.5%	7 712	5.3%	820	8.2%	790	4.6%	305	3.7%	609	0.9%
nil	8 130	3.2%	815	7.9%	9	3.8%	5 105	3.5%	89	0.9%	323	1.9%	293	3.6%	1 496	2.2%
Float	5 314	2.1%	330	3.2%	0	0.0%	2 022	1.4%	1 457	14.6%	1 227	7.1%	182	0.7%	222	0.3%
Char.	4 311	1.7%	65	0.6%	52	22.0%	3 188	2.2%	45	0.5%	146	0.8%	56	2.2%	633	0.9%
total	257 156	100%	10 362	100%	236	100%	144 454	99.9%	9 947	100%	17 339	100%	8 194	100%	66 860	99.8%

Polymath differs from the other systems with very few *strings* or *symbols* literals, 4.2% and 1.2% respectively, but much more *float* (14.6%) and *array* (8.2%) literals.

Characters seem to be generally used rarely, except for in *Parser*, which uses these characters in its parsing rules.

4.3 Distribution Over AST Roles

Table 3 shows that literals, on average, appear half of the time (49.9%) as message *argument* and a quarter of the time (14.1%) as *operand*. Considering its mathematical content, we were surprised that *Polymath* ranks only third (28.7%) in proportion of *operands*, after *Roassal* (31.4%) and *Morphic* (34.7%), two graphical systems.

VMMaker has a very high proportion (22.9%) of literals used in *pragmas* (*i.e.,* annotations) compared to the second (*Roassal* with 1.7%), because Yann ►*Could we provide some reason?*◄

The proportion of literals *returned* is low overall, 5.9%. Two systems stand out: *Morphic* on the high end with 9.0% and *Polymath* on the low end with 1.0%. We cannot explain particularly these numbers, that seem to depend only on the ways these systems are implemented.

The two graphical systems, *Morphic* and *Roassal*, differ from the others as the only two above average for the proportions of literals directly *assigned* to variables, 6.8% and 7.1% respectively. Again, we cannot explain particularly these numbers, that seem to depend only on the system implementations.

4.4 Conclusion?

Bcp de literaux. Quid Magic Besoin d'une etude empirique pour definir les magic literaux

5 Detecting Magic Literals

Now that we have characterised magic literals and the prevalence of literals in the seven systems, we want to propose heuristics to detect magic literals in source code. We asked developers to study and characterise literals as magic or not. Then, based on their characterisation, we built heuristics for detecting magic literals, which we evaluated against a golden standard built manually.

Table 3. Distribution of the AST roles of the literals for each studied system (numbers of literals per role; percentages of literals over all literals for the system).

	all		*Morphic*		*Parser*		*Pharo*		*Polymath*		*Roassal*		*Seaside*		*VMMaker*	
	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
Arg.	128 471	49.9%	4 306	41.6%	90	38.1%	80 953	56.0%	5 653	56.8%	8 453	48.8%	5 852	71.4%	23 164	34.6%
Oper.	62 032	24.1%	3 596	34.7%	116	49.2%	32 597	22.6%	2 853	28.7%	5 439	31.4%	1 263	15.4%	16 168	24.2%
Prag.	17 665	6.9%	12	0.1%	0	0.0%	2 038	1.4%	21	0.2%	287	1.7%	0	0.0%	15 307	22.9%
Ret.	15 092	5.9%	937	9.0%	5	2.1%	8 366	5.8%	97	1.0%	866	5.0%	393	4.8%	4 428	6.6%
Assig.	12 879	5.0%	704	6.8%	20	8.5%	7 063	4.9%	412	4.1%	1 227	7.1%	311	3.8%	3 142	4.7%
Rec.	12 630	4.9%	319	3.1%	1	0.4%	9 090	6.3%	820	8.2%	802	4.6%	160	2.0%	1 438	2.2%
Seq.	7 770	3.0%	439	4.2%	4	1.7%	3 707	2.6%	91	0.9%	215	1.2%	215	2.6%	3 099	4.6%
total	257 392	99.7%	10 362	99.5%	236	100%	144 454	99.6%	9 947	100%	17 339	99.7%	8 194	100%	66 860	99.8%

Table 4. Descriptive statistics on the qualitative experiment (numbers of classes, methods, and literals; percentages of methods with literals; average numbers of literals per method that do have literals).

	Overall (including tests)						Tests only				
	# Class	# Method	# Literal	#uniq. literal	Literal redun.	Literal /Method	# Method	# Literal	#uniq. literal	Literal redun.	Literal /Method
Morphic	75	194	544	536	1.5%	2,8	20	73	73	0.0%	3.7
Parser	4	52	204	204	0.0%	3,9	0	0	0	0.0%	0.0
Pharo	853	1 137	6 110	5 129	19.1%	5,4	344	2 716	2 181	24.5%	7.9
Polymath	192	644	5 076	3 995	27.1%	7,9	287	3 549	2 742	29.4%	12.4
Roassal	282	607	4 108	3 112	32.0%	6,8	85	963	682	41.2%	11.3
Seaside	277	648	3 487	3 003	16.1%	5,4	193	1 985	1 603	23.8%	10.3
VMMaker	145	495	4 393	3 894	12.8%	8,9	54	968	684	41.5%	17.9
overall	1 828	3 777	23 922	19 873	20.4%	6,3	983	10 254	7 965	28.7%	10.4

5.1 Qualitative Analysis of Magic Literals

We ask 26 developers to study and characterise literals as magic or not. Yann ►*Need some info. about the developers*◄ . We provided each developer with Yann ►*How many?*◄ batches of 50 methods having literals. We presented the methods to a participant one at a time with the source code of the method, its class and package, and the list of its literals Yann ►*in addition to the method body?*◄ . The participant would classify it as *Magic*, *NotMagic*, or *Undecided*. The participant had access to the entire Pharo environment if needed (to study the method in its class, related methods, etc.).

Participation was entirely voluntary and anonymous Yann ►*Was it anonymous?*◄ , and participants could choose to work on any system(s). We only asked them to keep an overall balance Yann ►*What does this mean?*◄ on the number of batches analysed in the seven systems.

Table 4 shows 23,922 literals, 19,873 of them unique, where evaluated: 20% redundancy: $(23,922 - 19,873)/19,873$. The fact that there is more redundancy for tests (28.7%) is probably due the larger numbers of literals per test methods.

We designed literal redundancy to control for any bias from the participants themselves, *i.e.,* whether participants' classification of literals related to some of their own characteristics. We focused Yann ►*What does this mean?*◄ on the developers' expertise in a given system. We assumed Yann ►*Do we study this assumption somewhere? We should mention that at the beginning of this section.*◄ that experts in a system would qualify literals as *Magic* less often than novices.

By design Yann ►*Why do that?*◄ , test methods are slightly over-represented by ensuring that all batches had at least five or 10% of *test* methods. The experiment considered 3,777 methods with literals (5.8% of the 65,142 total methods with literals) and 983 *test* methods with literals (7.4% of the 13,275 *test* methods with literals). Again, we wanted to ensure that we had enough test literals to analyse them separately.

The number of literals per methods is almost always larger in the experiment (overall: 6.3) than for the entire systems (overall: 4.0), see Table 1). For all methods, this could be a result of the tests being over-represented because they do have more literals per methods.

Table 5 summarises the results of the participants' analyses and shows that:

- We do not include *Parser* because:
  - It is very small;

Table 5. Numbers of literals per type of literals for each studied system and percentages of magic literals.

	all		*Morphic*		*Pharo*		*Polymath*		*Roassal*		*Seaside*		*VMMaker*	
	#	magic	#	magic	#	magic	#	magic	#	magic	#	magic	#	magic
Int.	13 958	31.7%	242	18.6%	3 239	27.0%	3 469	35.5%	2 764	27.0%	1 315	18.4%	2 929	43.7%
Strng	3 056	10.9%	36	0.0%	874	6.5%	156	12.8%	351	26.2%	1 231	9.9%	408	10.5%
Symb.	1 935	6.6%	99	2.0%	792	6.1%	68	4.4%	269	18.6%	171	7.0%	536	2.2%
Bool.	907	9.9%	56	0.0%	234	5.6%	46	23.9%	91	17.6%	159	3.1%	321	14.0%
Arr.	1 285	21.2%	26	7.7%	441	14.7%	330	33.3%	219	26.9%	214	7.9%	55	36.4%
nil	750	14.7%	42	4.8%	263	13.3%	52	36.5%	95	21.1%	222	7.2%	76	23.7%
Float	1 408	42.8%	29	17.2%	91	25.3%	933	51.9%	311	27.3%	32	6.3%	12	33.3%
Char.	419	22.7%	14	0.0%	176	38.1%	22	9.1%	8	0.0%	143	9.8%	56	21.4%
total	23 718	25.5%	544	10.3%	6 110	19.4%	5 076	37.1%	4 108	26.0%	3 487	12.3%	4 393	32.6%

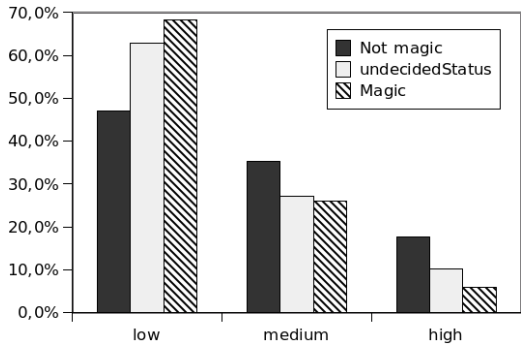

Figure 1. Classification of literals according to expertise of the participant. Note that there are about 17 000 *NotMagic*, 6 000 *Magic*, and 1 000 *Undecided*

  - It was evaluated by only one participant;
  - It has no redundancy;
  - All its literals are considered *NotMagic*;
- Results vary per system: *Morphic* and *Seaside* have only 10.3% and 12.3% of literals considered *Magic* while *Polymath* has 37.1% and *VMMaker* 32.6%.
- *Float* literals are more often considered *Magic* (42.8%) even for system in the mathematical domain (*Polymath*: 51.9%). Results for float literals in *Morphic*, *Pharo*, *Seaside*, and *VMMaker* may be insignificant because such literals seldom occur (only 6.3% in *Seaside*).
- *Symbols*, *Boolean* literals, *Strings* are considered generally less *Magic* literals. Our explanation is that *Symbols* and *Strings* are self-explanatory and *Boolean* literals have only two possible values.
- *nil* and *array* literals are considered *Magic* depending on the systems: only 4.8% of *nil* literals are considered *Magic* in *Morphic* while 36.4% of *Array* in *VMMaker* and 36.5% of *nil* in *Polymath*). However, these results may not be very significant because they have few occurrences in the systems.

Figure 1 shows, for each category (16 883 *NotMagic*, 986 *Undecided*, and 6,053 *Magic* literals), the proportion according to the participants' expertise, as self-declared by the participants. It shows that there is a correlation between considering a literal magic or not and the expertise in a system. This observation is confirmed by a χ^2 test, p-value < 0.001.

5.2 Heuristics for Magic Literals

We now explain how a heuristic could detect magic literals in practice.

Literals Considered as Not Magic. From the definition provided in Section 3.2, true, false, nil Yann ►*But we discussed nil in the previous subsection, in the bullet list, last bullet?*◄ , #(), {}, and empty string (' ') are never considered as magic by the heuristic.

Yann ►*I would put this later, after describing all the rules, maybe even after running some experiments, which would justify the need for a whitelist.*◄ Additionally, we decided to extend this set of literals with a white list of literals to consider as non-magic. We created this white list empirically by exploring literals of the system. It is customisable by users of the heuristics. By default, it includes: 0, 1, and -1 because we observed that these literals were understandable when reading a method body.

Literals Related to Coding Conventions. We provide a mechanism to ignore literals that are either receiver of or provided as argument of certain methods because they follow known code conventions. This mechanism is based on rules to ignore occurrences of literals in certain context.

Table 6 lists literals ignored in the context of certain messages. The "Selector" column is the selector of the message setting the context in which the literal is to be ignored. The "Role" column describes whether the literal is receiver or argument of the message. The "Literal" column describes the literal (or group thereof) to ignore. The "Arg. Index" column contains a value only if the role value is "Argument" and

Table 6. Literals ignored in the context of certain message because they follow known code conventions.

Selector	Role	Literal	Arg. Index
-	Argument	1	1
nextPut:	Argument	Character	1
«	Argument	Character	1
nextPutAll:	Argument	String	1
name:	Argument	String	1
«	Argument	String	1
ffiCall:	Argument	Array	1
ffiCall:option:	Argument	Array	1
ffiCall:module:option:	Argument	Array	1
to:	Receiver	Integer	N/A
to:by:	Receiver	Integer	N/A
to:do:	Receiver	Integer	N/A
timesRepeat:	Receiver	Integer	N/A

describes the position of the literal in the message arguments (index starting at 1).

Additionally, our heuristics ignore magic literals for all methods of certain classes because their usage is just part of their normal implementation. We identified two kinds of classes: sub-classes of BaselineOf and ConfigurationOf. Baselines and configurations are classes defining the packages forming a system, the dependencies among these packages, and the dependencies to external systems. They are descriptions of systems structures and, therefore, make a heavy usage of String literals to refer to package names, system versions, etc.

Heuristic to Detect Magic Strings. As discussed, some String literals contain natural-language sentences understandable when read by developers. Therefore, such literals are not *Magic*. However, detecting if a String contains a sentence in a natural language is difficult. Our heuristics uses a list of 479,000 English words[6] to determinate if a String literal contains natural-language words by:

1. Splitting the string on space characters and removing non-alphabetic characters.
2. Counting the number of sub-strings obtained that include English words.
3. Computing the ratio of sub-strings with English workds over the total number of sub-strings.

If the ratio is greater than a certain threshold, we consider a string literal as containing a natural-language sentence. After trials-and-errors, we fixed the ratio to 0.5, *i.e.,* if more than 50% of the words in a string are English words.

Yann ►*Could we put the heuristics for each type of literals (or family of literals) in a table? Maybe as "pseudo-code". Right now, it's hard to "see" what a heuristics may look like. Or at least give one example?*◄

[6]https://github.com/dwyl/english-words

5.3 Implementation and Evaluation

The heuristics to detect magic literals have been implemented in Pharo[7]. During the design, implementation, and evaluation of the heuristics, we observed at first many false positives, which limited their usefulness for developers. We took decisions Yann ►*What decisions? What changes to the heuristics?*◄ that favour usability over finding all magic literals (*i.e.,* precision over recall). We took some decisions that have counter-examples, which we discuss in Section 6.

To evaluate the accuracy of our heuristics to detect magic literals in source code, we run the heuristics on 112,500 methods of Pharo 7. We obtained 8,986 methods containing 23,292 magic literals. From these 8,986 methods, we randomly selected 100 methods and reviewed them manually. For each magic literal detected by the heuristics, we assigned one of the three labels:

- *True Positive.* The literal is is a indeed a magic literal.
- *False Positive.* The literal is not a magic literal.
- *Unclassified.* It is not clear whether the literal should be considered as magic or not.

In these 100 methods, the heuristics detected 243 magic literals. Among these 243 literals, we found that 151 are true positives belonging to 68 methods (with 60 containing only true positives), 74 are false positives in 28 methods (with 20 containing only false positives), and 18 could not be classified in 14 methods (with 11 containing only unclassified literals). We thus obtained a precision of 63%. These results suggest that the heuristics can reveal magic literals while being right about 63% of the time on this sample of 100 methods.

6 Discussion

We now discuss some aspects and limitation of our heuristics and their implementation.

6.1 Constant Values vs. Missed Parameter Literals

Some magic literals represent constant values that can never change. Examples of such constant are π, G (gravity constant), Euler number, etc. For these literals, we do not want to enforce modularity because it would be error-prone to let developers override the methods providing these constants. In Pharo, we could extract these constants and make them class variables in a SharedPool. Yann ►*Should we say about Java? C/C++?*◄

There are magic literals that should be instance variables or method parameters. They can be extracted into method that return directly the literal value or as a call to an accessor method whose instance variable is set to the literal value by default (as explained in Section 3.3).

The difference between constant values and missed Yann ►*?*◄ parameter literals cannot be extracted from structural

[7]https://github.com/juliendelplanque/MagicLiteralsInPharoExperiment

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

Listing 12. Example of literals as method argument that are explained by the method selector.

properties of a system. Thus, our heuristics cannot correctly detect such magic literals.

6.2 Magic Strings

We developed a heuristic to detect string literals Yann ►*Sometimes "string literals", sometimes "String literals", we should pick one.*◄ that are not self-explanatory because we consider that code understandability is important. However, from the point of view of modularity, the heuristic is wrong to consider recognized strings as understandable because they can limit the future evolution of the system. Yann ►*Not sure to understand what we mean here? Does it mean that any string literal should be considered magic?*◄

6.3 Method Selector Explaining Argument Literal

We observed that literals appearing as method argument are sometimes explained by the selector associated to this argument. For example Listing 12 shows the method selector Date class»year:month:day:. It is obvious that 2019 refer to the year 2019, 6 refer to June, and 1 refer to the first day of the month.

6.4 Symbols for Configuration

Julien ►*I am still not convinced it is a good practice. Need to discuss.*◄

6.5 Scripting vs. Extensible Code

Stef ►*put mini roassal example*◄

7 Related Work

To the best of our knowledge, no similar study of magic literals in Pharo or other Smalltalk dialects exist in the literature. This paper is thus the first to address this problem in the context of Smalltalk.

The concept of magic number is discussed by Fowler [4] and Martin [6], who describe it as a code smell that should be avoided. Both authors claim that magic numbers should be replaced with symbolic constant to improve understandability. They do not discuss the modularity problem caused by magic literals that should be parameterisable. (We chose to use the term "magic literal" because, as stated by Martin, *"The term "Magic Number" does not apply only to numbers. It applies to any token that has a value that is not self-describing."* [6].)

Smit *et al.*, [11, 12] identified the relative importance to maintainability of 71 coding conventions based on a survey of seven developers. They analysed the revisions of four different open-source systems and observed that, when developers are conscious of conventions (via explicit coding conventions policy and checks made by continuous integration servers), they put an effort in respecting these conventions. When developers are not conscious of conventions, violations are prevalent. One of the coding conventions is the usage of magic numbers. Their definition considered -1, 0, 1, and 2 as non-magic. Julien ►*Need to discuss more that, about the 2*◄ They report that avoiding magic numbers in source code is considered important by the developers. However, in the analysis of the four systems, they observed many magic numbers and "avoiding magic numbers" appeared three times as the third and once as the fourth most violated convention.

Julien ►*This page contains stuff to compare our paper claims with:*◄ http://wiki.c2.com/?MagicNumber

Julien ►*This page is about why we should only consider -1, 0, and 1 as valid literals:*◄ http://wiki.c2.com/?ZeroOneInfinityRule

Nundhapana and Senivongse [10] discussed the approach taken by an IT organization in Thailand to enforce naming conventions in Objective-C. They developed a library to check for naming conventions automatically. In particular, magic numbers are considered as violation of naming conventions because they are unnamed literal constants. They devised a regex-based checker to detect magic numbers in Objective-C source code. They reported Yann ►*Any precision? Any results?*◄

Both Smit *et al.*, [11, 12] and Nundhapana *et al.*, [10] distinguished "magic numbers" from "literal strings" but "literal strings" the string equivalent of magic numbers. The coding violation is different because literal strings concern multiple occurrences of the same string literal inside a Java / Objective-C file that should be extracted into a constant. Yann ►*Not sure to understand the point we're trying to make?*◄ The concept of string literals does pertain to the self-explanatory property of the source code but rather to its modularity. Indeed, the occurrence of a string can be misleading for a developer reading the code.

Mukherjee [9], that explained how to perform static analysis and visualization with Roslyn and JavaScript, presented three techniques for finding magic numbers in C♯ systems Yann ►*Just before we talked abot JS, is that a contradiction?*◄ , in particular in *arithmetic expressions*, *array indices*, and *conditions*.

Cris ►*Results from PMD, checkstyle, Sonarqube, and a book. To be fleshed out.*◄

The tool PMD[8] is a static-code analyzer that finds common programming flaws defined in rules. PMD has the following rules concerning literals:

- AvoidDuplicateLiterals https://pmd.github.io/latest/pmd_rules_java_errorprone.html#avoidduplicateliterals.

[8] https://pmd.github.io/

- AvoidLiteralsInIfCondition https://pmd.github.io/latest/pmd_rules_java_errorprone.html#avoidliteralsinifcondition.

```
@MyAnnotation(6) // no violation
 class MyClass {
  private field = 7; // no violation

  void foo() {
   int i = i + 1; // no violation
   int j = j + 8; // violation
  }
 }
```

Listing 13. Example of *checkstyle* rule for magic numbers.

Similarly, CheckStyle[9] has the following rules pertaining to literals:

- MultipleStringLiterals https://checkstyle.sourceforge.io/config_coding.html#MultipleStringLiterals Checks for multiple occurrences of the same string literal within a single file. Rationale: Code duplication makes maintenance more difficult, so it can be better to replace the multiple occurrences with a constant.
- MagicNumber https://checkstyle.sourceforge.io/config_coding.html#MagicNumber checks that there are no "magic numbers" where a magic number is a numeric literal that is not defined as a constant. By default, -1, 0, 1, and 2 are not considered to be magic numbers, as illustrated by Listing 13.

Sonarqube[10] has the following rules for literals in Java, many of which are very specific:

- String literals should not be duplicated https://rules.sonarsource.com/java/RSPEC-1192.
- Literal boolean values should not be used in assertions https://rules.sonarsource.com/java/RSPEC-2701.
- Ints and longs should not be shifted by zero or more than their number of bits-1 https://rules.sonarsource.com/java/RSPEC-2183.
- Regex patterns should not be created needlessly https://rules.sonarsource.com/java/tag/performance/RSPEC-4248.
- URIs should not be hardcoded https://rules.sonarsource.com/java/RSPEC-1075.
- Magic numbers should not be used https://rules.sonarsource.com/java/RSPEC-109 -1, 0, and 1 are not considered magic numbers.

Cris ►*Make a table of the (similar) rules supported by each tool?*◄

Cris ►*Note that I only considered Java language. Many tools support multiple languages, so perhaps more specific useage rules are present.*◄

Cris ►*end*◄

[9]https://checkstyle.sourceforge.io/

[10]https://www.sonarqube.org/

8 Conclusion

Yann ►*Not changed*◄

In this paper we explored the concept of magic literal generally and more specifically in the context of Pharo. We did an exploration of literals occurrence in the system leading us to a characterisation of acceptable literals and a definition of magic literals. We used the definition to implement a heuristic to identify magic literals as a Renraku [13] rule. We manually evaluated a subset of the heuristic results and found that it identify correctly magic literals 63% of the time.

The research we conduct in this article opens multiple perspectives. We want to dig deeper in the analysis of magic literals occurrence by studying how and why they occur project per project. We have the hypothesis that some domain are probably more subject to the usage of magic literals than others. We would like to test this hypothesis on a large set of Pharo projects addressing various problems of different domains.

In the same direction, some domains are probably using more magic literals of certain types than others (for example a mathematical library is likely to use more Number magic literals than String magic literals). Finally, an empirical study of the evolution of magic literals across multiple versions of these Pharo projects will help us to understand why magic literals appear. Such study consists in doing a post-mortem analysis of commits that occurred during the development of the project. We will compute the difference between each pair of consecutive versions of each project and watch for magic literal apparition.

References

[1] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion.* Addison Wesley, Boston, MA, USA, 1998.

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

[3] Stéphane Ducasse, Eliot Miranda, and Alain Plantec. Pragmas: Literal messages as powerful method annotations. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, August 2016.

[4] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, Boston, 2 edition edition, November 2018.

[5] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison Wesley, Reading, Mass., May 1983.

[6] Robert Martin. What killed Smalltalk could kill Ruby, too. (RailsConf 09 – http://blip.tv/file/2089545).

[7] Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009.

[8] Gerard Meszaros. *XUnit Test Patterns – Refactoring Test Code.* Addison Wesley, June 2007.

[9] Sudipta Mukherjee. *Source Code Analytics With Roslyn and JavaScript Data Visualization.* Apress, 2017.

[10] Ruchuta Nundhapana and Twittie Senivongse. Enhancing understandability of objective c programs using naming convention checking framework. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 1, 2018.

[11] Michael Smit, Barry Gergel, H James Hoover, and Eleni Stroulia. Code convention adherence in evolving software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 504–507.

IEEE, 2011.

[12] Michael Smit, Barry Gergel, H James Hoover, and Eleni Stroulia. Maintainability and source code conventions: An analysis of open source projects. *University of Alberta, Department of Computing Science, Tech. Rep. TR11*, 6, 2011.

[13] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. Renraku: The one static analysis model to rule them all. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '17, pages 13:1–13:10, New York, NY, USA, 2017. ACM.

[14] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[15] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *Transactions on Software Engineering*, 33(12):800–817, 2007.

[16] Oleksandr Zaitsev, Stéphane Ducasse, and Nicolas Anquetil. Characterizing pharo code: A technical report. Technical report, Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille ; Arolla, jan 2020.