# Magic Literals in Pharo

Julien Delplanque, Stéphane Ducasse, Oleksandr Zaitsev

# Magic Literals in Pharo

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

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

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

## Abstract

Literals are constant values (numbers, strings, etc.) used in the source code. *Magic literals* are the ones used without a clear explanation of their meaning. Presence of such literals harms source code readability, decreases its modularity, and encourages code duplication.

Identifying magic literals is not straightforward. A literal can be considered self-explanatory in one context and magic in another. We need a heuristic to help developers spot magic literals.

In this article, we study and characterize the literals in Pharo. We implemented a heuristic to detect magic literals and integrated it as a code critic rule for System Browser and Critics Browser in Pharo 7.

We run our heuristic on 112,500 Pharo methods which reported 23,292 magic literals spread across 8,986 methods. We manually validated our approach on a random subset of 100 methods and found that 62% of the reported literals in those methods are indeed magic.

***Keywords*** Software analysis, Quality, Idioms, Extensibility

## 1 Introduction

*Code smells* [FBB+99, SSS14, SS18] are characteristics of source code that indicate certain coding practices lowering code quality. Those bad practices are formalized as *anti-patterns* [Koe95, BMMM98]. They describe code situation providing a counterproductive response to a recurring problem [Koe95, BMMM98, SSS14, SS18]. Multiple anti-patterns have been discovered and discussed in the literature [BMMM98, AKGA11].

One of them is *Magic literal*: literal of any kind that appears in source code without a clear explanation. For example, numbers 4 and 52 in Listing 1 are magic literals. Depending on the knowledge of the developer, it can be hard to infer the meaning and purpose of those values just by reading the source code. For a developer who has no knowledge of the domain of card games, it can be complicated to guess that 4 is the number of players and 52 is the number of cards in the game.

```
CardGame » initialize
  deck := (1 to: 52) asOrderedCollection.

CardGame » distributeCards
  ^ (1 to: 4) collect: [ :i |  cards remove: cards atRandom ]
```

**Listing 1.** Literals that are not clearly explained make code harder to comprehend and reuse.

Magic numbers are indeed an anti-pattern [Bec97, FBB+99]. They make source code harder to understand and decrease its modularity. When code is duplicated, magic numbers increment complexity and risk of bugs due to their use in different locations [Bak97, DRD99, KN01, KKI02].

It is recommended to replace magic literals with well-named constants or method calls. For example, we can improve the code in Listing 1 by replacing value 4 with argument named numberOfPeople and returning the value 52 from a method called deckSize. As can be seen in Listing 2, the code without magic numbers is much easier to comprehend. It also allows us to change both values either by passing a different number of players as argument or by overriding the "deckSize" method.

```
CardGame » initialize
  deck := (1 to: self deckSize) asOrderedCollection.

CardGame » distributeCards: numberOfPlayers
  ^ (1 to: numberOfPlayers)
      collect: [ :i | cards remove: cards atRandom ]
```

**Listing 2.** Magic literals should be replaced with well-named constants

In literature, authors use the term *"magic number"* to refer to any kind of undocumented literal. Robert Martin [Mar09] highlights:

> The term "magic number" does not apply only to numbers. It applies to any token that has a value that is not self-describing.

These can be strings, symbols, characters, arrays, etc. Consider the following example:

```
label text: 'e4c0c62'.
```

**Listing 3.** Magic literal that is not a number.

The string literal 'e4c0c62' should be replaced with a descriptive variable named commitSHA. It is therefore the *"magic string"*. To avoid confusion, we adopt the term *"magic*

*IWST'19, August 27-29th, 2019, Cologne, Germany*
2019.

*literals"* and use *"magic numbers"* only when referring to magic literals that are numbers.

Magic literals seem to be very common in software projects [SGHS11b]. We need to assist developers both by warning them whenever they introduce a magic literal and by helping them extract the list of all magic literals from a given project to refactor them. However, detecting magic literals is not an easy task. Some literals can be self-explanatory in certain context. For example, number 2 in Listing 4 should not be considered as magic literal. In this case, 2 is not a constant that can be named otherwise.

```
length := 2 * pi * radius.
```

**Listing 4.** Some literals (here 2) can not be renamed.

In this article, we introduce a heuristic for identifying magic literals based on the context in which they appear. It is the algorithm that reports undocumented numbers such as those presented in Listing 1 as magic literals and try not report self-explanatory numbers like the one in Listing 4.

To introduce the heuristic for detecting magic literals, we study the usage of literals in latest Pharo 7 image. Syntax of Smalltalk programming language and the conventions adopted by Pharo developers create many exceptional cases that must be considered. For example, since indexing of Smalltalk collections starts from 1, number 1 in statement `1 to: n` is not a magic literal. And because of the special nature of tests, numbers and other literals that appear in tests are not considered magic [VDDR07]. We implement a heuristic for detecting magic literals in Pharo and add it as a code critic rule integrated with System Browser and Critics Browser[TGN17]. In Section 7.5, we propose several strategies for refactoring magic literals and benchmark those strategies to compare their performance.

The rest of the article is organized as follows: In Section 2 we provide a definition for "literal" and in Section 3 we explore the distribution of literals in Pharo. In Section 4 and 5 respectively we characterize acceptable literals and define magic literals. In Section 6 we present the heuristic developed to detect magic literals in Pharo source code and evaluate the accuracy of this approach. In Section 7 we discuss properties and limitations of our heuristic. It is followed by Section 8 where we do an overview of research related to the topic of this article. Finally, in Section 9 we conclude the article and discuss perspectives.

## 2 Literals in Smalltalk

A commonly accepted definition for the concept of "literal" in the context of a programming language is: *"the notations for constant values of some types"* [ada, pyt]. A corollary of this definition is that data provided by a literal involves no computation during the program execution.

In Smalltalk, literals refers to the objects that are created by the parser and not via execution of messages.

Originally, Smalltalk-80 specifications define literals as *"Literals describe certain constant objects, such as numbers and character strings. [...] Five kinds of objects can be referred to by literal expressions. Since the value of a literal expression is always the same object, these expressions are also called literal constants. 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"* [GR83: p18–19]. This definition is similar to definitions used by other languages while being more precise on the kind of objects that can be referred to by literal expressions.

In the context of this paper, we use the Smalltalk definition of literals cited previously and slightly extend it with `true`, `false` and `nil`. We consider these 3 tokens as literals since they always refer to their respective object and their value is constant. Furthermore, in Pharo abstract syntax tree `true`, `false` and `nil` are not special nodes. They are the same literal nodes as numbers, characters, strings, symbols and array of other literals. This strengthens our proposition to consider them as literal.

## 3 Exploring Literals in Pharo

Using the previous definition of literals in Pharo, we explore their occurrences to understand how they are used. This analysis help us identify magic literals. To achieve this task, we collect all literals from latest Pharo 7 image[1] and analyze how they are distributed over data types, over tests and non-tests, and over the kinds of parent abstract syntax tree nodes.

### 3.1 Distribution Over Data Types

In Figure 1, we can see how the 169,133 literals found in Pharo 7 are distributed over the data types. We observe that literals referring to integer are the most common literal found in the system making more that one third of all literals. String is the second most common type of literals, their proportion being about one fourth of all literals followed by Symbol literals with slightly less than one fifth.

Boolean literals occupy the fourth place with a proportion of 8%. Array and UndefinedObject (nil) literals occur with a proportion close to 4%. The proportion of Character literals is about 2% and all other types combined take the remaining 2% of all literals.

### 3.2 Distribution Over Test and Non-Test methods

We analyze the distribution of literals over test and non-test. Pharo 7 image has 20,170 test methods (18%) and 92,330 non-test methods (82%).

Literals occur more often in test methods than in non-test methods. 63% of test methods have at least one literal, which holds for only for 40% of the non-test methods.

[1] Pharo-7.0.3+build.159.sha.87d28366ab24e00f43dbd5d91f1c0b01ec519e6f (64 Bit)

In Section 4.5, we discuss the special case of literals used in test methods and claim that they should not be considered magic.

### 3.3 Distribution Over the Kind of Parent Node

We identify 7 groups of literals based on the type of their parent node in the abstract syntax tree:

1. *Receiver.* The literal is the receiver of a message send.
2. *Argument.* The literal is an argument of a message send.
3. *Assignment.* The literal is assigned to a variable directly.
4. *Return.* The literal is returned by the method.
5. *Sequence.* The literal appears in a sequence of statements to be executed.
6. *Pragma.* The literal is an argument to a pragma.
7. *Array.* The literal appears in a curly-braces array.

Note that while it seems strange to have literals in a *Sequence*, the explanation is in fact straightforward: this is because of block closure. For example, this case appears when a literal is used in the block closure argument of a ifTrue: message.

In Figure 2, we can observe that literals mostly appear in message node (as receiver or argument). Furthermore, they appear much more often as argument (69% of literals) than as receiver (13%).

## 4 Acceptable Literals in Source Code

The presence of literals in source code is not wrong by essence. In some contexts, it is legit or needed to insert a literal value at a specific location in the code. It is then important for the detection of magic literals to identify the legit usage of literals. We present now the categories of acceptable literal values in source code.

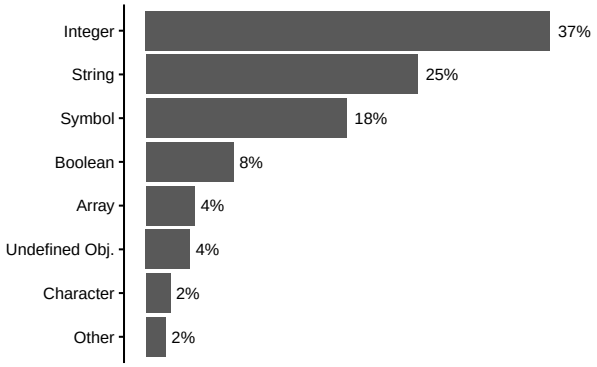


**Figure 1.** Distribution of literals over data types

### 4.1 Literals self-describing their semantics.

Some literals directly refer to the data they hold. In Smalltalk, true, false, nil, #() (empty literal array), {} (empty curly braces array) and '' (empty string) are part of this group. The two first refer directly to 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 semantic is obvious and well known by developers.

We will discuss two additional types of literals: *String*s and *Symbol*s. Introducing *String* literals in the source code might be legit. Indeed, if its contents aims to be read by a human (i.e., it is a sentence or part of a sentence in natural language) it is self-explanatory. Thus this kind of *String* literals should be considered as acceptable. However, some *String* literals can be considered as magic if their meaning is not clear. Those should be considered as unacceptable. An heuristic to distinguish these two kinds of Strings is discussed in Section 6.4.

In Pharo, *Symbol*s are a special kind of unique *String* – there is a single instance for each sequence of characters (Symbols are flyweight [ABW98]). Symbol uniqueness is what make them one of the cornerstones of identity-based data structures: they are readable unique keys. In the core of the system, symbols are used to represent method name (i.e., its selector), the name of a class, the name of a slot, to represent the name of a pragma, etc...

Symbols are also used in many projects as key in associative data structures such as dictionaries. A developer often uses a symbol as a key to get access to the value object.

Icon management is a good example. Indeed, an application does not directly refer to an icon object, instead it refer to it sending the message Object»iconNamed: with a symbol as argument being the name of the icon (see for example

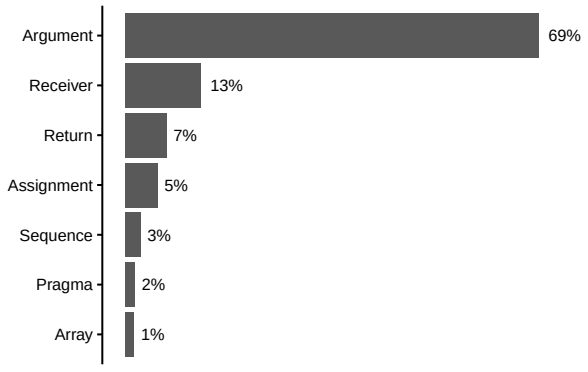


**Figure 2.** Distribution of literals over the kind of parent node. 82% of all literals appear in message sends: 69% as arguments and 13% as receivers

Listing 5). This mechanism allows one to change the set of icons to be used in Pharo without updating all icons users.

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

**Listing 5.** iconNamed: method call.

These examples drive us to the conclusion that occurrences of *Symbol*s in the source code should be considered as acceptable because they correspond to the name of a unique object. In addition a symbol has a readable form, contrary to a simple number which is also unique but does not convey a semantic information beside its actual value.

### 4.2 Literals that are part of API.

Programming languages usually define some internal conventions that must be respected by developers. An example of such a convention is the manner in which collections are indexed. For example, Java uses 0 as first index of its collections. Python uses index −1 to refer to the last element of a list. Such conventions do not only concern indexing of collections, in C there is no boolean data type. Instead, the language considers that 0 is false and any other value is true by convention. In practice, macros are used to avoid putting explicitly 0 or 1 literals in the source code.

In Pharo, such coding conventions are more contextual (in the sense that they depend on the exact message send). This is due to the extremely compact syntax of Smalltalk and the fact that all control flow operations are not part of the syntax but messages like any others. In Pharo a literal can be the receiver or an argument of a message. For example, copyFrom:to: sent to an instance SequenceableCollection copies the collection from the first argument index to the last. However, when a collection needs to be copied from beginning to a certain index, the literal 1 is provided as first argument (e.g., aCollection copyFrom: 1 to: n will copy the n first elements of aCollection as shown in Listing 6).

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

**Listing 6.** Literals as arguments.

As we discuss in Section 7.3, the same arguments can be considered magic in other languages, such as Java or C. Unlike in Pharo, method names in those languages can not describe multiple arguments.

Such literals are acceptable because the programming language explicitly expects developers to use them. Furthermore, since these conventions are described in the language specifications, it is highly unlikely that they get changed.

### 4.3 Literals directly assigned to a variable / returned by a method.

Some literals appear assigned to a variable (*e.g.* Listing 7) or returned by a method (*e.g.* Listing 8).

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

**Listing 7.** Literals assignments.

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

**Listing 8.** Method returning a literal directly.

In this case, the variable name or the method name has the responsibility to hold the semantics of the stored or returned literal. Both possibilities are acceptable.

### 4.4 Literals located in a method annotation arguments

In Pharo, methods can be annotated using *pragmas* [DMP16]. A pragma is a static annotation of the method. It acts as a tag and supports a declarative registration mechanism that can be used to easily retrieve and execute all methods having a specific tag.

Listing 9 shows the source code of a method that is annotated as example. This tag is used by the integrated development environment to let user run examples.

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

**Listing 9.** An example of <example> pragma usage.

Pragmas can be parameterized with arguments. However, such arguments can only be literals (in the sense of definition given previously [GR83: p18–19]) because pragmas are created at parse time.

Listing 10 shows an example of gtInspectorPresentationOrder: pragma. This pragma supports inspector extension. It adds an additional view on the object being inspected. The argument provided to this pragma is a number that orders the inspector views (the smaller, the first to appear). Thus, any literal that is the argument of a pragma is legit because, by design, it could not be another way.

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

**Listing 10.** An example of <gtInspectorFormIn:> pragma usage with an argument.

### 4.5 Literals located in test and example methods.

As we observed in Section 3.2, a part of literals in the system appear in test code. While exploring non-tests literals, we also saw that some of them are part of example code [vDMvdBK01, VDDR07].

Literals in test / example methods are acceptable for the following reasons:

1. Developers tend to keep tests and example source code as simple as possible. In this context, calling methods to get some values or using a class variable would make the code more complex.
2. Literals in test or example code are used to build instances of fake objects that are used locally in the method.
3. Example and test source code are usually not reused by other parts of the system.
4. Literals arrays offer a compact way to create collections of different kinds using conversion methods e.g., #( 2 3 5 2) asSet. They are widely used and acceptable in tests and examples.

## 5 Unacceptable Literals: The Magic Ones

Objective reasons can be found to reject the usage of literals in some contexts. Those literals are said to be *magic* because their role and meaning in the computation is obscure.

In the context of this article, we define *magic literal* as source code literal that does not fall into any of the acceptable literals categories described previously in Section 4.

We identify three main reasons why magic literals are bad from a code quality perspective: readability, logic duplication and modularity.

### 5.1 Readability

The usage of magic literals heavily reduces the code readability and as such the program logic understanding. A case where it is especially true is when a magic literal is used to apply bit operation on some input.

Let's take for example Listing 11. This method has a good comment about what it returns: an integer which encodes the format describing the kind of object an instance is. However, this method is not easy to understand with the two literals it holds. Line 8 is shifting the bits of the integer returned by the call to format method 16 times to the right. This step is used to discard the 16 least significant bit. Line 9 is applying the bit-wise "and" operation to the shifted integer and 16r1F literal.

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

**Listing 11.** Example of magic literals used for bit operations.

On the first look, it is not explicit what the bit-wise "and" aims to compute if one is not familiar with base-16. If one look at the representation of this literal in base-2 (2r11111), the purpose of this operation becomes clearer: it extracts the 5 least significant bits from the receiver of bitAnd:.

In the present case, using the base-2 representation of -16 is a constant time improvement. Another possible solution, to enhance the readability of this method is to extract 16 and 16r1F into two methods having named respectively instSpecOffset and instSpecMask.

The method names describe the purpose of these numbers. Applying this solution leads to Listing 12 source code. We discuss performances implications of such change in Section 7.5.

```
Behavior » instSpecOffset
 ^ 16

Behavior » instSpecMask
 ^ 2r11111

Behavior » instSpec
 [...]
 ^(self format bitShift: self instSpecOffset negated)
  bitAnd: self instSpecMask
```

**Listing 12.** Solution to remove magic literals used for bit operations.

Note that 16 literal was extracted in instSpecOffset instead of -16. The reason is that this number is negated only because of the call to bitShift: which expects positive numbers to shift bits right and negative numbers to shift bits left. Thus the negation of the offset is only there as a side effect of bitShift: usage. If, » method is used instead (see Listing 13), the argument does not need to be negated as this method shift the receiver bits to the right for positive numbers provided as argument.

```
Behavior » instSpec
 [...]
 ^(self format » self instSpecOffset)
  bitAnd: self instSpecMask
```

**Listing 13.** Alternative solution to remove magic literals used for bit operations.

### 5.2 Logic Duplication

When the same magic literal is repeated over and over again, it participates to create complex to understand source code

subject to duplication. This duplication of the magic literal can be quite harmful to the evolution of software. For example, if the literal is complex, a typo can be introduced in some literals that should refer to the same value creating potential bugs and confusion for people reading the code.

Furthermore, if a duplicated magic literal needs to be changed in the future, changing the source code will be difficult and error-prone.

For example, the magic literal 1024 appears 7 times in 3 methods of the class EncoderForV3 responsible of encoding a certain version of the byte-code. After reading these 3 methods, it appears that 1024 is the maximal length of a jump in the byte-code.

```
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 14.** One of EncoderForV3 method with multiple occurence of 1024 literal.

It is not possible to be 100% sure that all 1024 literals refer to the maximal length of a jump in the byte-code because it requires the knowledge of Smalltalk byte-code experts. However, some of these literals probably refer to the same concept and should be extracted in a method providing it a name.

### 5.3 Modularity

Magic literals reduce the modularity of methods in which they occur. Indeed, since the occurrence of a magic literal freezes its value in the source code preventing sub-classes or client to change its value.

We identify five strategies to address this problem.

1. Extract the literal in a class-variable and set it in its initialize method. This strategy is relevant if the literal refer to a constant that should never be changed such as a physical constant.

```
initialize
 MEANING_OF_LIFE := 42
```

2. Extract the literal in a method that directly returns it. This strategy allows sub-classes to change the value by redefining the method. However, it can not be customized per instance.

```
meaningOfLife
 ^ 42
```

3. Extract the literal in a method that returns the value of an instance variable initialized previously with the literal by default.

```
initialize
 meaningOfLife := 42

meaningOfLife
 ^ meaningOfLife
```

4. Extract the literal in a method that returns the value of an instance variable initialized with the literal lazily.

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

5. Extract the literal in an instance variable initialized previously with the literal and reference it directly.

```
initialize
 meaningOfLife := 42
```

Strategies 3 and 4 allows users of instances to customize the value of meaningOfLife to fit their needs.

We do not argue that one strategy is better than another, it depends on what kind of value the developer deals with. In Section 7, we evaluate differences between these strategies in terms of performance.

## 6 Detecting Magic Literals in Pharo

We have implemented the heuristic for detecting magic literals in Pharo[2]. We realized that it becomes unusable if too many false positives are generated. Thus, we took the decision to favor the usability of the heuristic over finding all magic literals. For this reason, some decisions taken in the implementation to identify magic literals can have counter-examples creating false negatives. These counter examples are discussed in Section 7.

In the following subsections, we explain how the implementation proceeds to detect magic literals in practice. Then, an evaluation of the implementation is performed on 100 methods considered as containing magic literals in Pharo.

### 6.1 Literals Considered as Not Magic

From the definition provided in Section 4, true, false, nil, #(), {} and empty string ('') are never reported as magic by the heuristic.

Additionally, we decided to extend this set of literals with a white list of literals to consider as non-magic. This white list has been created empirically while exploring literals of the system and is configurable by users of the heuristic. This white list includes: 0, 1 and -1. The rational behind this choice is that from our exploration we observed that most of the

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

time these literals were understandable when looking at the whole code of a method. Thus, while it happens that the white list make the heuristic miss some magic literals, it makes it more usable.

### 6.2 Literals that are part of API

We provide a mechanism to ignore literals that are either receiver of or provided as argument of certain methods because they are part of the API. This mechanism is based on a list of rules to ignore occurrences of literals in certain context.

Table 1 lists literals ignored in the context of certain message sends. "Selector" column is the selector of the message setting the context in which the literal is ignored. "Role" column describes whether the literal is receiver or argument of the message send. "Literal" column describes which literal (or group of literals via the class of these literals) is ignored. "Arg. index" contains a value only if the role value is "Argument" and describes the position of the literal in the message arguments (index starting at 1).

**Table 1.** Literals ignored in the context of certain message sends because they are part of the API.

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

### 6.3 Baselines and Configurations

Our implementation ignores magic literals for all methods of certain classes because their usage is just part of the API. We identified two kinds of classes: subclasses of BaselineOf and ConfigurationOf. Baselines and configurations allows one to define which packages belong to a project, the dependencies between these packages and the dependencies to external projects. Because these classes are descriptions of projects structures, they make a huge usage of String literals to refer to package names, project versions, etc... Thus, a lot of false positives are generated by these methods of these classes.

### 6.4 An heuristic to detect magic Strings

As discussed previously, some String literals contain natural language sentences which are understandable when read by a human. From that observation, these String literals as legit. However, detecting if a String contains a sentence in natural language or not is a complicated task.

In the context of this article, we used a simple but yet effective heuristic to deal with this problem. We build a trie data structure from a list of 479,000 English words[Eng]. This data structure allows one to check if a string is included in a list of string or not efficiently. This trie is used to determine if a String literal contains natural language sentences or not by:

1. Splitting the string on space characters and removing non alphabetic characters.
2. Counting the number of sub-strings obtained that are included in the trie.
3. Computing the ratio sub-strings included over the total number of substrings.
4. If the ratio is greater than a certain threshold, the string literal is considered to contains natural language.

The ratio of English words required a string literal as non-magic has been fixed to 0.5. The idea behind this threshold is that if more than 50% of words in a string literal are English words, the content held by this literal should be understandable by a developer.

### 6.5 Evaluation of the Approach

To evaluate our approach for detecting magic literals in the source code, we run the heuristic on the 112,500 methods of Pharo 7 and gathered 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 heuristic, we assigned one of the 3 following labels:

- *True Positive.* The literal is recognized as a magic literal which means the heuristic is right.
- *False Positive.* The literal is not a magic literal, it is a false positive generated by the heuristic.
- *Unclassified.* It is not clear whether the literal should be considered as magic or not.

The results are the following. In these 100 methods, the heuristic detected 243 magic literals. From these 243 literals, we find out that 151 are true positives split across 68 methods (where 60 contain only true positives), 74 are false positives split across 28 methods (where 20 contain only false positives) and 18 could not be classified split across 14 methods (where 11 contain only unclassified literals).

These results suggest that the heuristic works quite well since it is able to reveal magic literals while being right about 62% of the time on this sample of 100 methods.

# 7 Discussion

In this section we discuss some aspects and limitation of the heuristic implementation.

## 7.1 Constant Values v.s. Missed Parameterization

Some magic literals represent *constant values*: they can never change. Examples of such constant are $\pi$, G (gravity constant), Euler number, etc. For these literals we do not want to enforce modularity, quite the contrary in fact. It would be error-prone to let developers override those constants. The objective when removing these magic literals is to make the code more readable and to avoid inserting typo in some occurrences of the constant in the source code. In Pharo, extracting these constants as class-variable in a SharedPool is the way to go.

On the other hand, there are magic literals that should be instance variables or method parameters. The developer *missed* the opportunity to make the code *parameterizable*. Those literals can be extracted into a method that return directly the literal or as a call to an accessor method for which the instance variable is set to the literal value by default (these two possibilities have been addressed in Section 5).

The difference between constant values and missed parameter literals can not be extracted from structural properties of the software. Thus, our code critic rule is not able to propose the correct extraction strategy to the developer.

## 7.2 Magic Strings

We developed a heuristic which aims to remove string literals that are not self-explanatory. This is acceptable if one considers that only readability of the source code is important. From the modularity point of view, the heuristic is wrong to not consider strings recognized as understandable. Indeed, these strings might still cause problem for future evolution of the software.

It is not clear how to solve this problem. Extracting all string literals to make them parameterizable would be the best from a modularity point of view. However, it will make the number of methods (and eventually instance variables) increase which is an overkill because most of these strings will probably never be changed and are not repeated multiple time. Further research is required to tackle this problem correctly.

## 7.3 Method Selector Explaining Argument Literal

We observed that, for literals appearing as method argument, it happens that the part of the method selector appearing just before the argument allows developer to understand it. A good example of such method selector is Date class»year:month:day:. Consider the code snippet in Listing 16, it is obvious that 2019 refer to the year 2019, 6 refer to June and 1 refer to the first day of the month. Still, from a modularity point of view the occurrence of these literals is not optimal.

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

**Listing 15.** Example of literals as method argument that are explained by the method seector.

Method names in most other programming languages do not describe multiple arguments as well as the names in Pharo. Consider, the same initialization written in Java. In this case, all three arguments can be considered magic literals:

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

**Listing 16.** Arguments of the same initialization are non-magic in Pharo, but magic in Java.

## 7.4 Scripting vs. Extensible Code

While scripting (i.e., writing specialized code that is usually not reusable) developers tend to introduce magic literals. Indeed, since a script should be one-shot, developers will not put effort in making it parameterisable.

The problem arise when a developer find out that the script he wrote can actually be reused. It is likely that developer copy-pastes the script code into a method and starts working on it from the script version. This behavior increases the occurrence of magic literals in the source code.

## 7.5 Performance Impact of Magic Literals Extraction

Depending on the extraction strategy chosen by a developer, the time for the execution of a method can increase. We identified 5 possible strategies:

1. Extract the literal into a class-variable (Listing 17).
2. Extract the literal into a method that directly returns it (Listing 18).
3. Extract the literal into a method that returns the value of an instance variable initialized previously with the literal (Listing 19).
4. Extract the literal into a method that returns the value of an instance variable that is initialized lazily if it is nil when read (Listing 20).
5. Extract the literal into an instance-variable initialized previously with the literal (Listing 21).

To compare these strategies, we implemented each of them and performed a simple benchmark. Details of the setup for this benchmark can be found in Appendix A.

Figure 3 shows the results of the benchmark for each strategy. *Strategies 1* and *5* are impacting the least time to run a method. Indeed, retrieving the value held by an instance / a class variable is fast since it only requires to read the variable's value.

*Strategies 2* and *3* are slower than Strategy 1. This can be explained by the fact that each time the value needs to be retrieved, a method lookup is triggered. The mechanism involved creates an overhead which impact method performances. However, it seem that there is no significant difference between benchmarks of strategies 2 and 3. This is interesting because it suggests that reading the value of an instance variable costs virtually nothing.

*Strategy 4* is the slowest which can be explained by the need to check whether the instance variable is nil or not each time the method is called. This create an overhead that is added to the time consumed by method lookup.

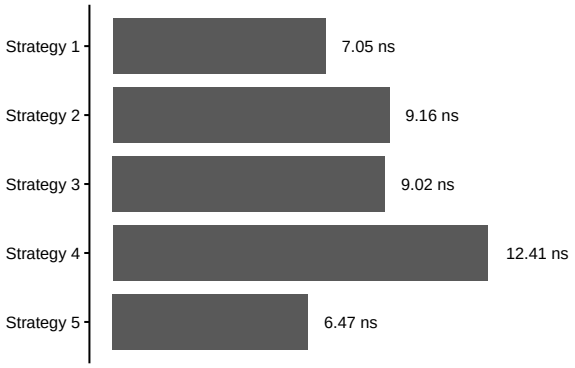


**Figure 3.** Average execution time of each strategy. Measured by benchmarking each strategy for 10 seconds

## 8 Related Works

To our knowledge, no similar study about magic literals in Pharo nor in Smalltalk exists in the literature. This article is thus probably the first to address this problem in the context of Smalltalk.

The concept of a magic number is discussed by Martin Fowler [FBB+99] and Robert C. Martin [Mar09] describing it as a bad practice that should be avoided. Both authors claim that magic numbers should be replaced with symbolic constant to ease understanding of software. However, the modularity problem caused by magic literals that should be parameterizable is not discussed. As we discuss in Section 1, in this article we choose to use the term "magic literal" instead of magic number.

Smit et. al. [SGHS11a, SGHS11b] identified the relative importance of 71 coding conventions to maintainability based on a survey of 7 software engineers. The authors analyzed the revisions of four different open-source projects and observed that when developers are conscious of conventions (via explicit coding conventions policy and checks made by continuous integration servers), they put effort to respect those convention. When developers are not conscious of conventions, violations are prevalent. One of these coding conventions is the usage of magic numbers. The definition of magic number used by the authors is only about number literals (which contradicts with the definition of Robert C. Martin [Mar09] cited previously). Furthermore, their definition considers -1, 0, 1 and 2 as non-magic. The results of the survey suggest that avoiding magic numbers occurrences in source code is considered as important by software engineers. However, in the analysis of the 4 open source projects, "avoiding magic numbers" coding convention appears three times as the third and once as the fourth most violated convention.

Nundhapana and Senivongse [NS18] discuss the approach taken by an IT organization in Thailand to enforce naming conventions in Objective C software. The authors developed a library to check for naming conventions automatically. In particular, magic numbers are considered as a violation of naming convention since they are unnamed literal constants. A regex-based checker is used to detect magic numbers in Objective C source code.

Both Smit et. al. [SGHS11a, SGHS11b] and Nundhapana et. al. [NS18] distinguish "magic numbers" from "literal strings" but "literal strings" are not the "string" equivalent of magic numbers. The coding violation is different since literal strings concern multiple occurrence of the same string literal inside a Java / Objective C file that should be extracted into a constant. The concept of "literal strings" does not deal with the self-explanatory property of the source code but rather with its modularity. Indeed, the occurrence of a string can be misleading for a developer reading the code (think of commit hash example in the introduction of this article).

The WikiWikiWeb [wik] has a very interesting page related to magic numbers. The content of this page is similar to the explanations we provide in this article while providing a less precise characterization of magic numbers. In particular, the authors provide three rules to determinate if a literal is a magic number:

1. *Strict Magic Number rule*: Literals should only appear on the right hand side of a constant declaration statement.
2. *Practical Magic Number rule*: A literal is a not a magic number if the most meaningful variable name for it is the same as the spoken name of the literal.
3. *ZeroOneInfinityRule*: The only constants that should appear without a name in a program are 0 and 1, and then only if they are used in integer arithmetic or comparisons.

We agree with these 3 rules and the work presented in this paper basically extend them. However, rule 2 is for developer and can probably not be handled by an automatic analysis.

## 9 Conclusion

In this paper, we explored the concept of magic literal generally and more specifically, in the context of Pharo. We did

an exploration of the distribution of literals in the system leading us to a characterization of acceptable literals and a definition of magic literals. We used the definition to implement a heuristic to identify magic literals as code critics rule. We manually evaluated a subset of the results produced by our heuristic and found that it identified correctly magic literals 62% of the time. To our knowledge, this contribution is the first heuristic to detect magic literal.

## 10 Future Work

The research we conduct in this article opens multiple perspectives. We want to dig deeper in the analysis of magic literals 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 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 occur 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.

***Acknowledgements.*** The authors thank Arolla (http://www.arolla.fr) for the funding of Oleksandr Zaitsev.

## References

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, Boston, MA, USA, 1998.

[ada] Rationale for the design of the ada programming language. http://archive.adaic.com/standards/83rat/html/ratl-02-01.html. Accessed: 2019-05-21.

[AKGA11] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190, March 2011.

[Bak97] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal of Computing*, October 1997.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[DMP16] 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.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE Computer Society, September 1999.

[Eng] english-words: A text file containing over 466k english words. https://github.com/dwyl/english-words. Accessed: 2019-07-24.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[KN01] Georges Golomingi Koni-N'sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Bern, June 2001.

[Koe95] Andrew Koenig. Patterns and antipatterns. *Journal of Object-Oriented Programming*, March 1995.

[Mar09] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[NS18] 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.

[pyt] The python language reference. https://docs.python.org/3.7/reference/lexical_analysis.html#literals. Accessed: 2019-05-21.

[SGHS11a] 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.

[SGHS11b] 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.

[SS18] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 2018.

[SSS14] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.

[TGN17] 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.

[VDDR07] 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.

[vDMvdBK01] 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.

[wik] Magic number on wikiwikiweb. http://wiki.c2.com/?MagicNumber. Accessed: 2019-06-19.

# A Setup to Benchmark Magic Literal Extraction Strategies

This appendix contains the experimental setup for the performance measurement discussed in Section 7.5.

```
Object subclass: #Strategy1
  instanceVariableNames: ''
  classVariableNames: 'MEANING_OF_LIFE'
  poolDictionaries: ''
  category: 'BenchmarkExtractionStrategies'

Strategy1 class » initialize
  MEANING_OF_LIFE := 42

Strategy1 » methodAccessingLit
  ^ MEANING_OF_LIFE
```

**Listing 17.** Implementation of strategy 1.

```
Object subclass: #Strategy2
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'BenchmarkExtractionStrategies'

Strategy2 » meaningOfLife
  ^ 42

Strategy2 » methodAccessingLit
  ^ self meaningOfLife
```

**Listing 18.** Implementation of strategy 2.

```
Object subclass: #Strategy3
  instanceVariableNames: 'meaningOfLife'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'BenchmarkExtractionStrategies'

Strategy3 » initialize
  super initialize.
  meaningOfLife := 42

Strategy3 » meaningOfLife
  ^ meaningOfLife

Strategy3 » methodAccessingLit
  ^ self meaningOfLife
```

**Listing 19.** Implementation of strategy 3.

```
Object subclass: #Strategy4
  instanceVariableNames: 'meaningOfLife'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'BenchmarkExtractionStrategies'

Strategy4 » meaningOfLife
  ^ meaningOfLife ifNil: [ meaningOfLife := 42 ]

Strategy4 » methodAccessingLit
  ^ self meaningOfLife
```

**Listing 20.** Implementation of strategy 4.

```
Object subclass: #Strategy5
  instanceVariableNames: 'meaningOfLife'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'BenchmarkExtractionStrategies'

Strategy5 » initialize
  super initialize.
  meaningOfLife := 42

Strategy5 » methodAccessingLit
  ^ meaningOfLife
```

**Listing 21.** Implementation of strategy 5.

```
|s1 s2 s3 s4 s5 benchs|
s1 := Strategy1 new.
s2 := Strategy2 new.
s3 := Strategy3 new.
s4 := Strategy4 new.
s5 := Strategy5 new.
benchs := {
  [ s1 methodAccessingLit ] benchFor: 10 seconds.
  [ s2 methodAccessingLit ] benchFor: 10 seconds.
  [ s3 methodAccessingLit ] benchFor: 10 seconds.
  [ s4 methodAccessingLit ] benchFor: 10 seconds.
  [ s5 methodAccessingLit ] benchFor: 10 seconds.
}
```

**Listing 22.** Code performing the benchmark.