The Object Repository

Pulling Objects out of the Ecosystem

Boris Spasojević
University of Bern
spasojev@inf.unibe.ch

Mohammad Ghafari
University of Bern
ghafari.unibe.ch

Oscar Nierstrasz
University of Bern
oscar@inf.unibe.ch

Abstract

In this paper we propose the idea of constructing an Object Repository – a repository of code snippets that, when executed, produce an instance of some class. Such a repository may be useful for several software engineering tasks like augmenting software documentation, testing object inspectors, improving program comprehension *etc.*

We mine code snippets from existing software systems via brute force execution of code segments obtained through converting AST nodes of methods to source code. The gathered snippets are known to be executable, and this is a novelty which is not possible with existing approaches.

We show that applying the proposed approach to 141 open source Pharo projects results in an Object Repository that can instantiate almost 80% of the available classes in these projects.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords keyword1, keyword2

1. Introduction

Every object-oriented language includes a mechanism for creating new objects, and developers leverage this mechanism to create complex objects needed in software systems. Most classes contain a default constructor, a way to create a default instance of that class, but some classes do not have a default representation, and are instantiated through more complex construction mechanisms such as parametrized constructors, factory methods, the builder design pattern and so on. Fully formed objects can also be created through any valid object usage protocol, which may include an arbitrary number of interacting objects and method invocations.

Finding a way to properly instantiate classes can be non-trivial, yet, code snippets that instantiate these classes exist throughout their client classes. Existing approaches for mining code snippets provide useful information about object creation and usage, but they lack the feature of being executable which is important to enable approaches that require objects on demand.

We propose to mine available software projects for code snippets that instantiate classes, henceforth referred to simply as snippets. We aim to create a repository of such snippets, which can be used to facilitate several software engineering tasks such as augmenting documentation, new testing approaches, support for program comprehension and others. We realize this approach by extracting all AST nodes from all methods of all available classes, converting them to their source code representation and attempting to execute them. If the execution is successful, *i.e.*, produces an object, we save the snippet in a database and associate it to the type of the produced object.

For our case study we chose Pharo, a Smalltalk inspired language. This choice was made because of the high reflectivity of Smalltalk [6], which enables us to move quickly with implementing such a system. We applied this approach to 141 open source Pharo projects and a selection of classes contained in the base Pharo image. We find that the result of this approach is that around 10% of AST nodes, when converted to source code and executed, produce objects for almost 80% of all the analysed classes. We check several aspects of the snippets to better understand their properties and also analyse the nodes that failed to produce objects, and discuss how to tackle the reasons for the failures.

The paper is organised as follows: Section 2 discusses potential uses of the Object Repository, Section 3 describes the proposed approach to realize such a repository, Section 4 presents the evaluation and discusses the results, Section 5 review related work to this research. Section 6 explains our future plan, and Section 7 concludes the paper.

[Copyright notice will appear here once 'preprint' option is removed.]

2. Motivation

This section presents some of potential use cases for the Object Repository. Though there are several software engineering tasks that can benefit from the availability of such a repository, our discussion advocates three of which that we found most applicable.

2.1 Software Documentation

Documentation, when available and up to date, is still the most reliable and widely used resource for understanding software systems and APIs. Much work has been done around the idea of mining usage examples to enrich documentation [2, 9, 23], however, none of the example snippets given are usable to directly create objects. Code snippets from the Object Repository could be used as a complementary source for such examples, with the additional knowledge that the snippets are immediately executable. The main requirement for this use case is to have concise and representative snippets.

Also, since the snippets in the Object Repository produce objects, one could introduce a concept of a "playground" within the documentation where a developer could experiment with a given live instance of the class whose documentation she is reading. Many similar "playgrounds" exist for languages such as Go[1] and Haskell[2], allowing developers to simply try out parts of the language.

This requires at least one snippet associated with the documented class. Unlike the previous case, the quality of snippet is of no importance, as the user is only meant to interact with the object.

2.2 Software Testing

Modern approaches to inspecting objects rely on object specific representations [4]. This means that the author of a class, or anyone else through extensions, can specify a way that the object can represent itself. The object inspectors provide the user all available ways to represent the object, and the user chooses one that suits the current context.

According to our discussion with researchers in the field, testing new representations is laborious since they are usually required to create the objects manually. Alternatively, the Object Repository could provide a set of objects gathered from the ecosystem that enables testing the new representation of such objects automatically.

Testing software by generating random test inputs is a well researched field [3, 5, 17]. Integrating objects from the Object Repository with the random input generated by these approaches could help cover the corner cases that are hard to detect with raw random testing. Moreover, starting from actual instances from the Object Repository instead of, or in combination with, random ones could improve the results. For instance in case of genetic algorithms [15], this can guide the genetic algorithm to an acceptable population of input data much faster while also avoids local maximums.

Method arguments are usually checked for validity at the beginning of a method. In case the argument is not valid, the method should signal this fact to the caller in a standardized manner *e.g.*, by throwing an exception, or returning an error value. In order to test this a developer would require multiple instances of the argument type both valid and invalid in the context of being input for that method. Assuming that arguments of the method are of a type that is present throughout the ecosystem, the Object Repository should contain code snippets needed to create such instances. This facilitates, to some extent, verifying automatically that the validation of the method arguments behaves as expected.

To realize these use cases the Object Repository should contain as many snippets associated with a class as possible. The snippets should also produce representative and diverse objects.

2.3 Software evolution and maintenance

While studying source code is the main way that developers interact with programs [10, 11] many program comprehension tasks require runtime observation [12, 13]. Nevertheless, running a system and placing it in a desired state can be challenging for several reasons like lack of input to the system, lack of knowledge about the system, long system running time before reaching a desired point.

Having a way to create a live object of a required type could spawn a running system at any point in the source code by filling all the gaps in the execution context with blank objects of the adequate type. These objects should be presented in an object-inspector-like interface allowing the developer to set the values of these context objects and guide the execution of the program, as one would do in a debugging session. One example of one such usage could be in the domain of application security. Executing parts of source code in a sandbox created from objects taken from the Object Repository could ensure that the execution does not have any unexpected side effects.

Objects taken from the Object Repository could be used to help disambiguate results of type inference engines for dynamically typed languages [19]. Many of these type inference engines provide a list of potential candidate types for a variable. Thanks to the Object Repository, having instances of those types, and attempting to execute the code with each of those objects assigned to the variable in question could shed some light on the types which are more likely false positives.

The main requirement for these use cases is to have an Object Repository that contains snippets associated to as many classes as possible, extending the applicability of this use case to more project.

[1] `https://play.golang.org/`

[2] `https://tryhaskell.org/`

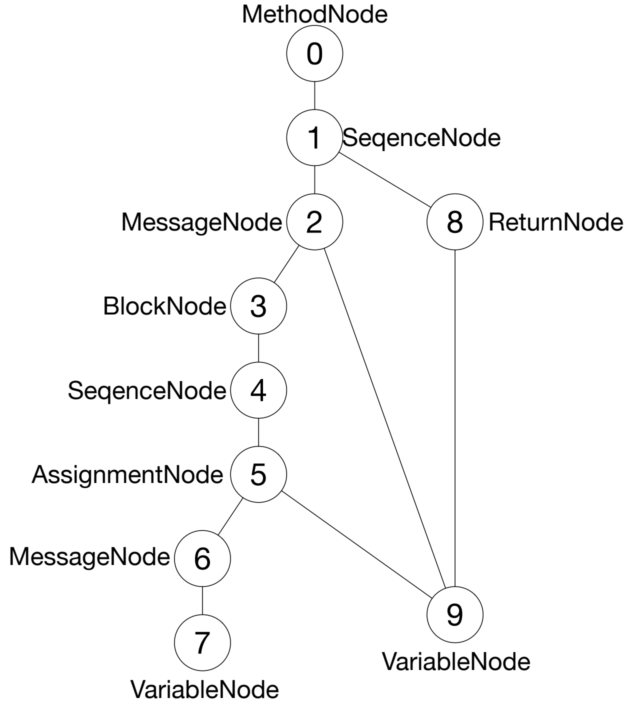

Figure 1. Abstract syntax tree of the method in Listing 1.

$$C : \textit{Domain of classes} \quad (1)$$

$$M : \textit{Domain of methods} \quad (2)$$

$$\textit{def}_m : M \rightarrow C \quad (3)$$

$$N : \textit{Domain of AST nodes} \quad (4)$$

$$\textit{def}_n : N \rightarrow M \quad (5)$$

$$S : \textit{Domain of snippets} \quad (6)$$

$$\textit{toCode} : N \rightarrow S \quad (7)$$

$$O : \textit{Domain of objects} \quad (8)$$

$$O_0 = O \cup \emptyset \quad (9)$$

$$\textit{instanceof} : O \rightarrow C \quad (10)$$

$$\textit{execute} : S \rightarrow O_0 \quad (11)$$

$$N_{exec} = \{n \in N | \ (\textit{execute} \circ \textit{toCode})(n) \neq \emptyset\} \quad (12)$$

$$S_{exec} = \{\textit{toCode}(n), \forall n \in N_{exec}\} \quad (13)$$

$$\textit{objectRepo}(c \in C) = \{s \in S_{exec} | \ (\textit{instanceof} \circ \textit{execute})(s) = c\} \quad (14)$$

Figure 2. The core domains and functions of the formal model.

3. The Approach

We aim to mine code snippets from projects by transforming all available methods into their abstract syntax tree (AST) representation, transforming each AST node[3] into its source code representation, attempting to execute it and observing the return value of the execution. We ignore this code snippet if the execution fails to compile, or to produce a return value. The ones that return an object are saved in the Object Repository and associated to the type of object produced by their execution.

For example, Listing 1 shows a method from the *Pomodoro* project[4]. This method, checks if an instance variable *progressBar* is *nil*(line 2) and, if so, assigns it a new instance of *ProgressBarMorph* (line 3). Finally, it returns this instance variable (line 5).

```
1 PomodoroMorph>>progressBar
2   progressBar ifNil: [
3     progressBar := ProgressBarMorph new
4   ].
5   ^ progressBar
```

Listing 1. Example method used to ilustrate the approach.

We first parse this method and build the AST. Figure 1 shows a graphical representation of this AST with all the nodes indexed and the type of the node shown next to it. We then transform each node into a source code snippet, and attempt to execute it.

Table 1 summarizes the results. This table represents each AST node by index, the type of the node, the source code representation of the node, and finally the type of the value obtained by executing the snippet. We can see from this table that out of the 9 AST nodes only 2 are, when transformed into source code, executable and produce objects.

3.1 Formal Model

To better explain the proposed approach, we introduce a small set-theoretical model. This subsection discusses domains and sets used in this model, the function that models the execution of the snippets, as well as the function used to retrieve snippets for a given class.

As shown in Figure 2, C (Equation 1) is the domain of classes, M (Equation 2) is the domain of methods defined in the classes and N (Equation 4) is the domain of AST nodes defined in the methods. Each method is defined in one class (Equation 3), and each node is defined in one method (Equation 5).

The conversion of the AST nodes into source code is defined as a function *toCode* (Equation 7). The codomain of this function is S, the domain of all code snippets. Since multiple AST nodes in N can have the same source code representation, *toCode* is a surjective function, and thus for

[3] Each AST node is technically an AST tree *i.e.*, the subtree of the AST of the method with the node in question at the root. We use the term “AST node” in place of “AST subtree with the node at the root” for simplicity.

[4] `http://smalltalkhub.com/#!/~TorstenBergmann/Pomodoro`

#	Node Type	Corresponding Snippet	Result Type
1	SeqenceNode	*progressBar ifNil: [progressBar := ProgressBarMorph new]. ˆ progressBar*	-
2	MessageNode	progressBar ifNil: [progressBar := ProgressBarMorph new]	-
3	BlockNode	[progressBar := ProgressBarMorph new]	-
4	SeqenceNode	progressBar := ProgressBarMorph new	-
5	AssignmentNode	progressBar := ProgressBarMorph new	-
6	MessageNode	ProgressBarMorph new	ProgressBarMorph
7	VariableNode	ProgressBarMorph	ProgressBarMorph class
8	ReturnNode	ˆ progressBar	-
9	VariableNode	progressBar	-

Table 1. The results of applying out proposed approach to the example method from Listing 1.

any $N' \subseteq N$ and the corresponding S' it holds that $|S'| \leq |N'|$.

The execution of source code from S is defined as a function called *execute* (Equation 11). Since not all snippets from S will, when executed, yield an object, the codomain of this function is the set O_0 (Equation 9). This set is defined as the union of object domain (Equation 8) and an empty set, used to denote a failed snippet execution. Each object is an instance of a class (Equation 10).

With all this in place, we define the set N_{exec} (Equation 12) as the set of all AST nodes that, when converted to source code and executed, produce an instance of any class from C. We call members of this domain "executable AST nodes". Correspondingly, S_{exec} (Equation 13), defines the domain of all "executable" code snippets.

Lastly, the *objectRepo* function (Equation 14) returns, for a given class from C, a set of snippets from S_{exec} that, when executed, produce an instance of the given class.

3.2 Implementation

We implemented the approach in Pharo Smalltalk because of it's reflective nature which allows us to move fast with the implementation. Most of the needed implementation was readily available in Pharo *i.e.*, parsing the source code to the AST, converting AST nodes to code snippets, *etc.* The main challenge was implementing the *execute* function

First, we wrap a code snippet inside a closure. We then create a temporary method in a temporary class whose source code is solely the execution of the mentioned closure. We then compile this method and, if the compilation is successful, we execute it wrapped in the Smalltalk equivalent of a *try-catch* block that catches all possible errors and exceptions.

This setup is enough to catch errors caused by a snippet not being compilable (*e.g.*, containing an undeclared variable) and the snippet failing to execute (*e.g.*, throwing a division by zero exception).

During the execution of code snippets, we encountered some never terminating executions. Further investigation revealed that such issues arise due to concurrency. For example, the snippet might wait on a signal from a different thread to continue the execution. However, we only execute a single snippet at a time, which means there is no chance of receiving such a signal. To restrain such executions, we limit the execution time for each snippet to 10 seconds. The time interval was chosen as an arbitrary cut of point with the reasoning that the execution of any snippet should terminate in less than 10 seconds in order for the snippets to be usable in any way. Though, a vast majority of snippets terminate quite quickly, we chose a very long timeout to include as many snippets whose execution will eventually terminate.

4. Evaluation

To evaluate our approach we developed a research prototype, and ran it on all classes taken from 141 open source Pharo projects whose source code was accessible from the Pharo package management system and on a selection of classes from the base Pharo image. We do not include all the classes from the base image because a large part of the functionality of the Pharo language is implemented in Pharo itself. Executing code snippets from such classes caused many errors that could not be handled from within the language, but required intervention at the Virtual Machine level. Examples are the contents of packages such of the *Kernel*, *Compiler*, *Debugger*, *NativeBoost* and others.

This evaluation includes a set of classes we call C' containing 13,909 classes. Correspondingly, the set of all methods from C' is noted as M' and contains 256,362 methods. These methods are comprised of 1,525,914 AST nodes defined in a set called N'. The number of nodes that are executable is only $|N'_{exec}| = 154,904$ or 10.15% of all the nodes. Converting these nodes into code produces $|S'_{exec}| = 92,460$ unique snippets of code. Table 2 presents the cardinalities of these sets.

In the rest of this section we define several sets, shown in Table 3, using which we discuss our findings from different perspectives.

We define a set C_d as the domain of the *objectRepo* function as shown in Figure 4, Equation 15. The cardinality of this set is 10,917 or 78.49% of all classes used in the evalu-

Set	Cardinality
C'	13,909
M'	256,362
N'	1,525,914
N'_{exec}	154,904 (10.15% of $\|N\|$)
S'_{exec}	92,460

Table 2. The cardinalities of the core sets used in the evaluation.

Set	Cardinality	$\%\|C'\|$	$\%\|Cd\|$	$\%\|C_1\|$
C_d	10,917	78.49%	-	-
C_1	8,779	63.12%	80,42%	-
C_{new}	2,384	17.14%	21.84%	27.16%
C_l	6,091	43.79%	55.79%	69.38%
C_g	2,442	17.56%	22.37%	-

Table 3. Cardinalities of the sets defined in Figure 4 and their relations.

ation. Being able to instantiate almost 80% of all the classes seems to be a promising result considering the minimalistic approach of extracting snippets.

4.1 Snippet Distribution

We call the set of all classes that can be instantiated through *only one* snippet C_1 as shown in Figure 4, Equation 16. This set helps us to better understand the quality of the snippets, by showing that 8,779 classes, or 80,42% of all instantiable classes, have only one associated snippet.

This, plus the fact that $|S'_{exec}| = 92,460$ shows that the distribution of snippet counts is heavily skewed to a minority of classes i.e., about 20%. Further inspection of the snippets shows that the ten classes with the most snippets accumulate over 60% of all snippets. Table 4 presents some information about these classes.

Most of the classes from Table 4, namely *Array*, *BlockClosure*, *ByteString*, *ByteArray*, *Point* and *Association*, have a very specific construction pattern. Some, like *Array*, *ByteString*, and *ByteArray*, have an idiomatic way of construction. For example, anything in source code between square brackets is considered a *BlockClosure*, anything between single quotation marks is a *ByteString*. Other classes, like *Point* or *Association*, have a very distinctive constructor, *e.g.*, two integers with the @ character between define a point with those integers as coordinates.

With all of this in mind we conclude that the heavy skewing of snippets is not surprising, but it might have a very negative impact on the usability of an Object Repository built with this approach. Although the Object Repository can instantiate almost 80% of all available classes, over 80% of those classes can be instantiated in only one way, and only

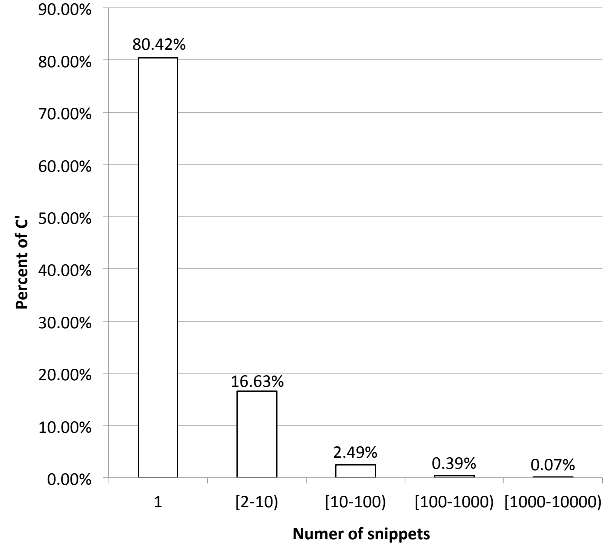

Figure 3. Distribution of classes in C_d according to the number of snippets that can instantiate each class.

$c \in C'$	$\|objectRepo(c)\|$	$\%\|S\|$
Array	16,907	18.29%
ByteString	14,968	16.19%
BlockClosure	8,826	9.55%
ByteSymbol	4,199	4.54%
ByteArray	3,807	4.12%
Point	2,725	2.95%
SmallInteger	2,502	2.71%
Association	1,237	1.34%
FixedDate	855	0.92%
Measure	759	0.82%
$\sum$	56,785	61.42%

Table 4. Ten classes with the most associated code snippets.

a handful of classes account for a majority of snippets. In Figure 3 we show a distribution of classes in C_d according to how many snippets can instantiate that class. We can see that just under 17% of classes can be instantiated by between two and ten snippets, while less than 3% with ten or more.

4.2 Trivial and Literal snippets

To further focus on the quality of the snippets we define the C_{new} set in Figure 4, Equation 17. This set includes all classes that have only one associated snippet, and that snippet is "trivial" *i.e.*, the default way of creating instances in Pharo. This is achieved by matching snippets with a regular expression that checks if the snippet is of form "*Class new*". An example of such a snippet would be "*Dictionary new*" which trivially creates a *Dictionary* object.

The cardinality of this set, as shown in Table 3, is 2,384 . This accounts for 27.16% of the classes with a single asso-

$$C_d = \{c \in C' |\ \exists s \in S', objectRepo(s) = c\} \quad (15)$$

$$C_1 = \{c \in C_d |\ |objectRepo(c)| = 1\} \quad (16)$$

$$C_{new} = \{c \in C_1 |\ \exists s \in objectRepo(c), regexMatch(s, \text{"\^{}[a-zA-Z0-9_]* new\$"})\} \quad (17)$$

$$C_l = \{c \in C_1 |\ \exists s \in objectRepo(c), regexMatch(s, \text{"\^{}[a-zA-Z0-9_]*\$"})\} \quad (18)$$

$$C_g = C_d \setminus (C_{new} \cup C_l) \quad (19)$$

$$C_o = \{c \in C |\ \exists n \in N'_{exec}, (def_m \circ def_n)(n) = c\} \quad (20)$$

Figure 4. Sets used during the evaluation of the Object Repository.

ciated snippet, or 21.84% of all classes from $|C_d|$. Considering that this is the default pattern of instantiating objects in Pharo, the percentage of classes instantiated only in this manner is not as high as might be expected.

We move on to other poor quality snippets by defining the C_l set as shown in Figure 4, Equation 18. This set is a subset of C_1, and contains all classes whose sole associated snippet is just one literal. This set is quite large as can be seen in Table 3. It has a cardinality of 6,091 or 55.79% of C_d. The size of this set is a result of Smalltalk's high reflective nature. Namely, following the "everything is an object" philosophy, each class in Smalltalk is essentially an instance of a corresponding metaclass, which in turn is an instance of the *Metaclass* class [8]. This leads to the phenomenon that executing a class name literal in Smalltalk will result in the object representing that class *i.e.*, an instance of the corresponding metaclass. This phenomenon accounts for all but 8 of the elements of C_l which are global variables which are mapped to concrete instances of regular (not meta) classes.

4.3 Promising Snippets

An interesting set to focus on is the set of all classes that can be instantiated by the *objectRepo* function in a non-trivial and non-literal way. This is essentially the domain of the *objectRepo* function excluding the sets C_{new} and C_l, and is defined as such in Figure 4, Equation 19. This set, named C_g, is actually containing the kind of data we wish to have to realize different use cases introduced in Section 2.

As shown in Table 3 this set contains 2,442 elements, or 17.56% of all classes included in the evaluation. This is not a large percentage of the classes analysed, but considering the minimalistic approach seems promising as the first step towards realizing the idea of building an Object Repository.

The main question regarding the quality of these snippets is whether or not they produce fully initialized objects. This question is outside the scope of this paper and is very complex as there is no automated way of concluding when an object is fully initialized. Investigating this question would most likely require input from authors of classes, as they have the necessary domain and code expertise to determine when an object is fully initialized.

Minimum	1
25% Quartile	16
Median	28
75% Quartile	51
Maximum	1,279,918

Table 5. Five number summary of snippet sizes.

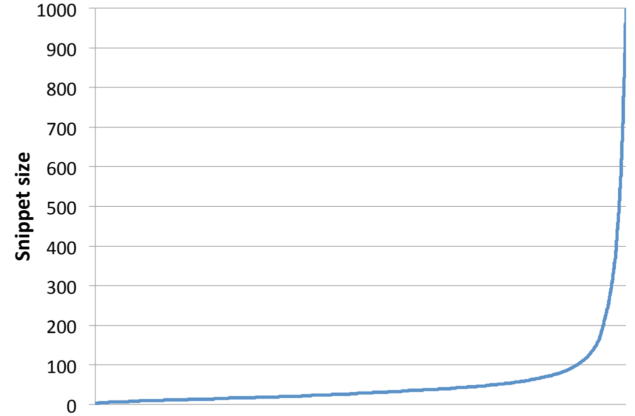

Figure 5. A sorted plot of sizes of snippet less then 1000 characters long.

4.4 Snippet size

The sizes of snippets in the Object Repository varies greatly. The smallest snippets are only one character long, an integer constant producing an instance of *SmallInteger*. The largest snippet is 1,279,918 characters long and is a declaration of a *ByteArray* object. Table 5 summarizes the distribution of snippet sizes. We can see by the first quartile (16), median (28) and third quartile (51) that the distribution of the snippet sizes is heavily centred around a much more reasonable size.

Since the maximum is so far away from the third quartile, we assumed there are outliers that need to be excluded. But, our attempt to exclude outliers using one and a half times the interquartile range as the limit marked 10.44% of the data as outliers, and we thus include all the data points.

Figure 5 demonstrates a sorted plot of all the sizes bellow 1000, a total of 90,839 snippets or 97.2% of the data set. The remainder of the set was excluded from the plot because the drastic increase in values made the plot very difficult to understand. We can see from the plot that values are mostly under 100, after which a small number of values rises dramatically.

4.5 Origin of snippets

To better understand our approach and the resulting snippets we look at where the snippets are coming from. Firstly, we wish to understand how many classes in the C' set actually

contributed snippets to the Object Repository. We call these classes "origin classes" and they are members of the C_o set defined in Figure 4, Equation 20. This set contains 9,138 elements, or 65.70% of C'. Manual inspection of a sample of the classes not in this set reveals that they are mostly classes with none or very few declared methods. These are very often meta classes with no functionality outside the trivial instantiation of objects.

We further investigate this set by identifying which classes in this set are meta classes or test classes. We find that meta classes account for 9.78% (894 elements) of C_o that indicates meta classes are less likely to contain snippets, but should not be discarded from the analysis. Test classes account for 16.45% (1,503 elements) of C_o. This initially seems to be not much better than the meta classes but considering the much smaller number of test classes in C' we can see that the contribution of test classes is much greater. Namely, there is a total of 1,797 test classes in C', which means that over 80% of the available test classes contributed an executable node.

We also find that only 112 classes (1.23% of C_o) contributed a snippet that produces an instance of the same class (a snippet that originated in $c \in C'$ and whose execution results in an instance of c). This, coupled with the fact that the remaining 73.77% of C_o are regular classes suggests that the clients of a class is the best place to look for snippets to instantiate that class.

Finally, we aim to answer which types of AST nodes are common sources for snippets. Table 6 shows the percentage of occurrences of each type in N'_{exec} with simple, synthetic examples for easier understanding. As one might expect, the most common type of AST node is the *LiteralValueNode*, as it represents a value in the source code and is thus executable by default. The second most common type is the *MessageNode*, which represents sending a message. This is also typical, since in Smalltalk everything happens by sending messages. Third on the list is *VariableNode*, which in our data denotes global variables *i.e.*, meta classes. The remainder of the list can be divided into two categories: wrappers and literals that we discuss in the following.

The wrappers are *SequenceNode*, *ReturnNode* and *CascadeNode*. The *SequenceNode* represents a sequence of nodes. For instance, the node indexed 4 in Figure 1 represents a sequence of one node indexed 5 and thus yields the same snippet. The *ReturnNode* just adds the return character in front of the node that it is wrapping. In Smalltalk, the last evaluated expression is returned by default and the return statement may not change the result of executing the snippet. For example, the snippets for *LiteralValueNode* and *ReturnNode* in Table 6 have the same execution result. A *CascadeNode* represents a series of message sends to one object. These types of nodes together account for 17.38% of executable nodes.

Node Type	%	Example snippet
LiteralValueNode	33.68%	*'A String'*
MessageNode	22.34%	*Dictionary new.*
VariableNode	17.50%	*Dictionary*
SequenceNode	9.45%	-
ReturnNode	7.59%	*^ 'A String'*
BlockNode	6.20%	*[1 + 1]*
LiteralArrayNode	2.63%	*#(1 2 3)*
CascadeNode	0.34%	*XMLWriter new tag: 'one'; tag: 'two'*
ArrayNode	0.27%	*{1 2 3}*

Table 6. The distribution of types of executable nodes with examples. The *SequenceNode* represents a sequence of other nodes so no example is given.

The literals are *BlockNode*, *LiteralArrayNode* and *ArrayNode*. Closures are very commonly used in Smalltalk and even have their own AST node representation, the *BlockNode*. The other two types of nodes represent a compile time array (*LiteralArrayNode*) and a run time array (*ArrayNode*). These together account for 9.1% of N'_{exec}.

4.6 Failed executions

Studying the reasons why about 90% of AST nodes failed to execute would be necessary towards improving the approach. We primarily hypothesized that a node execution may fail for following reasons:

- Undefined variable in snippet
- Error or exception[5]
- Code snippet returns *nil*

As one might suspect, the execution of the majority of nodes, i.e., over 82%, where prevented because the source code representation of the node failed to compile due to the snippet referring to an unidentified variable.

The other two hypothesised faults are not as numerous. Errors and exceptions account for 1,880 AST nodes (0.12% of the total failing AST nodes), and returning nil accounts for 17,425 AST nodes (1.14% of the total failing AST nodes).

Surprisingly an additional 113,954 nodes (8.31% of the total failing ones) fall outside the hypothesised faults. Manual inspection of a sample of the available logs identifies that the main reason for failure was the snippets expecting interactions from the user *e.g.*, opening a dialog for the user to choose a file. Such attempts were immediately shut down due to our code snippets being executed in a headless Pharo environment, meaning that no GUI elements are possible.

[5] This also includes the nodes terminated by our *timeout* mechanism described in Section 3.2

4.7 Missing classes

To understand why certain classes have no snippets attached to them, we took a sample of 20 such classes and did a manual investigation.

In our sample, 6 classes where test classes. It is not surprising that test classes are never explicitly instantiated, as they are only used by the unit testing framework. Out of all classes with no attached snippets in our data set, around 25% are test classes. Further, 4 classes of our sample where meta classes, and manual inspection shows that these classes, as well as their instances *i.e.*, corresponding non-meta classes, are never used. Looking at all the classes without associated snippets, we find that around 21% are meta classes. The remaining 10 elements of our sample are regular classes, and manual inspection finds that these classes are simply never instantiated. Some are never mentioned in the source code, and some have only class side methods invoked.

5. Related Work

Mining code snippets from existing repositories is not a novel idea and much work has already been done in the field. To the best of our knowledge, this is the first work focusing on mining executable snippets, with a special focus on gathering object creating snippets.

A strong use case for mining snippets in the existing work is to obtain real world examples of API usage. This is typically used to improve documentation or code search engines. Tools such as MAPO [23] focus on mining API methods that are frequently called together and their usages follow sequential rules. Other approaches such as those presented by Buse *et al.* [2] focus on a different kind of static analysis based on combination of path sensitive dataflow analysis, clustering, and pattern abstraction. A tool called PROSPECTOR [14] introduces the concept of *jungloid* source code in an attempt to simplify the mined snippets in order to enable synthesis and combining to form more complex code fragments. Other work, such as that of Ghafari *et al.* [7] focuses on mining examples from unit tests claiming that this is a good source of examples as they are concise, relevant and trustworthy.

Multiple approaches have been proposed which mine relevant code examples and use them to improve code completion tools. Holmes *et al.* propose Strathcona in an attempt to minimise the amount of effort for the developer to query for examples. Bruch *et al.* [1] explore three different strategies for using information gathered from existing code repositories. Their approach, given a set of methods that have been called on a variable and the enclosing method as context, recommend missing method calls for that variable. The PARSEWeb tool [21], rather than performing the analysis of open source systems itself, is build atop of code search engines in order to try to generalise their approach. Pavlinovic *et al.* [18] mines code snippets that occur more than a given threshold in a given code repository, and provides relevant snippets on demand and taking in to account the developers current context. Zhang *et al.* focus on automatically filling the parameter list of API calls automatically [22].

6. Future Work

We identify several directions of potential future work. The main focus of the future work should be bringing the use cases described in Section 2 to fruition, and performing user studies to determine how beneficial the Object Repository would be to developers. This means both improving the approach for building up the object repository which is still in its primary steps, as well as implementing the necessary tools that would serve as the front end facing the developer.

We believe the proposed approach can be improved via applying more solid static analysis techniques such as control and data flow analysis, constant propagation, function inlining and *etc.* For instance, in Section 4.6 we identified that the main reason that executing AST nodes fails is that code snippets contain unidentified variables. In a statically typed language, this could, to a large extent, be addressed by bootstrapping the Object Repository *i.e.*, using the Object Repository to instantiate all undefined variables at the beginning of the snippet, making all undefined variables not just defined, but instantiated. Lacking static type information would make this not so easily applicable, but still possible through type inference [16, 19, 20], or brute force. The downside of this approach is that the instances created would be somewhat synthetic rather than directly pulled from the ecosystem.

Finally, once we have an approach that is shown to be beneficial to developers, we would like to branch out to different programing languages and examine how different language features such as type systems, reflectivity or mode of execution (compiled vs interpreted) affect the benefits or usability of the Object Repository.

7. Conclusion

In this paper we propose the idea of building an Object Repository, a repository of code snippets that, when executed, produce an instance of a class. We present multiple software engineering tasks that could be improved by the Object Repository. We further present an initial attempt to realize the Object Repository for the Pharo language, through mining AST nodes and converting them to code snippets. We applied the approach to 141 open source projects , and discussed the results. Using this approach we could instantiate almost 80% of all classes that were included in the study, however, slightly less than 20% of instantiable classes have more than one associated snippets. Anyways, keeping the simplicity of our approach in mind, the obtained results still seem promising.

We also take a look at the percent of AST nodes that actually produce a type, and discuss the main reasons why other nodes fail to do so. We find that the main reason

for this is unidentified variables, accounting for more than 90% of nodes that failed to produce an instance. This is not unexpected, and can be addressed in several ways, including a bootstrapping of the Object Repository, *i.e.*, using data from a previous run of the Object Repository to instantiate missing variables.

References

[1] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. . URL `http://doi.acm.org/10.1145/1595696.1595728`.

[2] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337316`.

[3] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng. Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, pages 4–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2015-4. URL `http://dl.acm.org/citation.cfm?id=950789.951282`.

[4] A. Chiş, T. Gîrba, O. Nierstrasz, and A. Syrel. GTInspector: A moldable domain-aware object inspector. In *Proceedings of the Companion Publication of the 2015 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH Companion 2015, pages 15–16, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3722-9. . URL `http://scg.unibe.ch/archive/papers/Chis15b-GTInspector.pdf`.

[5] J. W. Duran and S. Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL `http://dl.acm.org/citation.cfm?id=800078.802530`.

[6] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, Oct. 1989.

[7] M. Ghafari, C. Ghezzi, A. Mocci, and G. Tamburrelli. Mining unit tests for code recommendation. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 142–145, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2879-1. . URL `http://doi.acm.org/10.1145/2597008.2597789`.

[8] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0. URL `http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf`.

[9] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. . URL `http://doi.acm.org/10.1145/1062455.1062491`.

[10] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. .

[11] J. Kubelka, A. Bergel, and R. Robbes. Asking and answering questions during a programming change task in the Pharo language. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 1–11, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2277-5. . URL `http://doi.acm.org/10.1145/2688204.2688212`.

[12] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. . URL `http://doi.acm.org/10.1145/1806799.1806829`.

[13] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0547-1. . URL `http://doi.acm.org/10.1145/1937117.1937125`.

[14] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. *SIGPLAN Not.*, 40(6):48–61, June 2005. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/1064978.1065018`.

[15] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, Dec. 2001. ISSN 0098-5589. . URL `http://dx.doi.org/10.1109/32.988709`.

[16] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[17] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. . URL `http://dx.doi.org/10.1109/ICSE.2007.37`.

[18] Z. Pavlinović and D. Babić. Interactive code snippet synthesis through repository mining. Technical Report UCB/EECS-2013-23, EECS Department, University of California, Berkeley, mar 2013.

[19] B. Spasojević, M. Lungu, and O. Nierstrasz. Mining the ecosystem to improve type inference for dynamically typed languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '14, pages 133–142, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1. . URL `http://scg.unibe.ch/archive/papers/Spas14c.pdf`.

[20] B. Spasojević, M. Lungu, and O. Nierstrasz. A case study on type hints in method argument names in Pharo Smalltalk projects. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 283–292, Mar. 2016. . URL http://scg.unibe.ch/archive/papers/Spas16a.pdf.

[21] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. . URL http://doi.acm.org/10.1145/1321631.1321663.

[22] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337321.

[23] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 318–343. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03012-3. . URL http://dx.doi.org/10.1007/978-3-642-03013-0_15.