What Do Developers Consider Magic literals? A Smalltalk Perspective

N. Anquetil, J. Delplanque, S. Ducasse

Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, France

O. Zaitsev

Arolla, France

C. Fuhrman

ÈTS Montreal, Canada

Y.-G. Guéhéneuc

Concordia University, Canada

Abstract

Context: 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. Relatively little literature can be found on the subject beyond the usual (and very old) recommendation of avoiding literals and preferring named constants. Yet, magic literals are still routinely found in source code. **Objective:** We studied literal values in source code to understand when they should be considered magic or not (i.e., acceptable). **Methods:** First, we perform a *qualitative* study of magic literals, to establish why and under which conditions they are considered harmful. We formalize hypotheses about the reasoning behind how literals are considered magic. Second, we perform a *quantitative* study on seven real systems ranging from small (a few classes) to large (thousands of 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). Third, we report on another study involving 26 programmers who analyzed about 24,000 literals, to understand which ones they consider magic. Finally, we evaluate the *hypotheses* defining specific conditions under which literals are acceptable. **Results:** We show that (1) literals still exist and are relatively frequent (found in 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) the hypotheses concerning acceptable literals are valid to various degrees. **Conclusion:** 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.

Keywords: Software analysis, Software Quality, Magic literals

1. Introduction

Literals are constant values in source code, and magic literals are constant values whose meaning might not be obvious to the reader of the code. They have long been considered a type of code smell, as they may hinder source-code comprehension, negatively impacting maintenance [3, p.166][14][15]. When the same literal is used in several places, there is also a risk that a software change fails to modify all the instances of the literal, thus resulting in a bug.

Yet, despite the very old recommendation of avoiding literals and preferring named constants [6, p.300], literals and magic literals are still easily found in source code. Smit et al. [14] reported that magic literals were common in the systems they studied.

In this paper, we contrast *Magic* literals, that would best be replaced by a named constant, with *Acceptable* ones, that may remain as is in the source code. We therefore propose that if one cannot eradicate literals in the source, 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* or not (i.e., *Magic*). This is difficult because the same value may have different meanings in different contexts, and be *Magic* in one and *Acceptable* in another. For example, "2" in the statement "`circumference := radius * pi * 2`" is not a magic literal [6, p.300]. In this context, it is self-explanatory and using a named constant (ex: "`DUE`[1]") would be detrimental to understanding. On the other hand, the same "2" in the statement "`r1 := ((depth - 2) * arcWidth) + off`" was considered *Magic* by one of the participants of our study.

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

- First, we study the manifestation of the "problem" with a *qualitative* study of literals. Why are they detrimental and under what conditions could they be acceptable? We deduce and formalize $5 + 1$ hypotheses to reason about acceptable literals.

- Second, we perform a *quantitative* study on seven real systems ranging from small (four classes) to large (7 700 classes) and covering different domains. We report the literals' types (number, string, Boolean, ...), their grammatical function (e.g., argument in a call, operand in an expression,

[1]"two" in Italian

value assigned, ...), or the purpose of the code in which they appear (test methods, regular code).

- Third, we study when a literal might be considered *Magic* or *Acceptable* with the help of 26 developers who analyzed about 24,000 literals, from more than 3,500 methods.
- Fourth, we formalize and validate $5 + 1$ hypotheses regarding how literals are considered *Magic* or *Acceptable*.

Results include:

- The confirmation that literals are frequent in source code. Almost 50% of all the methods in the seven systems (and up to 73% for one system) contained at least one literal. And they are more frequent in test methods, with 80% of them containing at least one literal;
- Most of these literals (close to 75%) were considered *Acceptable*;
- Our proposed hypotheses about *Magic* and *Acceptable* literals proved to be valid (giving results that were statistically significant with a χ^2 tests), although to varying degrees of practical significance.

The rest of the paper is organized as follows: Since the study presented is based on Pharo[2] systems, and Pharo has a syntax originating from Smalltalk, Section 2 gives an overview of this syntax and discusses its impact on the study. Section 3 describes our qualitative study of (magic) literals and formalizes the hypotheses about acceptable ones. In Section 4 we present the seven systems used for our *quantitative* study, we give numbers on the prevalence of literals for each of these systems, and we also test the validity of our hypotheses about the reasoning of how acceptable literals are distinguished from magic ones. Section 6 discusses properties and limitations of the studies. Section 7 discusses research related to the topic of this paper. Section 8 concludes and proposes possible extensions and future work.

2. Background

We used the Pharo system, projects, and libraries to perform our qualitative and quantitative analyses of (magic) literals. In the rest of this paper, several snippets of Pharo code also serve as examples.

Therefore, in this section, we first recall the Pharo syntax, then we give the list of Pharo's literals that we extended slightly for the purpose of this study. We also explain how named constants are defined in Pharo.

Although Pharo might not be the most popular programming language, we chose it for several reasons:

[2] `pharo.org`

Expertise: This paper is an extension of a preliminary study [2] itself done with Pharo;

Applicability: We hope to use the results of the study to improve the Pharo development environment to actively support developers in avoiding *Magic* literals;

Convenience: We had the possibility to enlist several experienced Pharo developers in our study and classify literals as *Magic* or *Acceptable* for us;

Tooling: Pharo includes various tools and a reflexive API that made the extraction of the data (literals) straightforward.

Throughout the paper we strive to make the discussion of results more generic. When meaningful, we compare Pharo's syntax or conventions to that of other programming languages and we adopted conventions that better match other programming languages. For example, although nil is not a literal in Pharo (see below) we did consider it as such in our study to match the conventions in languages like C, Java, or others.

2.1. General Pharo Syntax

Pharo syntax, following Smalltalk's tradition, is compact and fully fits on a postcard[3]. It is composed of literals (boolean values, numbers, strings, symbols, literal arrays, nil), variable definitions, assignments, returns, method definitions, method annotations, lexical closures, and messages. For example, there are no flow control instructions. Loops and conditionals are expressed with messages sent to boolean objects (e.g., methods "ifTrue:"[4] or "to:do:"[5]). Messages are discussed below but have the same meaning as in other object-oriented languages.

```
CardGame >> distributeCards
  "Returns four random cards (numbered from 1 to 52)"
  <menu: 'start'>
  | cards |
  cards := (1 to: 52) asOrderedCollection.
  ^ (1 to: 4) collect: [ :i | cards remove: (cards size atRandom) ]
```

Listing 1: Pharo code example.

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

[3]`https://richardeng.medium.com/syntax-on-a-post-card-cb6d85fabf88`

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

[5]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 number in the sequence

- Line 1: Method name – The actual method starts at its name (distributeCards). What comes before here is a convention to specify in which class it is implemented: CardGame;
- Line 2: Comment – Enclosed in double quotes, not to be mistaken for a string (see below);
- Line 3: Pragma – Enclosed in "<" and ">", here the pragma (annotation) is <menu:> and it has one argument ('start');
- Line 3: String – Enclosed in simple quote, here 'start';
- Line 4: Local variable declaration – Enclosed in "|". Variables are not statically typed, so only their name is declared, here cards;
- Line 5: Assignment – Denoted by := ;
- Line 5: Message – There is no dot before a message sent (as in other languages) but just a space. Here the message "to:" is sent to the object "1" with argument "52", the result of this call is an object to which the message "asOrderedCollection" is sent;
- Line 5: Statement separator – Denoted by a dot;
- Line 6: Message – Again the message "to:" is sent to the object "1" with argument "4" this time, and the message "collect:" is sent to the result with argument the lexical closure (see below)
- Line 6: Return – Denoted by ˆ (caret symbol);
- Line 6: Lexical closure – Enclosed in square brackets, ":i" denotes a parameter of the closure, and the code after "|" is the body of the closure.

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

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

In Pharo, developers tend to write short methods. Zaitsev et al. reported that the mean method length was only 5.7 lines and the median was 3 [17]. This could have an impact on our study, since shorter methods would be easier to understand and would contain fewer literals.

2.2. Definitions of literals and their use

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"* [4, p18–19]. This definition is similar to definitions used by other programming languages.

In our study, 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. As explained above, these three specific objects were added to the list of literals to better match the conditions of other programming languages.

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.

Named constants (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 to this convention in the next section (Listing 4, page 9).

3. Qualitative Study

In the literature, authors may call any kind of undocumented literal a *magic number*. Martin 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" [6, p.300]. 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.

We now discuss what makes literals *Magic* (Section 3.1) or *Acceptable* (Section 3.2). These definitions are expressed as 5 + 1 formal hypotheses that will be tested later on real cases.

3.1. Magic Literals

Usage of literals can detract from code quality, for example by reducing understandability, increasing logic duplication, or reducing modularity (a missed opportunity for customization).

Understandability. Martin defines magic literal as "any token that has a value that is *not self-describing*" (our emphasis) [6, p.300]. The usage of magic literals hinders the understanding of program logic. Developers (should) usually attempt to have code as self-explanatory as possible to reduce the need for documentation, and, in turn, to reduce the need for maintaining this documentation.

Magic literals can make source code less straightforward, because these literals have "hidden meanings" that cannot be extracted from the literals alone. They require additional knowledge that, if not documented, is difficult to obtain.

We may therefore hypothesize that the familiarity (experience) of developers in a system or a domain should influence what literals they consider *Magic*.

Hyp1: *Novices in a system consider literals as* Magic *more often than experts.*

Listing 2 shows a method with a detailed comment about its purpose and returned value. However, this method is difficult to understand because of its magic literals. Line 10 is shifting the bits of an integer (returned by the call to format) 16 times to the right, which is used to discard the 16 least significant bits. Then the bit-wise "bitAnd:" operation is applied to the result and the 16r1F literal.

```
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 et al)
6    1 = non−indexable objects with inst vars (Point et al)
7    [...]
8    24–31 = compiled methods (CompiledMethod)"
9
10   ^(self format bitShift: −16) bitAnd: 16r1F
```

Listing 2: Example of magic literals used for bit operations.

The purpose of the bit-wise "bitAnd:" is not explicit for anyone not familiar with base-16. The literal 1F in base 16 is 11111 in base 2, and the "bitAnd:" will therefore extract the 5 least significant bits from the receiver. Still, we do not know why these specific bits are extracted and there is no documentation (such as comments) to explain it.

Thus, using a base-2 representation could improve understandability. Another improvement would be to create "named constants" which would be done in Pharo by extracting the values 16 and 16r1F to be returned by methods with useful names such as instSpecOffset and instSpecMask, respectively.

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

For example, Listing 3 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”.

```
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 3: Example of a method with multiple occurrences of the magic literal 1024.

Note that modern development environments may offer better handling of literal repetition by offering refactoring operations that will modify all occurrences of a given literal. This refactoring, however, will not take care of literals with a related but different value (e.g., “1024”, “1023”, and “256”.) These will still need to be handled manually.

Modularity. Magic literals reduce the modularity and reusability of methods in which they occur. The occurrence of a magic literal freezes its value in the source code, preventing subclasses or client code to adapt the method to their need. By creating a named constant, one stabilizes the part of the code needing the value, while allowing the value to be changed easily.

This raises the issue of how constant a “constant” is. For example, the number pi is widely seen as an absolute constant. There is no need to worry about understandability and modularity here, although redundancy could be an issue [6, p.300].

But other values, assumed constants, might depend on circumstances. For example, gravity on Earth is cited as a constant by Fowler [3, p.166], but it is subject to variations[6] that might be important for some. It does not seem too big a stretch to imagine a physics program reusing a generic library where Fowler’s GRAVITATIONAL_CONSTANT is defined, and later the author of the program realizing he needs a more flexible definition of the “constant” and consequently having to redefine all library functions/methods using this value.

[6]https://en.wikipedia.org/wiki/Gravity_of_Earth

The Java convention to define named constants is to use public static final. This solves the understandability and duplication issues, but not the modularity one – a final variable may not be reassigned. It is, of course, possible to define a constant as non-final in Java, but this is not the accepted convention.

The convention in Pharo is to use a method returning a literal (e.g., Listing 4). This tackles all three issues, as the name of the method acts as the name of a Java constant, whereas the method can be overridden in subclasses to adapt its value.

```
earthGravity
  ^ 9.81
```

Listing 4: Example of how Pharo defines a named constant to improve modularity.

These three dimensions of code quality are orthogonal to the concept of *Magic* vs. *Acceptable* literals and will not be further discussed in this paper.

3.2. Acceptable Literals

Authors recognize that the presence of literals in source code is not a code smell *per se* [3, 6]. 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.

From the discussion in previously cited work (prominently [3, 6]) and the experience of the authors, we propose a list of *Acceptable* literal cases. We associate hypotheses with each of these categories, formalized here and tested in Section 4.4 on data from real projects. The cases of *Acceptable* literals may not be complete and other cases might be proposed, for example specific to other programming languages. If **Hyp1** holds, one should expect that other rules could be identified in specific application domains or for specific systems. Whether this effort would be worthwhile would depend on the domain or the system.

Self-describing literals. Some literals directly refer to the data that they hold. For example, the Booleans (true and false), the empty pointer (nil or null), the empty string, and in Smalltalk the empty arrays (#() and {}). Those literals are acceptable because the semantics is obvious upon reading.

Using *string* literals in the source code might also be legitimate, particularly if the target audience is end users. Developers understand that special efforts must be put in to allow end users to understand the messages of a program. If a constant is in natural language, then it is self-explanatory and therefore *Acceptable*.

Note, however, that if the target audience is developers (as for strings used as dictionary keys) it is akin to normal identifiers and therefore subject to the same shortcomings (too short, cryptic, unrelated). Dealing with this difficulty is outside the scope of this paper. Maybe some artificial intelligence or natural language processing solution could help deciding whether a string is actually self-describing?

Internationalization is another special case that we do not consider. Even if source code contains a string in some foreign language, we assume the developer understands it.

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

Listing 5: Example of iconNamed: method call.

Pharo has *symbol* literals (nonexistent in Java, see Section 2.2). We treat symbols as strings and also consider them *Acceptable* because they have a readable form, contrary to numbers that do not convey semantic information beyond their actual values. Listing 5 shows a method using the symbol #smallWarningIcon, which is explicit.

We hypothesize that "self-describing" literals are *Acceptable*.

Hyp2: *Self-describing literals are less often magic than other literals. Literals considered self-describing in this paper are Booleans, empty arrays, empty pointer, strings, and symbols.*

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; C considers 0 as false and any other value is true; C again denotes the end of strings with the character of ASCII code 0; etc.

Such literals are acceptable because the programming languages explicitly expect developers to use them. This assumption is also supported by Smit et al. [14] whose Java-centered study considered -1, 0, 1, and 2 as non-magic. Furthermore, these conventions are described in the language's specifications, and it is very unlikely that they change.

Pharo also has such conventions. They are often contextual, because even flow control operations are not part of the language syntax but are messages like any others. Thus to loop over an array, one would use "1 to: anArray size do:..." which sends the message to:do: to "1", the starting index of arrays ([4, p.21]). Similarly in Listing 6, copyFrom:to: is sent to an instance of a string, which is a SequenceableCollection (of Characters), to copy it from the first argument index to the last. Collections being indexed from 1 (as arrays), the literal "1" is often used as first argument of this message.

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

Listing 6: Example of literals as arguments.

Table 1: Literals ignored in the context of certain message because they follow known code conventions.

Selector	Literal	Role
to:	1	Receiver
to:by:	1	Receiver
to:do:	1	Receiver
+	1	1st arg.
-	1	1st arg.
nextPut:	Character	1st arg.
nextPutAll:	String	1st arg.
«	Character	1st arg.
«	String	1st arg.
name:	String	1st arg.

Table 1 lists the contexts (messages sent) we identified in Pharo where literals are acceptable because they fall in the *Coding Convention* category. In the table, the "Selector" column is the name of the message sent. The "Role" column describes where the literal should appear in the message sent (receiver of the message or argument). The "Literal" column describes the acceptable literal. The first three messages implement a *for* loop in Pharo and the receiver is the starting value of the loop (1 being a common value). The two arithmetic operators (+ and -) with an argument (second operand) of 1 correspond to the increment (resp. decrement) operator of other languages. The messages nextPut:, nextPutAll:, and « are used to append characters or strings to output streams (similar to the « on ostream in C++); they take as (first) argument the character or string to append to the stream. The last message (name:) is a setter for a name instance variable that is commonly found in classes; the (first) argument is a string denoting the name to give.

The list comes off the experience of the authors as Pharo programmers and from an early analysis of a small set of literals (100 literals, [2]). Defining such a list is a trade-off between recall (finding all *Acceptable* literals) and precision (finding only *Acceptable* literals). In specific contexts, other messages could be added to the list (see Section 6). We did not strive to reach the best possible list for the systems considered, but wished mainly to evaluate that such a list could make sense and give good results. The results are discussed in Section 4.4; they show that the list could be improved.

We hypothesize that all the usages listed in Table 1 represent legitimate cases of literals in the source code.

Hyp3: *Literals matching one of the contexts described in Table 1 are less often magic than others (context here includes the method sent and the specific literal in a specific role, e.g., receiver, first argument).*

Named Literals. As mentioned in Section 3.2, Java represents named constants as public static final variables to which a literal value is assigned. More generally, a literal directly assigned to any variable (e.g., Listing 7) may also be considered acceptable as the variable name provides the semantics.

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

Listing 7: Example of literal assignments.

In Pharo, public static final variables do not exist, but named constants are often implemented through simple methods only returning a value (Listing 4) allowing developers to access them explicitly.

Furthermore, McConnell [7] referred to named constants as a lower-level use of information hiding in design. We therefore hypothesize that 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 or method names.

Hyp4: *Literals directly assigned to a variable or directly returned by a method are less often magic than other literals.*

Literals in Method Annotations. Pragmas in Pharo and annotations in Java act as a tag attached to methods or classes. They may accept parameters.

Annotations and pragmas can be parameterized with arguments (e.g., in Java "@Test(timeout=1000)", or the same in Pharo "<timeout=5>", see Listing 8). These arguments can only be literals because the annotations or pragmas are interpreted during parsing and are static by nature. We give examples of a string argument to a pragma in Listing 1 and an integer argument in Listing 8. Thus, any literal that is an argument to a pragma is considered *Acceptable* because the language imposes it to be a literal – named constants are not allowed there. Note that although literals are mandatory in this case, programmers considered them magic in some cases (see sections 4.4 and 6.4).

Hyp5: *Literals that are arguments of a pragma (annotation) are less often magic than other literals.*

```
IntegerTest >> testNthRootTruncated
  <timeout: 5>
  | tooBigToBeAFloat |
  tooBigToBeAFloat := 1 << 2000.
  self assert: (tooBigToBeAFloat nthRootTruncated: 100) equals: 1 << 20.
  ...
```

Listing 8: Example of "<timeout:>" pragma usage with an argument.

Literals in Test and Example Methods. Literals may appear in test and example code [16]. They are usually acceptable for the following reasons:

1. Tests or examples should stay as simple as possible. Having to create extra entities (constant variables of methods) 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 or examples are used to build instances of demonstrative objects only used locally and transiently.

The *Literal Value* pattern [8] is a common way to specify the values of attributes of elements in a test. According to this pattern, they can be *self-describing values*, e.g., `'Irrelevant product name'` for a string argument in an object constructor used in a test method. However, literals can sometimes be code smells in tests. If they are repeated, then they should be replaced with a symbolic constant to reduce *test-code duplication.*

Example methods in Pharo are denoted with the `<example>` pragma. This does not fall in the previous hypothesis case, as we are talking here of the literals in the body of a method (and the `example` pragma has no argument).

Hyp6: *Literals in test methods or example methods (i.e., having the "*`example`*" pragma) are less often magic than other literals.*

4. Quantitative Study on the Prevalence of Literals

Now that we have defined and discussed magic literals qualitatively, we want to assess the prevalence of literals, magic or not, in the source code of different systems. First, we present the systems studied in this experiment, and descriptive statistics on the presence of literals (Section 4.1). Second, we perform a qualitative study to collect and report the prevalence of literals (Section 4.2). Third, we perform a manual validation of these literals, with external developers, to report the prevalence of magic literals (Section 4.3). Last, we validate the hypotheses of the preceding section (Section 4.4).

4.1. Subject Systems

We selected seven systems or libraries on which to perform our quantitative and qualitative study: *Morphic*, *Parser*, *Pharo*, *Polymath*, *Roassal*, *Seaside*, and *VMMaker* (see Appendix A for version and repository details). The systems were chosen from different application domains and we would have several experts or knowledgeable participants for classifying the literals in these systems. For some of these systems, we had expectations on what kind of literals they would contain (e.g., *Polymath* should contain numerical literals) and we aimed to cover different types of literals (character, string, numerical).

***Morphic*:** The graphical user-interface library of Pharo.

***Parser*:** A small parser and lexical scanner library. We expect it to contain many character literals. It does not appear much in the discussions because of its small size.

***Pharo*:** A Smalltalk-like programming language as well as a development environment. Pharo has heterogeneous domains of application, as it includes a programming language, a software development environment, process handling and multitasking, streams and input-output, etc. *Morphic* and *Parser* are usually considered a part of *Pharo*; here we explicitly excluded them to avoid overlaps.

***Polymath*:** A mathematical library to represent mathematical abstractions such as Fourier transforms, random number and statistical-distribution generators. We expect it to contain many numerical (integer, float) literals.

***Roassal*:** A visualization engine to draw graphs or graphics of data and source code. Because of the layout and drawing of graphs, we expect it to contain numerical literals, maybe more integers than floats.

***Seaside*:** A server-side framework for developing and running Web applications. Because of the HTML code generation, we expect it to contain many string literals.

***VMMaker*:** (also known as OpenSmalltalk-VM) A generator of the Smalltalk virtual machine (in C) from a description in a higher abstraction-level language. Because of the C code generation, we expect it to contain many string literals.

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 2 (first two columns).

Table 2 provides descriptive statistics of the systems studied including the literals they contain. See Appendix B for the details of how the data was extracted using the reflective API of Pharo.

Note that the number of classes is only used as indication of the size of the systems. There may be an overlap between regular and "test" classes as an otherwise regular class may contain an example method, or an otherwise test class can contain a regular method (e.g., a utility method to do some processing). For methods, there is no such overlapping and the distinction is clear.

Overall, a significant proportion of methods contain literals (regular methods=42.9%, test or example methods=82.2%). This confirms that literals are prevalent and *Magic* literals could be an issue.

One system, *VMMaker*, stands out because it has the highest proportion of regular methods with literals (72.5%) and the highest concentration of literals in regular methods with literals (5.9).

Tests have much higher proportions of methods with literals (82.2%) and also higher densities of literals per method (5.8) compared to 3.5 for regular methods.

Table 2: 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).

	#class	#method	meth. w/ lit.	#literal	lit./meth.
Regular methods (i.e. excluding tests and examples)					
Morphic	363	8 668	3 660 (42.2 %)	9 877	2.7
Parser	6	140	76 (54.3 %)	241	3.2
Pharo	7 190	80 534	29 851 (37.1 %)	81 307	2.7
Polymath	227	1 969	945 (48.0 %)	3 238	3.4
Roassal	867	9 783	4 313 (44.1 %)	16 361	3.8
Seaside	626	4 730	2 063 (43.6 %)	4 611	2.2
VMMaker	328	15 012	10 887 (72.5 %)	64 641	5.9
overall	9 607	120 836	51 795 (42.9 %)	180 276	3.5
Test or example methods only					
Morphic	26	123	89 (72.4 %)	485	5.4
Parser	3	65	56 (86.2 %)	248	4.4
Pharo	1 408	14 133	11 450 (81.0 %)	62 658	5.5
Polymath	97	813	754 (92.7 %)	6 709	8.9
Roassal	28	169	144 (85.2 %)	978	6.8
Seaside	96	607	564 (92.9 %)	3 583	6.4
VMMaker	28	232	218 (94.0 %)	2 219	10.2
overall	1 686	16 142	13 275 (82.2 %)	76 880	5.8

This observation seems natural as tests must compare computed values to some explicit standards.

Again, *VMMaker* stands out as the system with the highest densities of literals per test method, 10.2. *Seaside* and *Polymath* also come out as systems with a high proportion of test methods containing literals, 92.9% and 92.7%, respectively.

Considering the proportions of literals found in tests, there are two groups. On the one hand, *Parser*, *Pharo*, *Polymath*, and *Seaside* have close numbers of literals in regular methods and test methods, even though they may have much fewer test methods than regular ones (e.g., *Pharo*, 7,190 regular methods with 81,307 literals and 1,408 test methods with 62,658 literals). On the other hand, *Morphic*, *Roassal*, and *VMMaker* have much fewer literals in tests. They are also the systems with comparatively fewer test methods.

However, the relation between the numbers of test methods and of test literals is not direct. For example, one third $(813/(813+1,969) \simeq 29\%)$ of *Polymath* methods are tests, but they contain two thirds $(6,709/(6,709+3,238) \simeq 67\%)$ of all the literals. Similarly, 15% of *Pharo* methods are tests $(14,133/(14,133+80,534))$, but they contain 44% of its literals $(62,658/(62,658+81,307))$.

Table 3: 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).

	Int.	*Strng*	*Symb.*	*Bool.*	*Arr.*	*nil*	*Float*	*Char.*
Morphic	4 385	827	2 314	1 316	310	815	330	65
	42.3%	8.0%	22.3%	12.7%	3.0%	7.9%	3.2%	0.6%
Parser	96	196	74	25	33	10	1	54
	19.6%	40.1%	15.1%	5.1%	6.7%	2.0%	0.2%	11.0%
Pharo	52 390	37 834	24 307	11 505	7 679	5 095	2 021	3 134
	36.4%	26.3%	16.9%	8.0%	5.3%	3.5%	1.4%	2.2%
Polymath	6 864	414	120	138	820	89	1 457	45
	69.0%	4.2%	1.2%	1.4%	8.2%	0.9%	14.6%	0.5%
Roassal	8 738	3 442	2 113	560	790	323	1 227	146
	50.4%	19.9%	12.2%	3.2%	4.6%	1.9%	7.1%	0.8%
Seaside	1 629	4 598	650	481	305	293	56	182
	19.9%	56.1%	7.9%	5.9%	3.7%	3.6%	0.7%	2.2%
VMMaker	37 271	7 441	12 102	7 086	609	1 496	222	633
	55.7%	11.1%	18.1%	10.6%	0.9%	2.2%	0.3%	0.9%
all	111 373	54 752	41 680	21 111	10 546	8 121	5 314	4 259
	43.3%	21.3%	16.2%	8.2%	4.1%	3.2%	2.1%	1.7%

4.2. Prevalence of All Literals

Table 3 provides the numbers and percentages of each type of literal for the seven studied systems and overall.

Except in the case of *Seaside* and *Parser*, *Integer* is the most frequent type. We believe this is due to the ubiquitous nature of integers. They are needed for domain values just like *Strings* or *Floats*, but they are also required for algorithmic-like looping over arrays (see discussion of **Hyp3** in Section 3.2).

Seaside and *Parser* have more *string* literals (56.1%, 40.1%) than *integer* literals (19.9%, 19.6%), because string literals describe web content (HTML elements, MIME types, URIs, etc.) and parsing error messages respectively.

String and *Symbol* are the second and third most frequent types. The prevalence of *symbols* (18.1%) over *strings* (11.1%) in *VMMaker* is possibly due to its nature as a system performing batch computations with little to no user input.

Polymath differs from the other systems with very few *string* (4.2%) or *symbol* (1.2%) literals, but much more *float* (14.6%) and *array* (8.2%) literals. This clearly comes from its application domain.

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

4.3. Prevalence of Magic Literals

We asked 26 developers (including five of the authors) to study and characterize literals as magic or not. The participants can be described as follows:

- Level of education was high, with only three not having completed a master in computer science, 14 having a master (or equivalent) including height of them doing a PhD at the time, and nine participants already having a PhD;
- Experience in computer science ranges from "none" (only courses) for four, between 1 and 5 years for four, between 6 and 10 years for seven; between 11 and 20 years for five; and six with over 20 years;
- Most of them (23) were working in a research environment, either as a student (intern, PhD student), professor, or research engineer in a research institute. The three others were developers in companies;
- Most of them (21) were working for public organizations at the time, with 10 of these having previous experience in working for private companies;
- Level of expertise in Smalltalk was high among them, with all but two having several years of experience programming in Smalltalk as their main programming language. Four of the participants are among the lead contributors of Pharo.

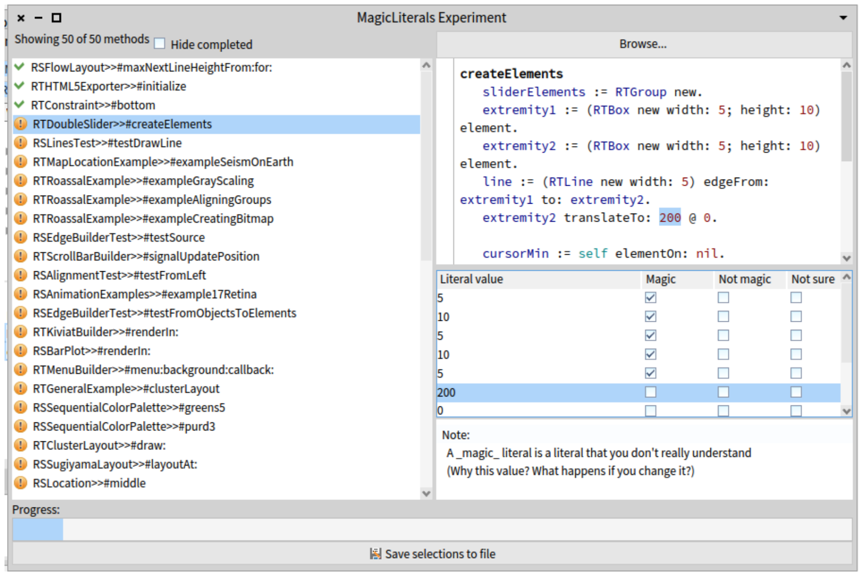
For each system studied, we collected all the methods having literals and grouped them in batches of 50 methods. Thus a batch contained at least but usually more than 50 literals (more than one literal per method). For each batch, the methods were chosen randomly. We forced a minimal level of redundancy by selecting 10% of a batch (5 methods) among methods pertaining to previous batches. Having the same literal evaluated by more than one participant allows to have different opinions on it to test the bias that experience in the system could introduce (see **Hyp1**). We also ensured that at least 10% of a batch were test methods. Thus test methods are overrepresented to ensure that we had enough data to analyze them separately (e.g., to validate **Hyp6**).

The result of this classification work is available as a replication package at `https://doi.org/10.5281/zenodo.5818035`.

A shared document listed all available batches, identified by the name of the subject system and a sequential number (e.g., "Pharo-12"). The developers were asked to choose one batch and put their name in front of it to mark it as done[7]. Having different participants evaluate systems that they might not know well allows us to test **Hyp1**.

For the study itself, a small tool (see Figure 1) presented the methods one at a time (figure, left) with the source code of the method (figure, top right), and

[7]Participants only knew who evaluated what batch, they did not have access to each other's evaluations



- ✓ RSFlowLayout>>#maxNextLineHeightFrom:for:
- ✓ RTHTML5Exporter>>#initialize
- ✓ RTConstraint>>#bottom
- ! RTDoubleSlider>>#createElements
- ! RSLinesTest>>#testDrawLine
- ! RTMapLocationExample>>#exampleSeismOnEarth
- ! RTRoassalExample>>#exampleGrayScaling
- ! RTRoassalExample>>#exampleAligningGroups
- ! RTRoassalExample>>#exampleCreatingBitmap
- ! RSEdgeBuilderTest>>#testSource
- ! RTScrollBarBuilder>>#signalUpdatePosition
- ! RSAlignmentTest>>#testFromLeft
- ! RSAnimationExamples>>#example17Retina
- ! RSEdgeBuilderTest>>#testFromObjectsToElements
- ! RTKiviatBuilder>>#renderIn:
- ! RSBarPlot>>#renderIn:
- ! RTMenuBuilder>>#menu:background:callback:
- ! RTGeneralExample>>#clusterLayout
- ! RSSequentialColorPalette>>#greens5
- ! RSSequentialColorPalette>>#purd3
- ! RTClusterLayout>>#draw:
- ! RSSugiyamaLayout>>#layoutAt:
- ! RSLocation>>#middle

```
createElements
   sliderElements := RTGroup new.
   extremity1 := (RTBox new width: 5; height: 10)
element.
   extremity2 := (RTBox new width: 5; height: 10)
element.
   line := (RTLine new width: 5) edgeFrom:
extremity1 to: extremity2.
   extremity2 translateTo: 200 @ 0.

   cursorMin := self elementOn: nil.
```

Literal value	Magic	Not magic	Not sure
5	✓	☐	☐
10	✓	☐	☐
5	✓	☐	☐
10	✓	☐	☐
5	✓	☐	☐
200	☐	☐	☐
0	☐	☐	☐

Note:
A _magic_ literal is a literal that you don't really understand
(Why this value? What happens if you change it?)

Figure 1: The small interface to evaluate a batch of methods with literals. On the left the list of methods, the first three already analyzed. On the top right, the current method with one literal ("200") highlighted. The "Browse" button allows seeing this method in a normal system browser for its class. On middle right, the list of literals for the current method with the three choices for each literal. At the bottom a progress bar for this batch and a button to save the result in a file.

the list of its literals (figure, middle right). The participant had to classify each literal as *Magic*, *Acceptable*, or *Undecided* (even though check boxes appear in the interface, it was not possible to check more than one). The participant had access to the entire Pharo environment if needed (figure, "Browse" button on the top right) to study the method in its class, related methods, etc.

Participation was entirely voluntary and participants were assured that their answers would not be divulged along with their names. They could choose to work on any system(s) by marking the batches as treated in the shared document. We only asked them not to concentrate all on the same systems, to keep a balance on the number of literals evaluated per system.

The study considered $3,857$ $(2,818 + 1,039)$ methods (5.9% of the 65,070 total methods with literals). From these, 1,039 were *test* methods (7.8% of the 13,275 *test* methods with literals).

Table 4 shows that 24,207 literals in total were evaluated $(13,705 + 10,502)$. Because there is some imposed evaluation redundancy in the methods (see above) the same literal (at a specific place in the source code) could be classified several times. To evaluate this point, we give the number of *source literals* (i.e.,

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

	#class	#meth.	#lit.	#srce lit.	lit. redun.	lit./meth.
Regular methods (i.e., excluding tests and examples)						
Morphic	64	174	471	463	1.7%	2.7
Parser	6	76	241	241	0.0%	3.2
Pharo	624	793	3 394	2 948	15.1%	4.3
Polymath	126	357	1 527	1 253	21.9%	4.3
Roassal	254	522	3 145	2 430	29.4%	6.0
Seaside	231	455	1 502	1 400	7.3%	3.3
VMMaker	131	441	3 425	3 210	6.7%	7.8
overall	1 436	2 818	13 705	11 945	14.7%	4.9
Test and example methods only						
Morphic	11	20	73	73	0.0%	3.7
Parser	3	56	248	248	0.0%	4.4
Pharo	235	344	2 716	2 181	24.5%	7.9
Polymath	73	287	3 549	2 742	29.4%	12.4
Roassal	29	85	963	682	41.2%	11.3
Seaside	58	193	1 985	1 603	23.8%	10.3
VMMaker	17	54	968	684	41.5%	17.9
overall	426	1 039	10 502	8 213	27.9%	10.1

the distinct literals in the source code). For regular methods, there are 11,945 source literals which gives 14.7% redundancy[8]: $(13,705 - 11,945)/11,945$. The fact that there is much more redundancy for tests (27.9%) is probably due to the larger numbers of literals per test method. *Morphic* has 0% redundancy for test methods because very few of them (only 20) ended up being classified. *Parser* has 0% redundancy because it was evaluated by only one participant.

The number of literals per method is almost always larger in this study (regular+test methods: 6.3) than for all the systems (regular+test methods: 3.9), see Table 2. We could not find an explanation for this fact.

Table 5 summarizes the results of the participants' analyses. Here are the important points:

- 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%. Note these numbers are not show directly in the table.

[8]redundancy $= \frac{\#duplicate}{\#source} = \frac{total - \#source}{\#source}$

Table 5: Percentages of magic literals for each system and for each literal type. In parentheses number of literals that were classified (to compare to the total number of literals in Table 3).

	Int.	*Strng*	*Symb.*	*Bool.*	*Arr.*	*nil*	*Float*	*Char.*
Morphic	18.6%	0.0%	2.0%	0.0%	7.7%	4.8%	17.2%	0.0%
	(242)	(36)	(99)	(56)	(26)	(42)	(29)	(14)
Parser	21.9%	0.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	(96)	(196)	(74)	(25)	(33)	(10)	(1)	(54)
Pharo	27.0%	6.5%	6.1%	5.6%	14.7%	13.3%	25.3%	38.1%
	(3 239)	(874)	(792)	(234)	(441)	(263)	(91)	(176)
Polymath	35.5%	12.8%	4.4%	23.9%	33.3%	36.5%	51.9%	9.1%
	(3 469)	(156)	(68)	(46)	(330)	(52)	(933)	(22)
Roassal	27.0%	26.2%	18.6%	17.6%	26.9%	21.1%	27.3%	0.0%
	(2 764)	(351)	(269)	(91)	(219)	(95)	(311)	(8)
Seaside	18.4%	9.9%	7.0%	3.1%	7.9%	7.2%	6.3%	9.8%
	(1 315)	(1 231)	(171)	(159)	(214)	(222)	(32)	(143)
VMMaker	43.7%	10.5%	2.2%	14.0%	36.4%	23.7%	33.3%	21.4%
	(2 929)	(408)	(536)	(321)	(55)	(76)	(12)	(56)
all	31.6%	10.3%	6.3%	9.7%	20.7%	14.5%	42.8%	20.1%
	(14 054)	(3 252)	(2 009)	(932)	(1 318)	(760)	(1 409)	(473)

- *Float* literals are more often considered *Magic* (42.8%) even for systems in the mathematical domain (*Polymath*: 51.9%). We give an example of a float considered magic in *Polymath* in Listing 10 (next section). Results for float literals in *Morphic*, *Pharo*, *Seaside*, and *VMMaker* may be insignificant because they occur much less (fewer than 40 cases for *Morphic*, *Seaside*, *VMMaker*; 91 cases for *Pharo*).

- *Symbols*, *Booleans*, and *Strings* are generally considered less *Magic* (10% or less of *Magic* literals among them). Our explanation is that *Symbols* and *Strings* are often self-explanatory and *Booleans* have only two possible values. See Listing 9 for an example of a *Boolean* considered *Acceptable* by two participants (and *Undecided* by a third one). Listing 11 (page 23) gives examples of a *String* and a *Symbol* literals considered *Acceptable*.

- Apart from *Float* (already discussed above), the types with most variation are *nil*, *array*, and *Character*. Only 4.8% of *nil* literals are considered *Magic* in *Morphic*[9] but 36.5% in *Polymath*; 7.7% *Magic array* in *Morphic* but 34.4% in *VMMaker*; 0% *Magic Character* in *Morphic* and *Roassal* but 38.1% in *Pharo*. However, these results may not be very significant

[9]We do not consider *Parser* here because it has very little *Magic* literals overall.

because they have fewer occurrences (only 473 *Character* literals in total over the 24,207 classified; 760 *nil* and 1,318 *array* literals.

- The literal *nil* was considered *Magic* in 15% of the cases. This is another indication that people may have different definitions of what is *Magic*, like understanding what nil means, but not understanding *why* it is used at a specific location in the code. Listing 9 shows an example where the *nil* literal was classified as *Magic* by a "low experience" participant (and *Acceptable* by another "low experience" participant and a "high experience" participant).

```
RSRotated>>#dragEnd: evt
    drag := false.
    prevAngle := nil.
```

Listing 9: Example from *Roassal*. RSRotated is a class handling user interaction (mouse drag rotation).

4.4. Validating Hypotheses

We may now check the validity of the 5+1 hypotheses formalized in Section 3. **Hyp1** (Section 3.1) differs from the others as it looks at the experience of the developer. The remaining five hypotheses (Section 3.2) only consider the source code.

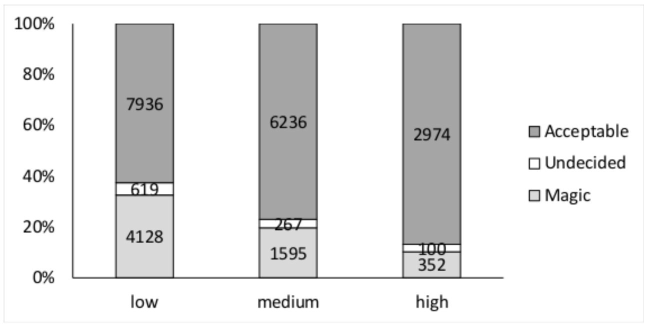


Figure 2: Classification of literals according to expertise of the participant (**Hyp1**). Bars show percentages, numbers give raw values.

Figure 2 shows, for each category, the classification of literals according to the participants' expertise in the subject system, as self-declared by the participants. One clearly sees that the proportion of literals considered *Magic* drops as the expertise increases. Although this is less clear in the figure, the same goes for the proportion of *Undecided* (4.9% for "low" expertise, 3.3% for "medium" expertise, and 2.9% for high expertise). A χ^2 test confirms that this hypothesis holds (p-value $< 1e{-}5$).

Listing 9, in the previous section, showed example of literals classified differently by participants with different expertise. Listing 10 shows an example of a domain-specific literal (Euler–Mascheroni constant, used in the Gumbel distribution[10]) that was classified as *Magic* by a (self declared) "low expertise" participant.

```
PMFisherTippettDistribution >> average
    "Answer the average of the receiver."
    ^0.577256649 * beta + alpha
```

Listing 10: Example from *Polymath* where *Float* literal `0.577256649` (Euler–Mascheroni constant) was classified as *Magic* by one participant.

Table 6: Number of *Magic* and *Acceptable* literals overall and for the five hypotheses based on the code. Hypotheses 2 to 5 give the numbers for the literals respecting its condition and the "other literals" (i.e., "Overall" - hypothesis). Hypothesis 6 (test methods) is not compared to "Overall" methods but to "Regular" methods.

		# hypothesis	# other	% magic
Overall	*Magic*	6 075	-	25.1 %
	Acceptable	18 132	-	
Hypothesis 2 (self describing)	*Magic*	672	5 403	9.4 %
	Acceptable	6 447	11 685	
Hypothesis 3 (lang. convention)	*Magic*	289	5 786	18.1 %
	Acceptable	1 310	16 822	
Hypothesis 4 (named literals)	*Magic*	285	5 790	20.8 %
	Acceptable	1 086	17 046	
Hypothesis 5 (pragma arg.)	*Magic*	74	6 001	8.5 %
	Acceptable	795	17 337	
Regular	*Magic*	3 137	-	26.6 %
	Acceptable	8 673	-	
Hypothesis 6 (tests/examples)	*Magic*	2 938	3 137	23.7 %
	Acceptable	9 459	8 673	

For each of the five remaining hypotheses (see Table 6) we count the number of *Magic* and *Acceptable* literals that respect the conditions of the hypothesis (e.g., number of *Magic* and *Acceptable* literals as pragma argument for **Hyp5**). These values are compared to the number of literals that do not respect the condition ("other literals" = Overall - hypothesis). For **Hyp6** , we compared to the number of "regular" methods, because "overall" also contain these text and example methods.

[10]`https://en.wikipedia.org/wiki/Gumbel_distribution`

We thus obtain a contingency table for each hypothesis and the independence of the categories is tested with a χ^2 test. If the p-value of the test is lower than 5%, we conclude that the distribution of *Magic*/*Acceptable* literals in the hypothesis does not follow the distribution in the other literals. We then compare the percentage of *Magic* literals in the hypothesis to the percentage in the other literals and see whether the literals in the hypothesis tend to be less magic.

The p-values of the χ^2 tests are $< 1e{-}5$ for the five hypotheses confirming that "hypothesis literals" and "other literals" do not follow the same distribution.

The percentage of *Magic* literals is inferior to 25.1% (overall percentage from Table 6) for all hypotheses, confirming that there are fewer *Magic* literals respecting the conditions of the hypotheses. In other words, these literals are more likely to be *Acceptable*.

However, all hypotheses do not have the same strength. We discuss each of them in more detail.

Hyp2 (self-describing literals) is strong with 9.4% literals considered *Magic*. In detail, symbols and empty arrays have around 6.5% *Magic* literals; Booleans and strings have around 10% *Magic* literals. To our surprise, the empty pointer (nil) literal was the worst, with almost 15% considered magic.

Listing 9 shows examples of the *nil* literal and a boolean one. These two have conflicting classifications: *nil* was classified as *Acceptable* by a "low experience" participant and a "high experience" participant, and *Magic* by another "low experience" participant; the Boolean false was classified as *Acceptable* by the same "low experience" and "high experience" participants, and *Undecided* by the "low experience" participant.

Listing 11 shows an example of a string literal ('All superclasses') and symbol literal (#systemIcon) that were classified as *Acceptable* by a "medium experience" participant.

```
Behavior >> spotterSuperclassesFor: aStep
<spotterOrder: 40>
aStep listProcessor
    title: 'All superclasses';
    allCandidates: [ self allSuperclasses ];
    itemIcon: #systemIcon;
    filter: GTFilterSubstring
```

Listing 11: Example from *Pharo* where the string 'All superclasses' and symbol #systemIcon were evaluated as *Acceptable*. The pragma argument (40) was judged *Magic* by the same participant.

Hyp3 (language conventions) is somehow weak with 18.1% literals considered *Magic*. In detail, three conditions are less or equal to 8% *Magic* literals: "1" as starting value of a loop; strings and characters appended to output streams; and strings in the name: message. The condition on increment/decrement of 1 is much less effective with 25.5% *Magic* literals (value not shown), worse than the overall percentage. This is especially surprising, as incrementing (or decrementing) by 1 has often been considered an *Acceptable* literal (e.g., see the MagicNumber rule of Checkstyle: `https://checkstyle.sourceforge.io/config_coding.html#MagicNumber`).

Listing 12 shows a typical example of a loop where the literal "1" was classified *Acceptable* by two "low experience" participants when used as the initial value in the loop `1 to: self size do:`.

```
PMVector >> log
   "Apply log function to every element of a vector"
   1 to: self size do: [ :n | self at: n put: (self at: n) log].
```

Listing 12: Example from *Polymath* where the "1" literal is *Acceptable* as initial value in the loop `1 to: self size do:` .

Hyp4 (named literal) is also weak with 20.8% literals considered *Magic*. In detail (not in the table), the literals assigned to a variable (20.5%) is a bit better than literals directly returned by a method (22.3%). It is surprising that, when defined as a constant, one in five literals is still considered *Magic*. It might be due to poor name choices, or it might suggest that the problem of literals' semantics goes beyond simply giving them a name. Another possible explanation would be that the participants only see a small part of the code (the method containing the literal). It may be that on a larger scale, the interest of having named constants is more obvious.

Listing 13 shows a named literal in the code for the Calypso browser (Pharo's tool to browse classes and their methods). It was evaluated as *Magic* by a "high experience" participant and *Acceptable* by a "medium experience" participant. The value here is used to order entries in a menu.

```
ClyMetaLevelToolbarGroup >> order
   ^3
```

Listing 13: Example of a named literal with conflicting evaluation.

Hyp5 (literals in pragmas) is the strongest with 8.5% literals considered *Magic*. This seems natural as the literals are mandatory in pragmas (annotations). However, a counter-example is proposed in Listing 11 where the pragma argument (`40`) was classified *Magic* by a "medium experience" participant.

Finally, **Hyp6** (test and example methods) is the worst with 23.7% of literals considered *Magic*. This result must not be compared to the number for *Overall* which also includes the test/example methods; it must be compared to the number of literals considered *Magic* in regular methods: 26.6%.

Although the result for this hypothesis is statistically relevant (different from the result for regular methods), it seems too close to the "regular" percentage to have practical relevance. In detail (not shown in the table), literals in example methods are *Magic* at 26.8%, slightly worse than the "regular" percentage. Literals in test methods (23.1%) are slightly better but still not different enough from the "regular" percentage. One might consider treating literals in test methods the same as in regular methods.

5. Threats to validity

We now consider the threats to validity of our study:

Construct validity. Are we asking the right questions?

Magic literals are well known to all software developers and the question asked to the participants was quite straightforward ("Is this literal *Magic* or not?").

Yet, we were surprised to see that 20% of literals defined as constants (according to the Pharo convention of defining a method solely returning a literal) were classified as *Magic*. It could be a sign that naming a literal is not always enough, or perhaps the names were unclear, or despite the well known rule, it could be the sign that the definition of *Magic* literal is not as universal and clear cut as one could think. We did try to mitigate this risk by adding a note at the bottom of the tool (see Figure 1): "A _magic_ literal is a literal that you don't really understand (Why this value? What happens if you change it?)".

We could have tried to be more specific on what should be considered a *Magic* literal, but we believe this would have biased the study towards *our* understanding of the term. Our goal was to evaluate what developers (at large) would consider *Magic* or not.

Internal validity. Is there something inherent to how we collect and analyze the data that could skew our findings?

The analysis of the participants' answers is very simple, consisting only in statistics on the number of literals. This does not seem to raise any issue.

The validation of the heuristics (Section 4.4) has a problem in that the same literals may appear several times in different hypotheses. For example, strings are used to evaluate **Hyp2** (they are self-describing) and some of them are also used when they are an argument of a pragma (**Hyp5**), an argument of specific messages sent (**Hyp3**), or returned by a method (**Hyp4**). The usual way to deal with this is to multiply the p-value by the number of tests applied (Bonferroni correction method). Given that our p-values were several orders of magnitude lower than the level of acceptance (5%), this issue has been amply addressed.

There might be different, maybe competing, notions of what constitutes magicness. By combining the ratings of all participants into one large dataset, these different notions of magicness would get blurred. This is a threat that we did not explore in this first study.

External validity. Are our results generalizable to other environments?

Our studies were conducted on only one programming language that is not a mainstream language. This is an issue for generalization of the results and we do not claim they can be applied to all systems or programming language. We did raise some interesting points that should be investigated in other contexts to see whether they still hold.

The choice of the subject system could also introduce a bias although we did try to select very different systems, from different domains, of different sizes, with different developers.

The literals were manually classified which always introduce a risk of a bias if the number of participants is too small, for example. We have a fair number of participants (26), but there could be a "community bias" as they are all Smalltalk developers. This goes back to our first point on the programming language.

Reliability. Can others replicate our results?

The classification of literals as *Magic* vs. *Acceptable* would be "easy" to replicate given the needed resources. All literals and their classification by participants are available at `https://github.com/NicolasAnquetil/MagicLiteralsData.git`. The README.md file contains explanation on how to reproduce the results of the hypotheses validations from the data.

6. Discussion

We have defined magic literals and shown that literals in general and magic literals in particular are highly prevalent in software systems. We also validated our hypotheses on what can make a literal acceptable.

In this section, we discuss other points of interest.

6.1. Magic literal detection

One purpose of this research would be to implement rules in programming environments to help developers improve their code by dynamically checking (and advising the developer) whether a particular instance of literal is acceptable or should be named more explicitly.

Because the topic is such an old one and has made so little progress in all these years, we believe it would be important to have a mechanism that would point to *Magic* literals with a high degree of confidence (good precision), even at the cost of missing some (medium recall). Unfortunately, our hypotheses are currently geared toward detecting *Acceptable* literals and not the opposite.

As an early experiment, we implement the hypotheses as rules to detect *Acceptable* literals. We obtained very good precision (88%: 1 out of 10 literals tagged *Acceptable* should actually have been considered *Magic*) and acceptable recall (34%: one third of all *Acceptable* literals were detected). Unfortunately, this means that what we believe is needed (detecting *Magic* literals) gave opposite results (87% recall and 32% precision). Only one third of the literals pointed to as *Magic* by these rules (i.e., the opposite of our hypotheses) would really be *Magic*. This seems too low to get a good acceptance from developers.

6.2. Magic Strings

We could develop better heuristics to detect string literals that are not self-describing (**Hyp2**). One possibility would be to look for real words in the strings, for example using a dictionary of common English words (`https://github.com/dwyl/english-words`). Strings with too few real words would be declared not self-describing and therefore *Magic*.

A possible follow-up would also be to consider application domain vocabulary, maybe using natural language techniques and topic modeling techniques.

6.3. Method Selectors

As shown before, the Pharo method selector gives a context for some literals (**Hyp3**). This section explores how this context can be better exploited. For example, some Pharo projects make heavy use of external C libraries (Cairo graphics, libgit, SQLLight, etc.). They rely on FFI (Foreign Function Interface) methods (ffiCall:, ffiCall:option:, ffiCall:module:option:) that take as a first argument an array of symbols describing the C function to call (e.g., #(String strcpy #(String dest , String aString)). In such a context, the array of symbols is *Acceptable*.

The use of keywords to denote parameters in Pharo can also be very revealing. For example, the class-side (i.e., static) message "year:month:day:" of the Date class gives a clear meaning to its literal arguments in the call: "date := Date year: 2019 month: 6 day: 1."

We also identified another Pharo convention relating to the description of project configurations and dependencies. These are usually defined in a BaselineOfXYZ or ConfigurationOfXYZ class with specific methods. In these classes and methods, most integer literals correspond to version numbers and strings or symbols describing other project names, or specific configurations' names. They are thus *Acceptable*.

In conclusion, specific projects may have more of these conventions. An in-depth study of Pharo and any interesting project needs to be done to identify more interesting contexts.

6.4. Location of Literals

The locations in the source code where literals can appear depend on the abstract syntax tree (AST) of the source code. Thus, literals can play different roles in an AST:

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 the following).
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 (which are like lambdas) for the return value of the block.
7. *Pragma:* Argument to a pragma (which are like annotations in Java).

Table 7: 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).

	Arg.	*Oper.*	*Pragma*	*Return*	*Assig.*	*Receiv.*	*Seq.*
Morphic	4 306	3 596	12	937	704	319	439
	41.6%	34.7%	0.1%	9.0%	6.8%	3.1%	4.2%
Parser	263	122	0	5	60	33	4
	53.8%	24.9%	0.0%	1.0%	12.3%	6.7%	0.8%
Pharo	80 690	32 475	2 038	8 361	7 003	9 057	3 703
	56.0%	22.6%	1.4%	5.8%	4.9%	6.3%	2.6%
Polymath	5 653	2 853	21	97	412	820	91
	56.8%	28.7%	0.2%	1.0%	4.1%	8.2%	0.9%
Roassal	8 453	5 439	287	866	1 227	802	215
	48.8%	31.4%	1.7%	5.0%	7.1%	4.6%	1.2%
Seaside	5 852	1 263	0	393	311	160	215
	71.4%	15.4%	0.0%	4.8%	3.8%	2.0%	2.6%
VMMaker	23 164	16 168	15 307	4 428	3 142	1 438	3 099
	34.6%	24.2%	22.9%	6.6%	4.7%	2.2%	4.6%
all	128 381	61 916	17 665	15 087	12 859	12 629	7 766
	49.9%	24.1%	6.9%	5.9%	5.0%	4.9%	3.0%

Table 7 shows that literals, on average, appear half of the time (49.9%) as message *arguments* and a quarter of the time (14.1%) as *operands.* Considering its mathematical content, *Polymath* surprised us by ranking only third (28.7%) among the projects with respect to *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 others (the second is *Roassal* with 1.7%). This is because the VM is generated from a high-level description in a language (Slang [9]) that requires metadata described in pragmas.

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.

7. Related Work

To the best of our knowledge, no similar study of magic literals in software systems exist in the literature. This paper addresses the problem in the context of Pharo.

In 1978 Kernighan and Ritchie [5] stated it is bad practice to place "magic numbers" in a program, because they "convey little information to someone who might have to read the program later, and they are hard to change in a systematic way." As such, they discuss the modularity problem caused by magic literals that should be parameterizable. Parnas [12] and later McConnell [7] described this problem as *excessive information distribution*, which is a barrier to information hiding. The consequence is increased complexity, and code that is harder to change when the same magic numbers are buried throughout the program. Fowler [3] and Martin [6] described it as a code smell that should be avoided. All these authors claim that magic numbers should be replaced with symbolic constants to improve understandability. McConnell [7] described named constants as a lower-level use of information hiding in design. None of the authors evaluated the concept of magic numbers as thoroughly as we did on real systems.

Smit et al. [13, 14] identified the relative importance to maintainability of 71 coding conventions based on a survey of seven developers. They analyzed 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 avoiding the use of magic numbers. Their definition [14] considered -1, 0, 1, and 2 as non-magic (note that other people only consider -1, 0, and 1, see for example `http://wiki.c2.com/?ZeroOneInfinityRule`, see also static analyzers' rules below). 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.

Nundhapana and Senivongse [11] discussed the approach taken by an IT organization in Thailand to enforce naming conventions in Objective-C. They developed a library to check naming conventions automatically. In particular, magic numbers are considered as violation of naming conventions because they are unnamed literal constants. They devised a checker based on regular expressions to detect magic numbers in Objective-C source code. They reported results of a readability experiment involving code before and after it was revised according to violations of conventions identified by their tool. However, no specific data was given regarding violations of magic numbers.

Both Smit et al. [13, 14] and Nundhapana et al. [11] distinguished "magic numbers" from "literal strings," the latter being the string equivalent of magic numbers. Therefore, they assume that all string literals are *Magic*. Our experiment shows that this is hardly the case. Their different interpretation may come from mixing understandability and maintainability (Section 3.1). A string literal is usually understandable (**Hyp2**: self-describing literals), but if it occurs multiple times in the code, it is detrimental to maintainability. The solution is the same: multiple occurrences of the same string literal should be extracted into a constant.

Mukherjee [10], who explained how to perform static analysis of C#, presented three techniques for finding magic numbers in C# systems, in particular in *arithmetic expressions*, *array indices*, and *conditions*.

Literals in test code are addressed by Deuresen et al. [16] in the context of a test smell they called *Sensitive Equality*. They are discussed as a symptom of a bigger problem, which is a lack of an equality method. Meszaros [8] documented the *Literal Value* pattern, which states to use literal constants for attributes and assertions in tests. The intention of the pattern is to make tests less obscure (more readable). A potential disadvantage of the pattern is that it can be overused, e.g., a key for a database defined as a literal value can actually make a test more obscure.

There are various static analysis tools to verify the quality of the source code. We consider three of them here that are well known.

The PMD[11] tool 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`.
- AvoidLiteralsInIfCondition.
  `https://pmd.github.io/latest/pmd_rules_java_errorprone.html#avoidliteralsinifcondition`.

Similarly, CheckStyle[12] has the following rules pertaining to literals:

- MultipleStringLiterals. Checks for multiple occurrences of the same string literal within a single file.
  `https://checkstyle.sourceforge.io/config_coding.html#MultipleStringLiterals`
- 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.
  `https://checkstyle.sourceforge.io/config_coding.html#MagicNumber`

Sonarqube[13] 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`.

[11]https://pmd.github.io/
[12]https://checkstyle.sourceforge.io/
[13]https://www.sonarqube.org/

- Literal boolean values should not be used in assertions `https://rules.sonarsource.com/java/RSPEC-2701`.
- URIs should not be hardcoded `https://rules.sonarsource.com/java/RSPEC-1075`.
- Magic numbers should not be used, -1, 0, and 1 are not considered magic numbers. `https://rules.sonarsource.com/java/RSPEC-109`

Some of these rules are specific ("URIs should not be hardcoded") and could fall in our **Hyp3** (Language conventions). These tools do seem to miss many *Magic* literal instances. They may have stumbled on the same problem as ours, explained in Section 6.1.

8. Conclusion

In this paper we explored the concept of magic literals generally and more specifically in the context of Pharo. We looked at how literals occur in systems leading us to a definition of why *Magic* literals are to be avoided, and what could characterize acceptable literals.

We show that literals are still very prevalent in real systems and even more so in test methods. We also evaluated the prevalence of different types of literals (integers, strings, etc.) and show that if there is some variation in specific domains (e.g., mathematical), integers and strings are the most prevalent.

We validated our hypotheses about the reasoning behind acceptable literals on real data showing that they hold, even if at least one of them (**Hyp6**) does not seem particularly interesting. Some intriguing points emerged in this evaluation for which we do not have immediate explanation:

- There are more literals in test methods, but they are not considered less *Magic* than in non-test methods;
- The Pharo convention for creating constants is to have a method solely returning a literal (the constant is a method, not a variable as commonly done in Java and other languages). Yet 20% of these (named) literals were considered *Magic* (as opposed to 25% of all literals being considered *Magic*). This is not a huge improvement and suggests that naming a literal as a constant does not solve everything. It could also be the result of showing only a small part of the code (the method containing the literal) to participants. It could be that on a larger scale, the interest of having named constants is more obvious.
- the literal "1" in $x + 1$ and $x - 1$ is not considered less *Magic* than the average literals (25% *Magic*)
- The empty pointer literal (`nil` in Pharo) was considered *Magic* in 15% of the cases. This is another indication that people may have different definitions of what is *Magic*, like understanding what `nil` means, but not understanding *why* it is used at this specific location in the code.

The research we conduct in this article opens multiple perspectives. We want to dig deeper in the analysis of magic literal occurrence by studying how and why they occur system per system. We think that some domains are probably more subject to *Magic* literals than others. Our experiment did show that the domain had an influence on the type of literal used (like *Character* literals for *Parser*). We would like to test this hypothesis on a large set of Pharo subsystems addressing various problems of different domains.

Finally, an empirical study of the evolution of magic literals across multiple versions of these Pharo subsystems will help us to understand why magic literals appear. Such a study consists in doing a *post-mortem* analysis of commits that occurred during the development of the system. We will compute the difference between each pair of consecutive versions of each system and watch for the appearance of magic literals.

Appendix A. Subject systems

Table A.8 shows the version information for the seven systems used in the studies of this article.

Table A.8: Versions of systems used in the studies.

Project	Version / commit-ish	Date
Morphic	*Pharo 8.0* (see below)	
Parser	*Pharo 8.0* (see below)	
Pharo	*Pharo 8.0*, `https://github.com/pharo-project/pharo/commit/a153e04ae4e325509ed78bfe25c9ce560afb24b0`	Mar 27, 2020
Polymath	`https://github.com/PolyMathOrg/PolyMath/commit/b73ac039715d7997942fa78baf16782d9b5300af`	Jan 24, 2020
Roassal	`https://github.com/ObjectProfile/Roassal3/commit/6534657660bf8ed518963b95dd5942d0e2037e1d`	Apr 7, 2020
Seaside	`https://github.com/SeasideSt/Seaside/commit/506db6a883588b27ccfc6edc56054c20a1e9093f`	Jun 28, 2019
VMMaker	`https://github.com/pharo-project/opensmalltalk-vm/commit/2baf863adcb44881ba4fd9c0cb92ed31a96c9546`	Dec 2, 2019

Appendix B. Data extraction using reflective API

The data from Table 2 was extracted from the reflective API of Pharo:

1. First specify the list of all classes of interest for a given system (for example for *Seaside*, all classes in a package prefixed by "Seaside");

2. then get all methods for each class (aClass methods);
3. get the parse tree of the method (aCompiledMethod parseTree);
4. visit the Abstract Syntax Tree to get all literals.

References

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

[2] Julien Delplanque, Stéphane Ducasse, and Oleksandr Zaitsev. Magic literals in pharo. In *International workshop of Smalltalk Technologies*, 2019.

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

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

[5] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall Software Series, 1978.

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

[7] Steve McConnell. *Code Complete.* Microsoft Press, Redmond, Wash, 2 edition edition, June 2004.

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

[9] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. Two decades of smalltalk vm development: live vm development through simulation tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 57–66. ACM, 2018.

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

[11] 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.

[12] David Lorge Parnas. Designing software for ease of extension and contraction. In *International Conference on Software Engineering (ICSE'78)*, pages 264–277, 1978.

[13] 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.

[14] 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.

[15] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. "automatically assessing code understandability" reanalyzed: Combined metrics matter. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 314–318, 2018.

[16] 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.

[17] 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.