b
UNIVERSITÄT
BERN

Evaluating the dynamic behavior of Smalltalk applications

Bachelor Thesis

Roger Stebler
from
Balsthal, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

21. April 2015

Prof. Dr. Oscar Nierstrasz
Boris Spasojević

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Contents

Abstract

Since Smalltalk is a dynamically typed programming language it remain agnostic to variable types at compile time, and only take them into account at runtime. This gives more freedom to the developer, as one variable can take on radically different types during program execution. Lack of understanding of dynamic behavior of Smalltalk applications can lead to misleading conclusions about the presents of dynamic features such as duck typing. By instrumenting Smalltalk source code and tracking the different types that variables receive, we concluded that duck-typed variables account for around 1% of total variables. We gather this information using our tool named VariableTracker. We also present an analysis of usage patterns of duck-typed variables from our collected data.

1 Introduction

A type system is a component in programming languages that defines rules how a type can be associated to a variable. It can ensure the absence of erroneous behaviours using the information about types of values that are computed by the program [6]. A fundamental purpose of a type system is to make sure, the program does not do any syntactically or semantically inconsistent operations on the variables, parameters or class members.

Languages are, based on their type system, roughly divided in two groups:

Statically typed languages, such as C, Java or Pascal, are languages that use static type-checking.
The developer is required to specify the type of each variable in the source code. During compilation the source code is verified by a type-checker. This allows the compiler to detect specific type errors during compile time and can reduce the numbers of run time errors.

Dynamically typed languages, such as JavaScript, Python, Ruby or Smalltalk, are languages that use dynamic type-checking.
In dynamically typed languages the developer does not need to specify the type of a variable in the source code. Also the compiler will not do a type-checking process. This behavior gives the developer more freedom because a variable can take completely different types during the execution of the program. The disadvantage is that this can increase the number of run time errors. Also the compiler cannot perform certain optimizations - for example function inlining or more efficient machine instructions - based on knowing the types of variables at compile time.

Let's look at an example in Smalltalk:

```
| a |
a := 42.
a := a + a.
a := 'Hello'.
a := a, ' world'.
a := Dictionary new.
a at: 1 put: 5.
```

This code assigns three values of different types (an `Integer`, a `String` and a `Dictionary`) to a temporary variable `a` and invokes some methods on it. This is syntactically correct Smalltalk code. In a statically typed language that would not be possible. You could define `a` as `Object` to assign values of different types, but the invocation of the methods would result in a compile time error since `Object` does not implement all these methods. But as Smalltalk is a dynamically typed language; it allows us to do so. This is sometimes refereed to as "Variable Reuse".

In 1967 Christopher Strachey distinguished in his lecture notes between two main classes of polymorphism:

The first kind is the *parametric polymorphism* which you get when a function works uniformly on a range of different types which normally all have a common structure.

The second kind is the *ad-hoc polymorphism* where a function can handle different types which don't need to have a common structure. Additionally it may behave in different ways depending on the type of its arguments.

This classification was then refined by Luca Cardelli and Peter Wegner in the paper "On Understanding Types, Data Abstraction, and Polymorphism" [3]. They introduced the differentiation between "universal" and "ad-hoc" polymorphism. One form of universal polymorphism they called *inclusion polymorphism* or also known as *subtype polymorphism* that allows a function which operates on a type `T` also to work on any subtype `S` of `T`.

Another form of polymorphism is called *duck-typing* [8]. This concept describes the type of an object not by its inheritance from a particular class or interface but by its properties and methods. So the type of an object is defined by its behavior. This approach refers to the so called duck-test. James Whitcomb Riley described the duck-test as follows:

> *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."*

Polymorphism is also common practice in statically typed languages. For example in Java generic functions are a type of parametric polymorphism. An example for ad-hoc polymorphism is the concept of function overloading.

For simplicity's sake we will only distinguish between non-polymorphic, subtype polymorphic and potentially duck-typed variables in this thesis.

The questions that we want to answer in this thesis are:

- Do developers use variable reuse?
- If developers use this feature, is it more present in instance variables, method arguments or temporary variables?
- What patterns emerge in variable reuse?

To investigate these questions we first need a way to monitor how developers use variables. To monitor these variable assignments I have written a program called *VariableTracker*. It can be used to instrument Smalltalk code to collect information about a variable every time a value is written to it.

This collected data about variable assignments can then be saved into a database (optional) or directly be analyzed. The analysis provided by VariableTracker will find potentially duck-typed variables *i.e.,* variables whose types don't have a common ancestor in the class hierarchy (except `Object` or `ProtoObject` as

these are common ancestors for all classes in Smalltalk). For example the common superclass of `Float` and `ByteString` is `Object` so we consider them completely different. That means a variable that, at run time, is of both types is potentially duck-typed.

To find patterns in variable reuse we focus on potentially duck-typed variables and ignore variables that have only similar types assigned - for example `Float`, `Double` and `Integer`. Their common superclass is `Number`. These three types are all numbers. So they don't represent real dynamic behavior in a program. These kind of variables represent subtype polymorphism but not duck-typing.

For additional simplicity we ignore the interface of the variable. For example a variable could have values of completely different types assigned but we only invoke methods that are implemented in `Object`. Or there are no method invocations at all on this variable. So we can not be completely sure that a variable is duck-typed. It's just a heuristic to find potential duck-typed variables.

VariableTracker was used in 5 case studies to investigate the variable usage of projects written in Smalltalk:

- Nautilus
- Roassal
- Glamour
- Phratch
- Pangea

In each of those case studies we instrumented the project, collected information about variable usage and analyzed them using VariableTracker to find potentially duck-typed variables. These potentially duck-typed variables were then manually investigated to categorize its kind of usage in different patterns.

2 VariableTracker

2.1 What is VariableTracker?

VariableTracker is a tool used to collect and analyze information about variables being written during run time in Pharo[1] - an open source IDE and virtual machine for Smalltalk which allows us to directly inspect and modify objects and compile code during run time. The information about variables that VariableTracker collects are the class and method where the variable can be found, the type of the value that is assigned, the scope of the variable (temporary, instance variable, method argument) and more. Whenever a value is assigned to a variable VariableTracker will collect this information about the variable and cache it in a dictionary.

This collected data can then be stored into a database or directly be analyzed using the analyzer provided by VariableTracker. The analysis can find variables that are assigned different types during run time. Also it is analyzing the class hierarchy of the type to detect variables using so called "completely" different types - *i.e.,* potentially duck-typed variables.

2.2 How does VariableTracker work?

2.2.1 The Reflectivity framework

VariableTracker is based on the Reflectivity framework. Reflectivity provides the possibility to annotate nodes of an abstract syntax tree with metalinks. That means with this framework we can easily extend and modify an abstract syntax tree. This provides us a simple way to add code to an existing method.

An abstract syntax tree (AST) is a tree representation of the source code of a program. The source code can be divided into different segments that represent the abstract syntactic structure of the source code [5].

[1] `http://www.pharo.org/`

Reflectivity creates a wrapper object of type `RFWrapper` (Reflectivity Framework Wrapper) containing this augmented AST. Installing the wrapper replaces the reference in the method dictionary of the corresponding class with a reference to this wrapper object.

Here an example how Reflectivity can be used to instrument all methods in the class `MyClass`. It adds code to print "variable written" whenever a value is written to a variable:

```
1 MyClass methods do:
2     [ :method |
3         method ast
4             forAllNodes: [ :node | node isAssignment ]
5             putAfter: [ RFMetalink fromExpression:
6                 'Transcript crShow: ''variable written''' ];
7             installWrapper ]
```

First we loop over all methods in `MyClass`. For each of them we get their AST (line 3). The next four lines manipulate that AST. On line 4 we go through all nodes in the AST and every time we find a variable assignment we add the code `Transcript crShow: 'variable written'` after the assignment. A wrapper is generated containing this extended AST which can be used to compile a new method containing the extended code. The generated wrapper replaces the original method by installing the wrapper as seen on line 7 in the example above.

On the first call to this method the virtual machine of Smalltalk detects that there is not an instance of `CompiledMethod` in the method dictionary but a wrapper object. So it sends the message `run:with:in:` to this wrapper. The wrapper will then compile the extended AST, replace itself with this newly compiled method in the method dictionary and resend the message to the method. From now on the new extended method is called and the message about the variable assignment is shown in the Transcript - the default output window in Pharo.

To make collecting the data as fast as possible, we want to compile the extended method immediately during instrumentation and not on the first message send to this method during the collection of the data. To do that we can change the install method of the Wrapper in the following way:

```
1 install
2    "Install the wrapper in the method dictionary to replace the compiled method"
3    self isInstalled ifFalse: [ self install: self ]
```

is changed to:

```
1 install
2    "Compile the method using the extended AST and install it in the method dictionary"
3    self generateCompiledMethod.
4    self installCompiledMethod.
5    self flushCache
```

In this way the new method is not compiled on the first call, but already when the wrapper is installed *i.e.,* when VariableTracker is activated. So when we start collecting the data the original method is already replaced by a `CompiledMethod` containing all code to collect the data and save it in the cache.

We are aware that compiling the extended methods already during the instrumentation can be slower than compiling the methods the first time they are called because there are potentially some methods that are compiled but never used. However, our goal is to improve the performance during the interactive part with the program. This means that we don't want to slow down the system by compiling the methods during the collection of the data.

2.2.2 How does VariableTracker use the Reflectivity framework?

We want the VariableTracker to store information about a variable whenever a value is assigned to it. To do so we can just extend the code from above:

```
1  pvtAddWrapperCode: aMethod
2     aMethod ast
3        forAllNodes: [ :node | node isAssignment ]
4        putAfter: [ :node | RFMetalink fromExpression:
5           'VariableTracker addToCache:',
6              '((Dictionary new)',
7              'at: ''name'' put: ''', node variable name asString, ''';',
8              'at: ''class'' put: self class name asString;',
9              'at: ''method'' put: thisContext method selector asString;',
10             'at: ''isClassSide'' put: self class isClassSide;',
11             'at: ''isInstanceVariable'' put: ', node variable isInstance asString, ';',
12             'at: ''isTemp'' put: ', node variable isTemp asString, ';',
13             'at: ''isArgument'' put: ', node variable isArgument asString, ';',
14             'at: ''isGlobal'' put: ', node variable isGlobal asString, ';',
15             'at: ''type'' put: ', node variable name asString, ' class asString;',
16             'at: ''count'' put: 1;',
17             'yourself).' ];
18       installWrapper.
```

On lines 6-16 we create a `Dictionary` object which stores information about the variable being written. It represents the JSON object that is stored in the database (see section 2.2.5). We pass this dictionary to the method `VariableTracker>>#addToCache:` which will store it in VariableTracker's cache. The Reflectivity framework does the rest. It creates a wrapper containing the extended AST. When the wrapper is installed this extended AST is used to compile a new method which replaces the original one.

2.2.3 What happens to the original methods?

If we want the possibility to deactivate VariableTracker again we need to be able to restore the original method. To do that we must store the references to the original methods so that the garbage collector will not deallocate them. These references are stored in a dictionary called *orginalMethods* in the VariableTracker class just before it is replaced by the wrapper. To deactivate VariableTracker on a method we can just restore the original method by replacing the current instrumented method in the method dictionary of the corresponding class with the reference stored in this dictionary.

2.2.4 Integrating VariableTracker into a development environment

Note: The modifications of the system classes in this section are needed only if we want to be able to change already instrumented methods or if we want to be able to see the source code of instrumented methods. In a normal use case where we just want to collect data we don't necessarily need these modifications.

If we want to change a method that is currently instrumented by VariableTracker the modified method should be instrumented again.

When we save the changed source code of a method the compiler is called and will generate and install a new method. So the following code was added to the compiler method:

```
1     key := self asString, '>>#', method selector asString.
2     (VariableTracker originalMethods includesKey: key)
3       ifTrue: [
4          VariableTracker originalMethods removeKey: key.
5          VariableTracker activateOnMethod: method ].
```

This will check if the method is found in the originalMethods dictionary. If so, the VariableTracker was activated for this method before. After compiling the new source code, VariableTracker will throw away the old original method and activate the tracking for this changed method again. So you can modify instrumented methods without any problems. The tracking is always reactivated on this new method if it was active before.

The other system modification is in the `CompiledMethod` class. We don't want to see the extended source code when we open an instrumented method in a class browser. Independent of the state of VariableTracker the displayed source code should always be the same. So the `sourceCode` method is changed from

```
1  trailer sourceCode ifNotNil: [:code | ^ code ].
```

to

```
1  trailer sourceCode ifNotNil: [:code |
2     (code trim = self codeForNoSource)
3        ifTrue: [
4           key := self methodClass asString, '>>#', self selector asString.
5           (VariableTracker originalMethods includesKey: key)
6              ifTrue: [ code := (VariableTracker originalMethods at: key) sourceCode]].
7     ^ code ].
```

Here we use the originalMethods dictionary again to check if VariableTracker is activated. If we find the method in this originalMethods dictionary we display the code of the original method.

2.2.5 Caching & storing the data in the database

To not collect millions of variable assignments, only one entry is created for each variable - type combination. For example:

Let's assume a variable has the following assignments:

- 20 times an `Integer`
- 5 times a `ByteString`
- 12 times a `Float`

If we would create an entry for every assignment we would have 37 entries. Using variable - type combinations we will just create 3 entries - the ones in the list above.

So we need to check if this variable - type combination was already seen before or if it's the first time. If it's the first time we create a new entry - otherwise we just increase the counter for this combination.

VariableTracker uses an internal cache to store and update these entries. After we have run the instrumented code and collected the data we can directly analyze it using the analysis provided by VariableTracker or we can optionally store the data in a MongoDB database.

If we would directly update these entries in the database without caching it first during the collection we would need to check the database for every assignment if this combination was already seen or not and then update it. This is really inefficient because a communication with an external component is slow. Thus we need a cache to temporarily store the collected data in our system without writing to the external database. This is done using a dictionary where the keys represent the variable - type combination and the

values represent the information that was collected about the variable including a counter that represents the number of times this combination occured.

When the collection of the data is completed we can store the whole content of the cache to the database in one step.

To optionally store the data in the MongoDB database the MongoTalk[2] driver is used. It provides the basic methods to communicate with MongoDB. MongoDB stores everything in JSON format.

JSON (JavaScript Object Notation) is a simple format to interchange data based on JavaScript. Its simple structure allows parsing and generating data in this format in an easy way. Also for humans it's easy to read and understand.[3]

Data in JSON format can be easily represented by nested dictionaries in Smalltalk. For example:

```
{ 'abc' :
    { 'hello' : 'world' }
}
```

is represented by:

```
outerDict := Dictionary new.
innerDict := Dictionary new.
innerDict at: 'hello' put: 'world'.
outerDict at: 'abc' put: innerDict.
```

Calling `VariableTracker>>#saveCacheToDB` will then add each element in the cache to the MongoDB collection:

```
saveCacheToDatabase
   cache values do: [ :entry |
      collection add: entry.
    ]
```

2.2.6 How does the analysis work?

VariableTracker provides a way to analyze the collected data to find potentially duck-typed variables. This section describes how this can be achieved.

If we have stored our collected variable information data in the database, we first we need to load this data from the database back into the cache.

With this information in the cache we can create a `Variable` object for every one of these variable entries. This `Variable` object holds the name of the variable, the class and method where it can be found, a list of types this variable had assigned during the collection of the data and the scope of the variable.

To check if a variable is potentially duck-typed we need the list of types of this variable stored in its `Variable` object. Then we compute the common superclass/ancestor of all these types. If the common superclass is `Object` we can assume that these types are completely different types and that this variable is potentially duck-typed. But if they have a common ancestor deeper in the class hierarchy it means that these types are all similar and this variable is subtype polymorphic and not duck-typed.

[2] `http://smalltalkhub.com/#!/~MongoTalkTeam/mongotalk`
[3] `http://www.json.org/`

Note that this procedure is just a heuristic. Since we don't take invoked methods into account we can not be 100% sure that the types are completely different. There are about 400 common methods implemented in `Object`. For example, every object understands the method `printString` because this method is implemented in the class `Object` and in Smalltalk everything is an `Object`. When we detect a variable with two types in completely different positions of the class hierarchy this heuristic assumes that these types are potentially duck-typed. But it could be that these objects behave completely different and we just invoke a common method inherited from `Object` or `ProtoObject` - or we don't invoke any methods at all.

Some examples for the common ancestor:

Types	Common superclass/ancestor
`Float`, `SmallInteger`, `Double`	`Number`
`ByteString`, `Symbol`	`String`
`ByteString`, `Float`	`Object`

So the first two examples are not potentially duck-typed variables - but the third one is because its type ancestor is `Object`.

To find the common superclass/ancestor of a variable we use the following algorithm:

We initialize the ancestor with the first type found in the list of types contained in the `Variable` object. If there are more types in the list we compare the current ancestor to the second type. Now we have one of the following cases:

- The second type is the same or a subclass of the current ancestor - In this case we don't need to do anything because the current ancestor is already higher in the class hierarchy.
- The second type is higher in the class hierarchy than the current ancestor - In this case we set the second type as the new ancestor.
- The second type is not in the same class hierarchy as the current ancestor - In this case we set the current ancestor to its superclass and check again until we are in the same class hierarchy. That means that current ancestor is now the root of the two subtrees representing the class hierarchy of these types.

When we have found the ancestor of the first two types we can compare this new ancestor to the third type in the list and so on until we have found the common superclass of all types this variable had assigned during the collection of the data.

Using this procedure we can find all potentially duck-typed variables by selecting the ones whose common ancestor is `Object`.

3
Case Studies

We have done 5 case studies by instrumenting common Smalltalk projects using VariableTracker.

The projects are:

- Nautilus[1] - the default system browser since Pharo 2.0
- Roassal[2] - a visualization engine
- Glamour[3] - a framework for browser creation
- Phratch[4] - a platform for kids to learn programming
- Pangea[5] - an analysis tool for OO software corpora

The results of the instrumentation were collected and stored in a MongoDB database. To analyze the data the corresponding method in VariableTracker was used to find potentially duck-typed variables. These results were then manually examined to see how developers use potentially duck-typed variables. After each case study you can find the results and the use patterns we have found.

In the appendix you can find more information about all found potentially duck-typed variables and its types for each case study.

For each case study we first describe the project itself. Then we explain what functionality was executed while the code was instrumented and at the end we show a table containing the results that were gathered by the analysis in VariableTracker. The table contains the following data:

- How many variable assignments were detected during the instrumentation

[1] `http://smalltalkhub.com/#!/~Pharo/Nautilus`
[2] `http://smalltalkhub.com/#!/~ObjectProfile/Roassal`
[3] `http://smalltalkhub.com/#!/~Moose/Glamour`
[4] `http://smalltalkhub.com/#!/~JLaval/Phratch`
[5] `http://smalltalkhub.com/#!/~SCG/Pangea`

- The total number of variables
- How many times on average was each variable assigned to
- The number of subtype polymorphic variables including potentially duck-typed variables
- The number of potentially duck-typed variables
- The total number of different types of all the assigned values

3.1 Nautilus

Since Pharo 2.0 the default browser is Nautilus by Benjamin Van Ryseghem. It can be used to browse the system, create classes, add methods, it provides cool features for refactoring and a lot more. It is probably one of the most used features in Pharo. To collect the data VariableTracker was activated on all packages starting with the name “Nautilus”.

3.1.1 What functionality was executed to exercise the instrumented code?

- Open Nautilus
- Add a new package
- Rename the package
- Delete the package
- Find the Nautilus package
- Create a new class
- Add a method
- Change the method
- Delete the method
- Add a class comment
- Create a method on class side
- Add a protocol
- Drag & drop the method to the protocol
- Rename the protocol
- Rename the method
- Rename the class
- Activate/deactivate groups view
- Activate/deactivate hierarchy view
- Find Nautilus class

- Rename an instance variable in the class Nautilus
- Rename a class variable in the class Nautilus
- Delete my created class
- Run all unit tests
- Close Nautilus

3.1.2 Results

Total variable assignments	68'102
Total variables	466
Average variable assignments	146.14
Total polymorphic variables	55 (11.8%)
Total potentially duck-typed variables	5 (1.07%)
Total different types	300

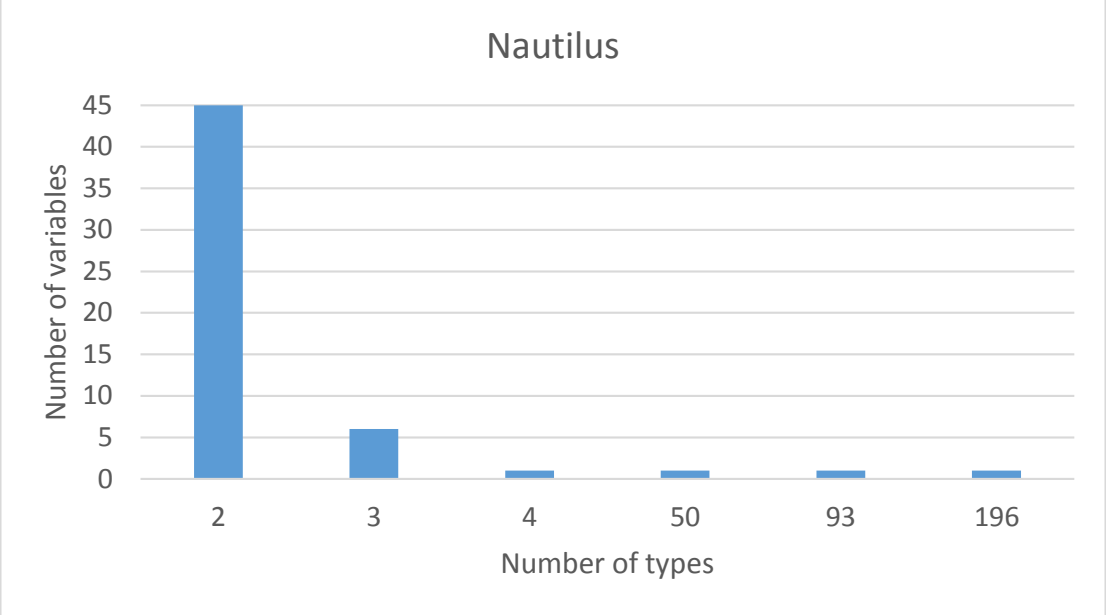

Figure 3.1: How many polymorphic variables have how many different types assigned.

Figure 3.1 shows the distribution of how many polymorphic variables were assigned how many different types in Nautilus.

3.2 Roassal

Roassal is an agile visualization engine for Smalltalk. It provides a large set of interaction facilities like painting, brushing, interconnecting, zooming, drag and dropping.[6]

[6] http://objectprofile.com/Roassal.html

3.2.1 What functionality was executed to exercise the instrumented code?

The data was collected in a Moose image built on 18. July 2014. It included 111 Roassal examples. 3 of them did not work (exception). And 22 were more complex ones that generated a huge amount of data (>1 million variable assignments). So only 2 of these complex examples were executed.

That means in total 88/111 Roassal examples were executed (79%)

What was NOT executed:

- Roassal examples:
  - Graph builder:
    - exampleForceBasedLayoutWithMinSize
    - exampleColoringGraph2
    - exampleColoringGraph3
    - examplePartitions2
    - exampleTreeMapAndNormalizeColor (exception)
    - exampleVisualizationRoassal
    - exampleDependenciesInRoassalAndTrachel
    - examplePartitionedGraph
    - exampleNormalizeColor (exception)
    - exampleConnectFormAndSelectiveColor (exception)
    - exampleLabelledGraph
    - exampleColoredGraph
  - Plain Roassal:
    - exampleGroupOfGroups
    - exampleEdgedWorm
    - exampleMovingGrid
    - exampleLinkingObjects
    - exampleForceBasedLayoutAnimated
    - exampleTimeLineOfRoassal
    - exampleForceBasedLayout
    - exampleVisualizingSoftwareWithBeziers
  - Spectrograph:
    - exampleRoassalVisualization
  - GraphET:
    - exampleScatterPlotLabelled
    - exampleMovies
- Roassal Easel:
  - Bezier Curve
  - Dynamic Force Based Layout
  - Force Based Layout

- Highlight Using Bezier
- Method Complexity
- Step 6
- Step 7
- Step 8
- Uml
- View Tree Map

3.2.2 Results

Total variable assignments	28’290’898
Total variables	2’641
Average variable assignments	10’712.19
Total polymorphic variables	209 (7.91%)
Total potentially duck-typed variables	35 (1.33%)
Total different types	666

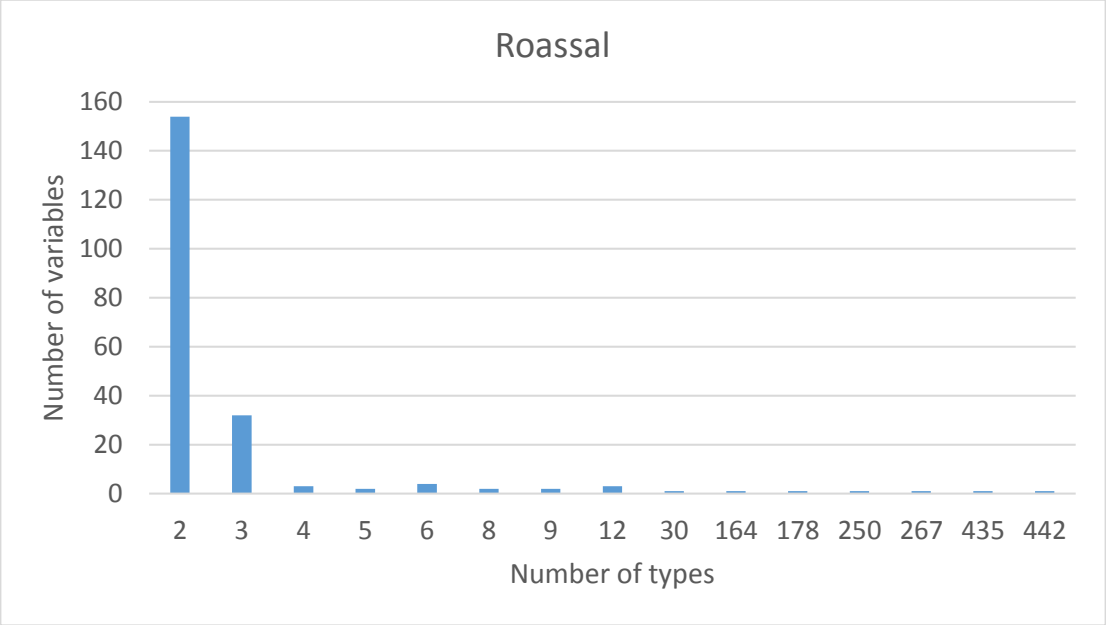

Figure 3.2: How many polymorphic variables have how many different types assigned.

Figure 3.2 shows the distribution of how many polymorphic variables were assigned how many different types in Roassal.

3.3 Glamour

Glamour is an engine for scripting browsers.

It was originally built by Philipp Bunge as a validation for his Masters thesis and Tudor Girba who actively maintains the current version. Andrei Vasile Chiș worked on the Seaside renderer. Lukas Renggli created the first version of the Seaside renderer, Jorge Ressia contributed to the core, and David Röthlisberger worked on the Morphic renderer.[7]

3.3.1 What functionality was executed to exercise the instrumented code?

The data was collected in a Moose image built on 23. July 2014. It included 62 Glamour examples. All of them were executed.

3.3.2 Results

Total variable assignments	1'288'100
Total variables	1'590
Average variable assignments	810.13
Total polymorphic variables	119 (7.48%)
Total potentially duck-typed variables	29 (1.82%)
Total different types	182

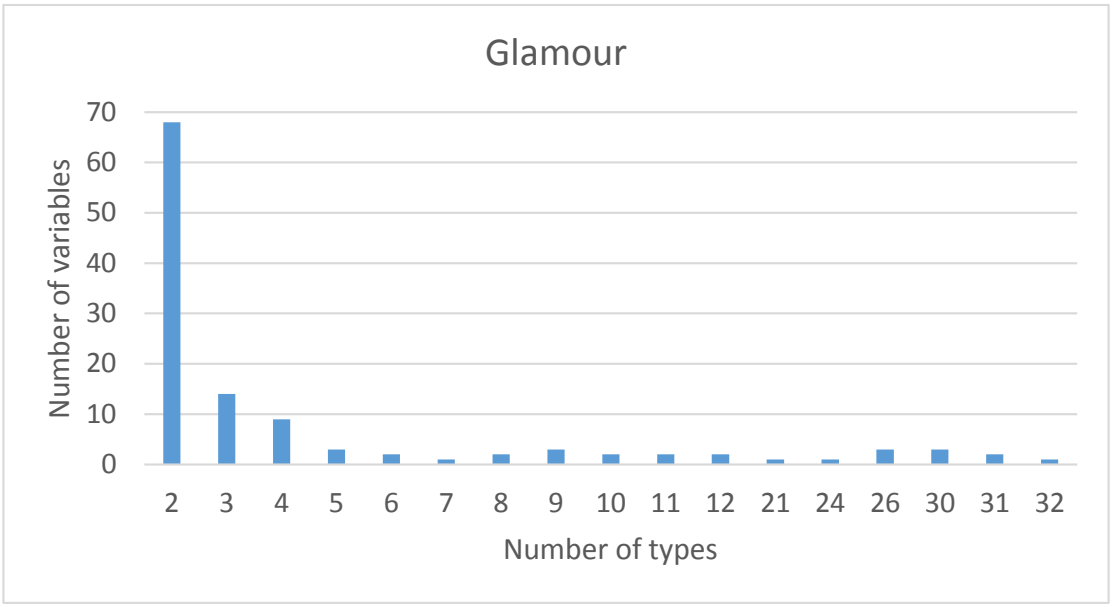

Figure 3.3: How many polymorphic variables have how many different types assigned.

Figure 3.3 shows the distribution of how many polymorphic variables were assigned how many different types in Glamour.

[7] http://gt.moosetechnology.org/

3.4 Phratch

Phratch[8] is a programming language that makes it easy to create your own interactive stories, animations, games, music, and art - and share your creations on the web. It is a port of Scratch (`http://scratch.mit.edu/`) on recent platforms (Pharo 2.0 and Pharo 3.0).

3.4.1 What functionality was executed to exercise the instrumented code?

Several spites and costumes were created. They were used to create and run various scripts. Each script was a random combination of commands selected from the different categories in the GUI of Phratch.

3.4.2 Results

Total variable assignments	9'197'807
Total variables	6'245
Average variable assignments	1'472.83
Total polymorphic variables	553 (8.86%)
Total potentially duck-typed variables	29 (0.46%)
Total different types	211

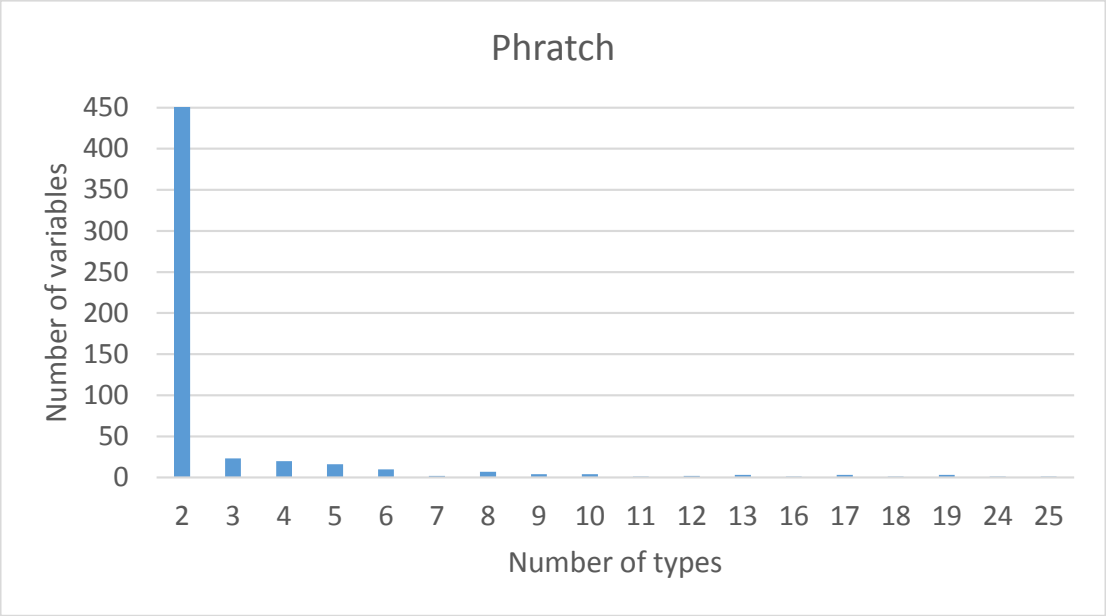

Figure 3.4: How many polymorphic variables have how many different types assigned.

Figure 3.4 shows the distribution of how many polymorphic variables were assigned how many different types in Phratch.

[8] `http://www.phratch.com/`

3.5 Pangea

Pangea[9] enables running language independent analyses on corpora of OO software projects using meta-models stored as object model snapshots.[2]

It's a project by the Software Composition Group of the University of Bern. It currently provides two corpora of software projects - Java: QualitasCorpus 2012 and Smalltalk: Squeak100. Pangea generates a Moose image for every project on the corpus. And Moose is used to analyze software systems by exporting a Famix Metamodel from the source code of said system. So we can run a Moose analysis across a whole corpora of projects.

3.5.1 What functionality was executed to exercise the instrumented code?

The data was collected in a Moose image built on 27. September 2014 using the "ant-1.8.2" Famix meta model from QualitasCorpus. All packages starting with "Famix" and "Moose" were instrumented. Then the three scripts in the paper "Pangea: A Workbench for Statically Analyzing Multi-Language Software Corpora" [2] by Andrea Caracciolo, Andrei Chiș, Boris Spasojević and Mircea F. Lungu were executed.

3.5.2 Results

Total variable assignments	346'430
Total variables	41
Average variable assignments	8449.51
Total polymorphic variables	4 (9.76%)
Total potentially duck-typed variables	0 (0%)
Total different types	21

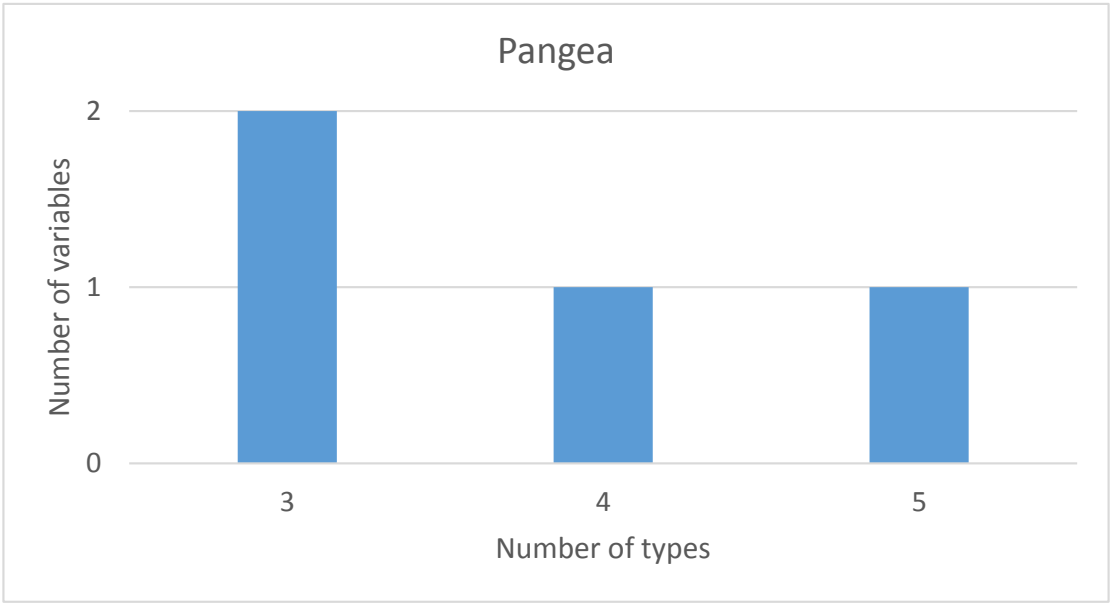

Figure 3.5: How many polymorphic variables have how many different types assigned.

[9] http://scg.unibe.ch/research/pangea

Figure 3.5 shows the distribution of how many polymorphic variables were assigned how many different types in Pangea.

4
Conclusion

We did five case studies to analyze the usage of potentially duck-typed variables. The case studies included five common Smalltalk projects:

- Nautilus - the default system browser since Pharo 2.0
- Roassal - a visualization engine
- Glamour - a framework for browser creation
- Phratch - a platform for kids to learn programming
- Pangea - an analysis tool on OO software corpora

To collect the data we built a tool called VariableTracker to collect data about a variable whenever a value is assigned to it. This data was stored into a database and then analyzed using VariableTracker to find potentially duck-typed variables. All found potentially duck-typed variables were then examined manually and categorized into patterns how the variable is used.

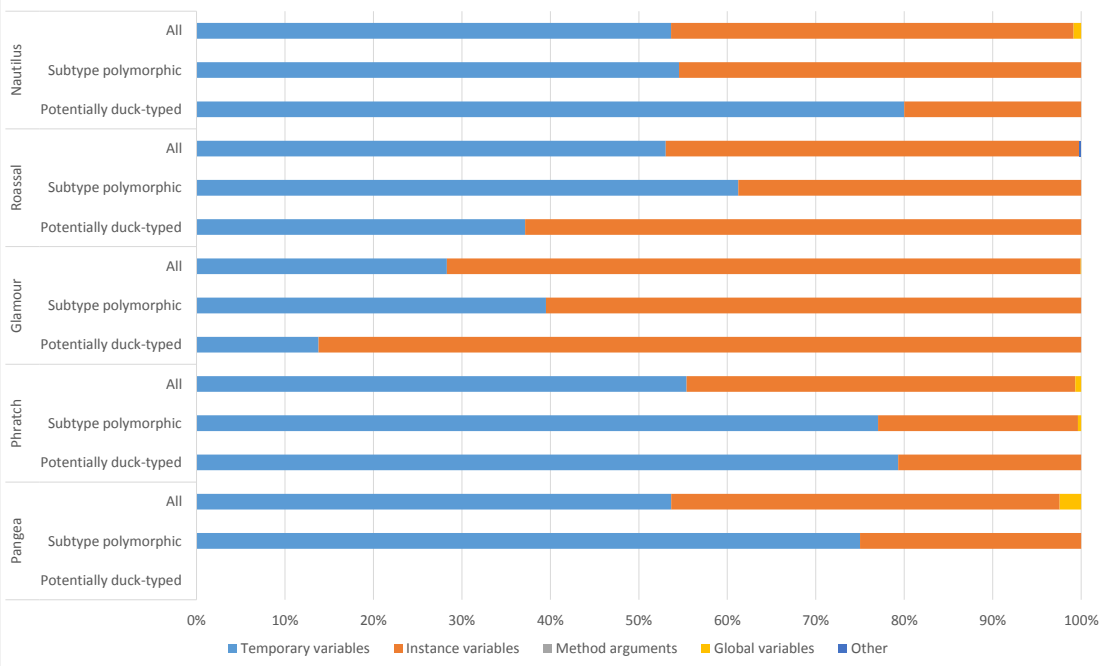

Figure 4.1: Distribution of temporary variables, instance variables, method arguments and global variables for the five case studies

Figure 4.1 shows the distribution of temporary variables, instance variables, method arguments and global variables of the potentially duck-typed variables for each case study. We see that a big part of the variables are temporary or instance variables. But there is no general pattern for the distribution of instance variables and temporary variables between the different projects. Some of them assign more temporary variables and others assign more instance variables. What they have in common is that method arguments, global variables or other kind of scopes are assigned to really rarely.

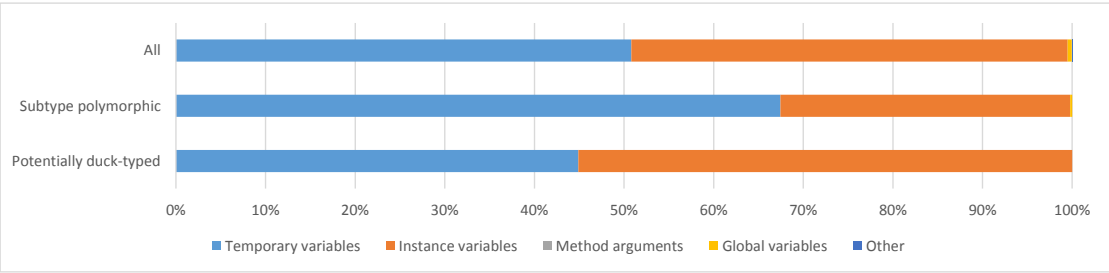

Figure 4.2: Distribution of temporary variables, instance variables, method arguments and global variables on average

Figure 4.2 shows the average distribution of the variable scopes over all the case studies. The number of temporary variables on average is about 51% and the usage of instance variables lies between 48% and 49%. Global variables are assigned in less than 1% of all cases.

Unlike in some other programming languages it is possible in Pharo Smalltalk to assign new values to a method argument. But in our collected data we did not find any variable assignment of this kind at all. It seems developers don't like to modify method arguments.

So we can say that in general about one half of the assigned variables are temporary variables and the other half are instance variables.

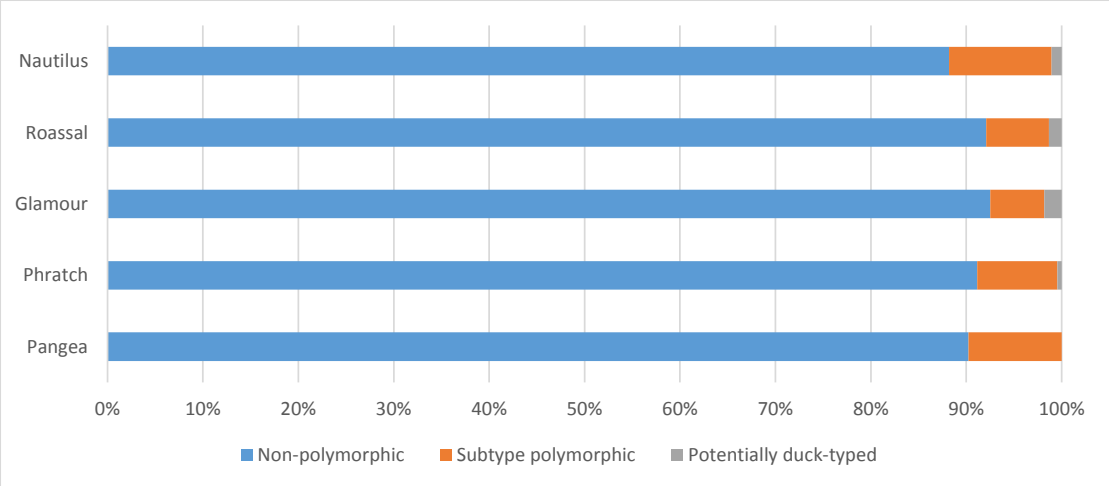

Figure 4.3: Distribution of non-polymorphic, subtype polymorphic and potentially duck-typed variables for the five case studies

Figure 4.3 shows the distribution of non-polymorphic, subtype polymorphic and potentially duck-typed variables for each case study. As in the diagram before we cannot see a general difference between the projects. All distributions are similar.

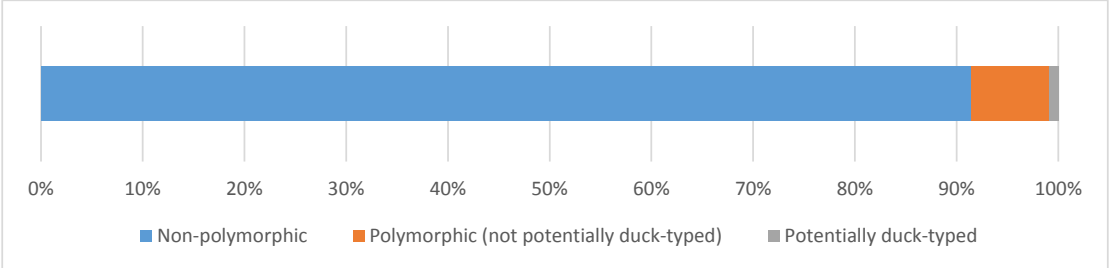

Figure 4.4: Distribution of non-polymorphic, subtype polymorphic and potentially duck-typed variables on average

Figure 4.4 shows the average usage of polymorphism over all the case studies. We can see that about 91% of all variables are non-polymorphic. That means they had only values with the same type assigned during the whole execution. About 8% of the variables are subtype polymorphic. These are the variables that had variables with different types assigned - but the types were all similar. And the part of potentially duck-typed variables - variables that had values assigned with completely different types - is only around

1% on average. As we can see the usage of duck-typing is quite rare.

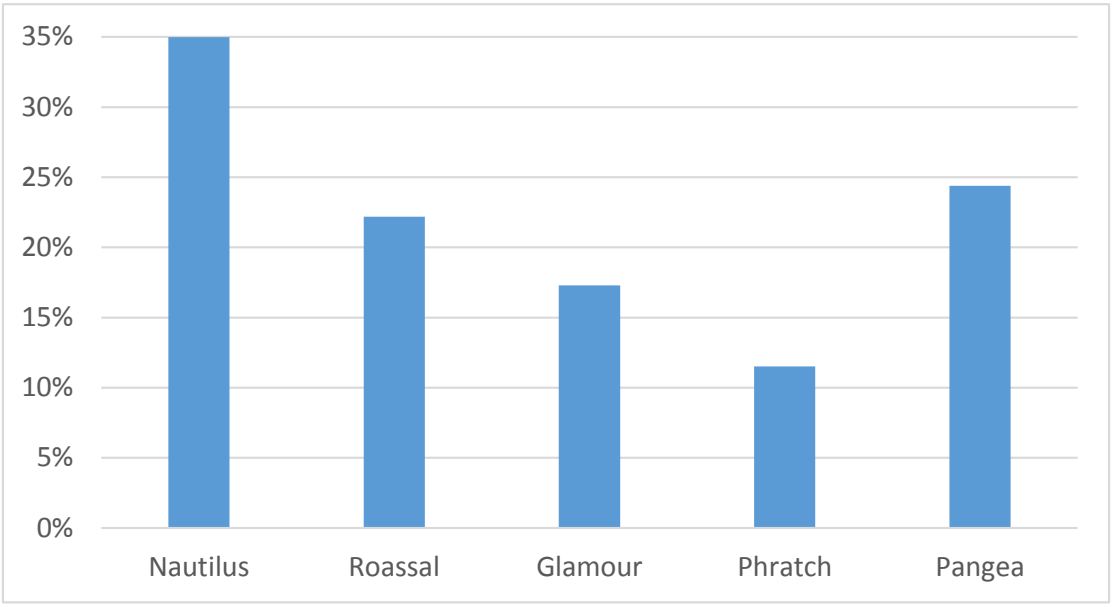

Figure 4.5: Fraction of variables that are only initialized and never reassigned.

Figure 4.5 shows the percentage of variables that are assigned to just once. That means they are only initialized but never reassigned. We can see that about 20% of all the variables are used this way. This kind of variables is often used to improve the readability of the code by splitting a complex code block into different parts, assign the parts to variables and formulate the complex code using the variables, e.g.

```
1 dict := Dictionary new at: 1 put: (Dictionary new at: 1 put: 2; yourself); yourself.
2
3 innerDict := Dictionary new.
4 innerDict at: 1 put: 2.
5 dict := Dictionary new.
6 dict at: 1 put: innerDict.
```

The first line is equivalent to lines 3-6. But the second code part has a better readability than the first one. Here the variable `innerDict` is only initialized but never reassigned.

In these 5 case studies we can say that 4 of them are rather development projects (Nautilus, Roassal, Glamour and Pangea) and the only real application project is Phratch. Looking at the results regarding this categorization we cannot find any difference between development and application projects. So we can assume that the developers of development projects use variable assignments and polymorphism in a similar way as the developers of application projects.

4.1 Patterns

After we found all the potentially duck-typed variables we examined them manually and tried to categorize them by how they are used. The following list shows the patterns we have found. To make the patterns

more clear we will look at an example for each pattern. The examples are real ones found in our case studies.

- **semantics are same, different APIs** - the variable takes objects that are semantically the same but use a different API.

  Example: In Roassal we find the method `RTMetricNormalizer>>normalizePosition:min:max:using:` that contains the following code:

```
1  ...
2     minValue := 1000000.
3     maxValue := -1000000.
4     elements do: [ :el |
5        | t |
6        t := transformation rtValue: (metricBlock rtValue: el model).
7        minValue := minValue min: t.
8        maxValue := maxValue max: t ].
9     (maxValue - minValue) ~= (0 @ 0) ifTrue: [
10 ...
```

  Here we want to normalize the position of the Roassal elements. We loop over all the Roassal elements and get its position - a `Point` - and update the current `minValue` and `maxValue`. Because we initialized `minValue` and `maxValue` with `1000000` and `-1000000` these variables in lines 7 and 8 can have two different types: `SmallInteger` and `Point`. Points and Numbers are very similar because a point is represented by two Numbers. So we have here a different API but the semantic is the same.

- **object or `BlockClosure`** - the variable takes objects and code blocks

  Example: For this pattern we will look at a basic example not related with the case studies.

```
1 coll := OrderedCollection new.
2 ...
3 coll collect: [ :each | each asFloat ].
4 coll collect: #asFloat
```

  Here lines 3 and 4 are equivalent. We can send a `BlockClosure` or a `ByteSymbol` to this method `collect:` and in both cases we get a collection containing the numbers as `Float`. So in this category we will collect variables that take objects - in this case a symbol - and code blocks.

- **any object as data model** - the variable takes objects that serve as data model

  Example: Roassal has the method `ROElement>>model:` containing:

```
1 model := anObject
```

  Whenever Roassal wants to visualize an object then it creates a Roassal element of type `ROElement` for this object. The method above is then called to store a reference to the object which is then used as data model for this Roassal element.
  Let's assume we want to visualize all classes in the system and the connections to their subclasses. So we can create a Roassal element for each class and each of them holds a reference to the actual class in the system. When we want to visualize the edges which represent the connections between a class and its subclasses, Roassal needs a way to find the Roassal elements that need to be connected. That means it needs to be able to find the Roassal elements that represent a specific object. This is simply done by comparing the object with the `model` variable of all the Roassal elements. In this way Roassal can find all the elements which represent the subclasses of a specific class and then

draw a connection to them.
So Roassal can use any object as data model and interactions with that model are given through metaprogramming.

- **collection or single element** - the variable takes a whole collection of objects or just a single element

  Example: Roassal has a method `ROTreeMapLayout>>prepareGraph:` that contains the following code:

```
1 roots := ROGraphTransformation new fromEdgesToNesting: nodeCollection edges: edges.
2
3 (roots isKindOf: Collection) ifFalse: [
4    roots := OrderedCollection with: roots.
5 ].
```

  When Roassal needs to create a tree map it asks for the root nodes of the tree. If we have multiple roots we get a collection of the roots. But if we only have one root we don't get a collection containing only this node but we get the node object itself. So the code checks if this `roots` variable is a collection and if not then it creates a new collection containing this root node.

- **real usage of potential duck-typing** - the variable can have completely different types from different parts of the inheritance tree. Since we don't investigate the invoked methods in this thesis it is still only a potential usage of duck-typing. Additionally it does not fit the other patterns.

  Example: An example from a method in Phratch:
  `PhratchUpdatingStringMorph>>readFromTarget`

```
1 | v |
2 (v := self valueFromTargetOrNil) ifNil: [^ contents].
3 lastValue = v ifTrue: [^ contents].
4 lastValue := v.
5 ^ self formatValue: v
```

  This method is called when a string morph is updated. E.g. a progress bar changes its value so this method is called to also update the text representation of the progress. Or a color picker that displays the name or the hex represenation of the selected color. On line 3 the method compares the current value `v` with the stored `lastValue` to see if it's needed to update the text. This variable `lastValue` does not store the string representation itself but the object that is represented. E.g. if the color is changed it does not store the color name as `String` but the `Color` object itself. So this variable `lastValue` can have different types assigned. There are no methods invoked on the variable; it's only stored for comparison. Since we ignore the interface of a variable in this thesis this variable `lastVariable` meets the conditions to count as a real usage of potential duck-typing.

- **potential wrong usage** - the variable takes values of types that do not make any sense regarding the name of the variable and the other types assigned.

  Example: In Nautilus we find the method `MethodWidget>>methodIconFor:` which contains the following line:

```
1 button := action actionIcon.
```

  Nautilus uses this method to display an icon next to a method in the method browser. Some of those icons are clickable (e.g. overriding/overridden arrows) and some are not (e.g. halt flag or no icon). So the type of the values assigned to this variable called `button` can be an `IconicButton` (clickable), a `ColorForm` (e.g. flags, not clickable) or `Form` (no icon). This can be confusing

since this variable `button` is not always used as a real button. In duck-typing the type of an object is defined by its behavior. But the behavior is not the same for all these icons; some of them are clickable and others are not. There is no run-time error, so it's still duck-typing. But because of the confusion we will categorize this example as potential wrong usage.

In the following sections we show the patterns for each project. The first column in the table is the pattern and the second column shows how many of the potentially duck-typed variables use this pattern.

4.1.1 Patterns for Nautilus

semantics are same, different APIs	4
potential wrong usage	1

4.1.2 Patterns for Roassal

object or `BlockClosure`	19
semantics are same, different APIs	8
collection or single element	4
any object as data model	2
real usage of potential duck-typing	2

4.1.3 Patterns for Glamour

object or `BlockClosure`	10
real usage of duck-typing	10
any object as data model	6
potential wrong usage	3

4.1.4 Patterns for Phratch

real usage of potential duck-typing	27
any object as data model	2

4.1.5 Patterns in total

real usage of potential duck-typing	39
object or `BlockClosure`	29
semantics are same, different APIs	12
any object as data model	10
collection or single element	4
potential wrong usage	4

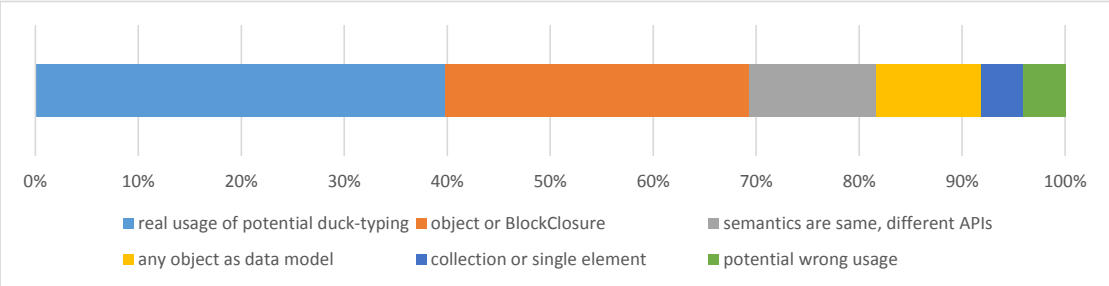

Figure 4.6: Distribution of use patterns for potentially duck-typed variables

In figure 4.6 we can see that about 40% of all potentially duck-typed variables use real duck-typing. Another 30% handle objects or code blocks, about 12% take objects that have a similar semantic but a different API, 10% of the potentially duck-typed variables are used as data model, about 4% handle single elements and collections and the other 4% are potential wrong usages.

4.2 Summary

In this thesis we evaluated the dynamic behavior of Smalltalk applications. We have written a tool called VariableTracker that can collect information about variables being written during the run time of a Smalltalk application. VariableTracker also allows us to do a basic analysis of the collected data to find potentially duck-typed variables. Optionally the collected data can be stored in an external MongoDB database.

This tool was then used in 5 case studies of common Smalltalk applications. The results show that almost all of the variables that are assigned are temporary or instance variables. On average about one half of the collected data are temporary and the other half are instance variables. Global variables or method arguments are only assigned in very rare cases.

More than 90% of the variables are non-polymorphic. About 9% are subtype polymorphic and only about 1% are potentially duck-typed. So the usage of duck-typing is quite rare.

These potentially duck-typed variables were then manually examined and categorized into 6 different patterns of how the variable is used. About 40% of the variables use real duck-typing. The rest is used in other kinds of duck-typing.

4.3 Threats to validity

4.3.1 External validity

In this thesis we evaluated the dynamic behavior of Smalltalk applications. Since we need to run all the programs to do a dynamic analysis we can only analyze a limited number of projects. So we decided to do 5 case studies. We tried to select projects that are often used in a Smalltalk environment. To not only have development but also an application project one of our case studies included the analysis of Phrach.

In a static analysis - as done by Oscar Callaú *et al.* (see Related work) - it would be possible to analyze a much bigger number of projects but we would lose a lot of information about the real dynamic behavior of the program.

4.3.2 Internal validity

The categorization of the potentially duck-typed variables to different usage patterns was done manually. In some cases the assignment to a specific category was not completely clear. It would be possible to categorize them to multiple patterns. So we added them to the most reasonable category from our point of view. Another person using the same data could get a slightly different result regarding the distribution of the usage patterns.

5
Related work

There exist a number of empirical studies about static and dynamic analysis of software projects.

One of the most related works is the statical analysis of the usage of polymorphism in Smalltalk by Oscar Callaú *et al.* [1] They did a static analysis on the 1000 largest projects found in Squeaksource. They found out that dynamic features are rarely used and they are more used in specific kinds of projects such as core system libraries and development tools than in normal applications. Also the most used dynamic features are supported by other static languages such as Java.

Another related work is an analysis of the dynamic behavior of JavaScript programs by Gregor Richards *et al.* [7] Their results show that libraries often change the built-in prototypes in JavaScript to add behavior to types. Also their assumption that the use of `eval` is infrequent and primarily used for deserialization turned out to be wrong.

There exists also an analysis about the dynamic behavior of Python applications by Alex Holkner and James Harland.[4] They concluded that most of the dynamic behavior - such as adding properties to an object - is done at the initialization phase of the object. And during the rest of the object lifetime they are more or less used in a static way.

Bibliography

[1] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: The case of Smalltalk. In *Proceedings of the 8th working conference on Mining software repositories (MSR 2011)*, pages 23–32, New York, NY, USA, 2011. IEEE Computer Society.

[2] Andrea Caracciolo, Andrei Chis, Boris Spasojević, and Mircea Lungu. Pangea: A workbench for statically analyzing multi-language software corpora. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 71–76. IEEE, September 2014.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[4] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pages 19–28. Australian Computer Society, Inc., 2009.

[5] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[6] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[7] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010.

[8] Andrew Hunt and David Thomas. Programming ruby: The pragmatic programmer's guide. In *New York: Addison-Wesley Professional.*, 2, 2000.

A

Anleitung zu wissenschaftlichen Arbeiten

This tutorial explains how we can use VariableTracker to instrument the code we are interested in to collect and optionally store data about the variable usage in a MongoDB database and then analyze it using the analysis provided by VariableTracker.

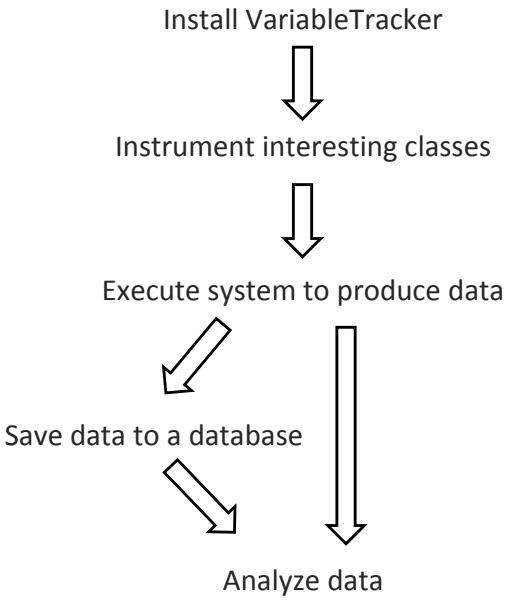

Figure A.1: Usage flow how to use VariableTracker

In figure A.1 you get a general overview how VariableTracker is used. After the data was collected we have the possibility to store it to an external database or we can directly analyze it.

Preparation

The source code for VariableTracker is hosted on SmalltalkHub [1].

It has some dependency projects:

- Reflectivity [2] - Used to hook the method when a variable is written
- MongoTalk [3] (optional) - Driver to talk to the MongoDB database
- MongoDB [4] (optional) - The database where the information about the variables are stored

MongoTalk and MongoDB are optional. The collected data is first cached in a dictionary. The analysis of the data can also directly be done using the cache - without storing the data to a database. If we don't want to store the data in a database we can skip the steps to set up a connection to the database or load it from the database back into our Smalltalk image.

Collect the data

After we have installed VariableTracker and its dependencies we are ready to collect variable type data.

To activate VariableTracker we can use one or multiple of the following commands:

```
1 VariableTracker activateOnClass: MyClass.
2 VariableTracker activateOnMethod: (MyClass>>#doSomething).
3 VariableTracker activateOnPackageName: 'Nautilus'.
4 VariableTracker activateOnPackagesStartingWith: 'Nautilus'.
```

The VariableTracker provides an interface to enable tracking all variables of a whole class, of a specific method or on complete packages. It's also possible to combine any of the commands above.

Now, every time a tracked variable is assigned, information about this variable is collected and cached.

As we have seen in section 2.2.5 about how the caching and storing of the data works we only collect variable - type combinations instead of creating an entry for every variable assignment. If we don't want this behavior but want an entry for each assignment we can change that by executing:

```
1 VariableTracker collectAll: true
```

To stop VariableTracker from collecting data we can just call the corresponding deactivation methods.

```
1 VariableTracker deactivateOnClass: MyClass.
2 VariableTracker deactivateOnMethod: (MyClass>>#doSomething).
3 VariableTracker deactivateOnPackageName: 'Nautilus'.
4 VariableTracker deactivateOnPackagesStartingWith: 'Nautilus'.
```

[1] http://smalltalkhub.com/#!/~rostebler/VariableTracker
[2] http://smalltalkhub.com/#!/~RMoD/Reflectivity
[3] http://smalltalkhub.com/#!/~MongoTalkTeam/mongotalk
[4] http://www.mongodb.org/

Interaction with the database

If we want to store the data to a MongoDB database we first need to establish a connection to the database. After we are sure MongoDB is installed and running on our system and we have loaded MongoTalk from Smalltalk Hub in our image we can connect to the database using the command:

```
VariableTracker connectToDefaultDB
```

This will create a database and a collection both called 'VariableTracker' (if not existing yet) and connect to it.

If we want to connect to another database or another collection we can use one of the following commands:

```
VariableTracker connectToDB: dbName
VariableTracker connectToDB: dbName collection: collectionName
VariableTracker connectToDB: dbName collection: collectionName host: hostname
VariableTracker connectToDB: dbName collection: collectionName host: hostname port: port
VariableTracker connectToDefaultDBWithCollection: collectionName
```

After we have a connection to our database we can store the content of the cache to the database:

```
VariableTracker saveCacheToDB
```

If we want to load the data from the database to the VariableTracker cache and we are connected to the database then we can run the following command:

```
VariableTracker loadDataFromDB
```

This will load all the data in the database into the Pharo image and store it in the VariableTracker cache.

Analyze the data

To analyze the collected/loaded data we need the `VariableTrackerAnalyzer` class. We open a Transcript and then run the command:

```
VariableTrackerAnalyzer analyze
```

The data is now being analyzed and we can see a summary in the Transcript.

Additionally an inspector of a collection of all subtype polymorphic variables and another inspector of a collection of all potentially duck-typed variables is opened. We can use these inspectors to further examine the variables and see the whole data that was collected about a variable.

B

Raw data from the case studies

Nautilus

Variable	Types	Count
SortHierarchicallyNode>>setElement:>>element	195 different class types	1475
	Trait	10
NautilusRefactoring>>class:andInstVariable:>>class	RBClass	1
	Nautilus class	1
MethodWidget>>methodIconFor:>>button	IconicButton	9
	Form	61
SortHierarchically class>>sortNodes:>>superior	49 different class types	463
	SortHierarchicallyNode	463
PackageTreeNautilusUI>>listSelection2At:>>elt	Trait	10
	92 different class types	1703

Roassal

Variable	Types	Count
RTLabelled>>initializeElement:>>t	SmallInteger	5
	248 different class types	248
	ByteString	3
RTAnimatedScatterPlot>>x:>>x	SmallInteger	1
	BlockClosure	8
	ByteSymbol	3
RTAnimatedScatterPlot>>y:>>y	SmallInteger	1
	BlockClosure	4
	ByteSymbol	3

ROEdge>>model:>>model	Association	110
	SmallInteger	732
	ROElement	40
	161 different class types	2167
ROLabel>>color:>>color	BlockClosure	1
	Color	111
ROLabel>>textFor:>>v	SmallInteger	98
	27 different class types	75
	ByteSymbol	22
	ByteString	292
RTMetricNormalizer>> normalizePosition:min:max:using:>>minValue	SmallInteger	5
	Point	157
RTMetricNormalizer>> normalizePosition:min:max:using:>>maxValue	SmallInteger	5
	Point	157
ROBox>>borderColor:>>borderColor	BlockClosure	2
	Color	294
ROBox>>color:>>color	BlockClosure	353
	Color	302
ROLine>>width:>>strokeWidth	SmallInteger	1
	BlockClosure	2
ROLine>>color:>>color	BlockClosure	84
	Color	367
ROElement>>model:>>model	BlockClosure	9
	ByteString	109
	Association	1
	ByteSymbol	37
	ROSelection	1
	171 different class types	2543
	Array	4
	SmallInteger	1597
RTShowEdge>>shape:>>shape	BlockClosure	1
	RTLine	1
	RTLine class	250
RTSugiyamaLayout>>reduceCrossing>>v	RODummyNode	29
	RTElement	532
RTSugiyamaLayout>>reduceCrossing>>u	RODummyNode	3
	RTElement	558
RTSugiyamaLayout>>addDummyNodes>>fromNode	RODummyNode	3
	RTElement	99
RTMenuBuilder >> menu:submenu:background:callback:>>parentMenu	TRLabelShape	69
	OrderedCollection	69
RTGDLayoutB>>if:>>condition	True	22
	BlockClosure	6
RTLinearMove>>to:during:on:>>element	TRLabelShape	50
	RODummyNode	3
	RTElement	1062
RTShapeBuilder>>if:fillColor:>>oldBlockOrValue	BlockClosure	3

	Color	2
RTElement>>model:>>model	ByteSymbol	13
	CompiledMethod	460
	Color	335
	Float	46
	Character	26
	Array	1043
	433 different class types	5501
	SmallInteger	3263
	Association	165
	ByteString	502
ROAthensCanvas>>canvas:>>nativeCanvas	AthensCairoCanvas	320
	RONullCanvas	79
RTGDEdgeB>>connectTo:>>connectTo	BlockClosure	2
	ByteSymbol	8
RTGDEdgeB>>initialize>>condition	True	13
	BlockClosure	13
RTGDEdgeB>>createEdgesFor:>>toObjects	Array	5116
	266 different class types	1078
RTGDEdgeB>>createEdgesFor:>>fromObjects	Array	5124
	434 different class types	5124
ROTreeMapLayout>>prepareGraph:>>roots	ROElement	3
	OrderedCollection	8
ROMondrianShapeBuilder>>color:>>color	BlockClosure	1
	Color	428
ROMondrianShapeBuilder>>if:borderColor:>> oldBlockOrValue	BlockClosure	1
	Color	1
ROEllipse>>color:>>color	BlockClosure	7
	Color	272
RTBezierLine>>controllingElements:>> controllingElements	Array	1
	BlockClosure	6
	RTGroup	2
ROMondrianFrame>>layout:>>layout	ROTreeLayout	50
	ROScatterplotLayout	1
	RONullLayout class	1
	RODominanceTreeLayout	2
	ROVerticalLineLayout	1
	ROTreeMapLayout	18
	ROBottomFlowLayout	1
	ROFlowLayout	1
	ROHorizontalTreeLayout	6
	ROCircleLayout	16
	ROGridLayout	13
	ROHorizontalLineLayout	319
RTMapBuilder>>countries:named:>>countryNames	BlockClosure	2
	ByteSymbol	1
RTMapBuilder>>color:>>color	BlockClosure	2
	Color	1

Glamour

Variable	Types	Count
GLMTextPresentation>>transformation:>> transformation	BlockClosure	40
	ByteString	4
GLMTextPresentation>>displayValue>> cachedDisplayedValue	GLMAnnouncingCollection	27
	SmallInteger	8
	ByteString	39
	ByteSymbol	5
	Interval	129
	Text	180
	Array	2
	OrderedCollection	10
	Character	23
GLMGenericAction>>action:>>action	GLMPortUpdater	25
	BlockClosure	2389
GLMTableColumn>>computation:>>computation	BlockClosure	289
	ByteSymbol	20
GLMTableColumn>>title:>>title	BlockClosure	1
	ByteString	308
GLMSingleUpdateAction>>transformation:>> transformation	BlockClosure	3
	Announcer	9
	ByteSymbol	12
GLMMagrittePresentation>>displayValue>> cachedDisplayedValue	GLMMagrittePersonExample	5
	GLMAnnouncingCollection	2
LazyTabPage>>labelMorph:>>labelMorph	AlphaImageMorph	1
	ByteSymbol	5
	ByteString	721
LazyTabPage>>lazyPageMorphCreation:>> lazyPageMorphCreation	BlockClosure	724
	PanelMorph	3
GLMSmalltalkCodePresentation>>displayValue>> cachedDisplayedValue	MethodContext	29
	ByteString	3
GLMSmalltalkCodePresentation>>transformation:>> transformation	BlockClosure	71
	ByteString	1
GLMListPresentation>>displayValue>> cachedDisplayedValue	GLMAnnouncingCollection	64
	Interval	67
	Array	52
	LinkedList	12
	OrderedCollection	59
	Character	6
GLMListPresentation>>title:>>title	BlockClosure	3
	ByteString	90
GLMListPresentation>>transformation:>> transformation	BlockClosure	155
	ByteSymbol	31
GLMTransmission>>value>>originalValue	ByteString	29
	GLMCompositePresentation	8
	GLMMultiValue	120
	GLMBasicExamples class	1

	MethodContext	174
	ByteSymbol	49
	GLMTextPresentation	21
	GLMAnnouncingCollection	19
	GLMTabulator	10
	SmallInteger	380
	Pragma	69
	LinkedList	24
	GLMMagrittePersonExample	4
	GLMFormatedPresentation class	1
	BlockClosure	7
	Character	97
	OrderedCollection	10
	DebugSession	58
	GLMWrapper	9
	Text	182
	Array	44
	Interval	154
	Association	6
	GLMActionListPresentation	21
GLMTransmission>>meetsCondition>>originValues	ByteString	21
	GLMCompositePresentation	6
	GLMMultiValue	145
	GLMBasicExamples class	1
	MethodContext	145
	ByteSymbol	5
	GLMTextPresentation	18
	GLMAnnouncingCollection	13
	GLMTabulator	8
	SmallInteger	194
	GLMMagrittePersonExample	2
	BlockClosure	6
	Character	25
	OrderedCollection	10
	DebugSession	58
	GLMWrapper	6
	Text	176
	Array	38
	Interval	152
	Association	3
	GLMActionListPresentation	18
GLMTransmission>>transformation:>>transformation	BlockClosure	89
	ByteSymbol	67
GLMPortEvent>> initializeOn:previouslyValued:in:>>oldValue	GLMCompositePresentation	3
	ByteString	34
	GLMDynamicPresentation	1
	GLMMultiValue	97
	MethodContext	29
	ByteSymbol	39

	GLMTextPresentation	23
	GLMTablePresentation	6
	GLMTreePresentation	19
	GLMTabulator	3
	GLMSmalltalkCodePresentation	1
	SmallInteger	444
	GLMAnnouncingCollection	9
	Pragma	137
	LinkedList	53
	GLMMagrittePersonExample	3
	BlockClosure	3
	Character	145
	OrderedCollection	306
	GLMWrapper	3
	GLMListPresentation	28
	Text	1786
	Array	7
	Interval	1282
	Association	2
	GLMActionListPresentation	9
GLMTreeMorphSelectionChanged>> selectionValue:>>selectionValue	MethodContext	58
	ByteSymbol	47
	OrderedCollection	19
	GLMFormatedPresentation class	1
	Character	53
	Pragma	69
	LinkedList	14
	Array	3
	GLMAnnouncingCollection	6
	SmallInteger	160
	Association	3
GLMTreeMorphStrongSelectionChanged>> strongSelectionValue:>>strongSelectionValue	GLMAnnouncingCollection	2
	ByteSymbol	15
	Pragma	2
	LinkedList	8
	SmallInteger	60
	Character	21
GLMCompositePresentation>>title:>>title	BlockClosure	20
	ByteString	7
GLMTreePresentation>>displayValue>> cachedDisplayedValue	Array	18
	Interval	17
	Character	4
	LinkedList	1
GLMTreePresentation>>transformation:>> transformation	Array	3
	BlockClosure	19
GLMMorphPresentation>>displayValue>> cachedDisplayedValue	SmallInteger	1
	ImageMorph	5
GLMMorphicMagritteRenderer>> magritteMorphFrom:>>toShow	GLMMagrittePersonExample	5

	GLMAnnouncingCollection	2
GLMContextChanged>>oldValue:>>oldValue	ByteString	97
	GLMCompositePresentation	3
	GLMDynamicPresentation	1
	GLMMultiValue	97
	MethodContext	29
	ByteSymbol	74
	GLMTextPresentation	42
	GLMTablePresentation	12
	GLMTreePresentation	48
	GLMTabulator	3
	GLMSmalltalkCodePresentation	3
	SmallInteger	856
	GLMAnnouncingCollection	16
	Pragma	274
	LinkedList	139
	GLMMagrittePersonExample	5
	BlockClosure	3
	Character	302
	OrderedCollection	633
	GLMWrapper	3
	GLMListPresentation	50
	Text	2978
	Array	22
	Interval	2363
	Association	4
	GLMActionListPresentation	9
GLMContextChanged>>value:>>value	GTSimpleMethodsBrowser	4
	GLMCompositePresentation	33
	ByteString	173
	GLMDynamicPresentation	3
	GLMMultiValue	304
	GLMBasicExamples class	4
	MethodContext	493
	ByteSymbol	219
	GLMTextPresentation	148
	GLMTablePresentation	12
	GLMTreePresentation	67
	GLMTabulator	38
	GLMSmalltalkCodePresentation	3
	SmallInteger	1676
	GLMAnnouncingCollection	62
	Pragma	280
	LinkedList	198
	GLMMagrittePersonExample	12
	GLMFormatedPresentation class	4
	BlockClosure	28
	Character	496
	GTGenericStackDebugger	58

	OrderedCollection	905
	DebugSession	116
	GLMListPresentation	71
	GLMWrapper	34
	Text	4028
	Array	155
	Interval	2933
	Association	12
	GLMActionListPresentation	86
GLMPanePort>>silentValue:>>value	GTSimpleMethodsBrowser	2
	GLMCompositePresentation	12
	ByteString	64
	GLMDynamicPresentation	1
	GLMMultiValue	114
	GLMBasicExamples class	2
	MethodContext	232
	ByteSymbol	105
	GLMTextPresentation	59
	GLMTablePresentation	6
	GLMTreePresentation	24
	GLMTabulator	15
	GLMSmalltalkCodePresentation	1
	SmallInteger	826
	GLMAnnouncingCollection	28
	Pragma	140
	LinkedList	72
	GLMMagrittePersonExample	6
	GLMFormatedPresentation class	2
	BlockClosure	12
	Character	219
	GTGenericStackDebugger	29
	OrderedCollection	427
	DebugSession	58
	GLMListPresentation	35
	GLMWrapper	13
	Text	1934
	Array	67
	Interval	1398
	Association	6
	GLMActionListPresentation	34
GLMPanePort>>changeValueTo:in:>>oldValue	GLMCompositePresentation	3
	ByteString	34
	GLMDynamicPresentation	1
	GLMMultiValue	97
	MethodContext	29
	ByteSymbol	39
	GLMTextPresentation	23
	GLMTablePresentation	6
	GLMTreePresentation	19

	GLMTabulator	3
	GLMSmalltalkCodePresentation	1
	SmallInteger	444
	GLMAnnouncingCollection	9
	Pragma	137
	LinkedList	53
	GLMMagrittePersonExample	3
	BlockClosure	3
	Character	145
	OrderedCollection	306
	GLMWrapper	3
	GLMListPresentation	28
	Text	1786
	Array	7
	Interval	1282
	Association	2
	GLMActionListPresentation	9

Phratch

Variable	Types	Count
ArgumentPlaceHolderMorph>>mouseDown:>>frag	BlockLabelFragment	2
	ByteSymbol	2
ReporterBlockMorph>>defaultArgs:>>defaultValue	PhratchSpriteMorph	13
	ByteSymbol	168
	ByteString	318
	ScriptablePhratchMorph	1
	SmallInteger	129
ReporterBlockMorph>>showValue>>msg	Float	2
	ByteString	17
	PhratchListMorph	2
	Form	2
	SmallInteger	7
PhratchStageMorph>>changeVar:by:>>n	SmallInteger	5
	ByteString	2
PhratchSpriteMorph>>lookLike:>>i	SmallInteger	37
	ByteString	1
PhratchProcess>>returnValueToParentFrame:>>obj	Color	15
	PhratchSpriteMorph	630
	ByteSymbol	3
	ByteString	111
	PhratchStageMorph	1
	Float	5
	SmallInteger	426
PhratchProcess>>evaluateSelfEvaluating>>value	SmallInteger	421
	Color	15
	ByteSymbol	3
	ByteString	98

PhratchProcess>>evaluateFor:>>expression	SpecialBlockMorph	1642
	ByteSymbol	2
	ChoiceOrExpressionArgMorph	3
	ReporterWatcherBlockMorph	22
	CustomCommandBlockMorph	8
	ExpressionArgMorphWithMenu	7
	SetterBlockMorph	52
	ChoiceArgMorph	31
	CommandBlockMorph	1729
	ReporterBlockMorph	43
	ExpressionArgMorph	477
	CBlockMorph	104
	Array	915
	TimeBlockMorph	1081
	ColorArgMorph	15
	SpriteArgMorph	3
	AttributeArgMorph	2
PhratchProcess>>popStackFrame>>command	SpecialBlockMorph	14
	ChoiceOrExpressionArgMorph	3
	ByteSymbol	1710
	ReporterWatcherBlockMorph	22
	CustomCommandBlockMorph	4
	ExpressionArgMorphWithMenu	7
	SetterBlockMorph	26
	ChoiceArgMorph	31
	CommandBlockMorph	1236
	ReporterBlockMorph	24
	Array	245
	CBlockMorph	16
	ExpressionArgMorph	477
	TimeBlockMorph	18
	SpriteArgMorph	3
	AttributeArgMorph	2
	ColorArgMorph	15
PhratchProcess>>applyPrimitive>>value	PhratchSpriteMorph	630
	ByteString	13
	PhratchStageMorph	1
	SmallInteger	5
	Float	5
PaintFrame>>scaleCanvas:>>numToScale	SmallInteger	3
	Float	1
	ByteSymbol	2
CommandBlockMorph>>argMorphToReplace:>>v	SmallInteger	2
	ByteString	4
CommandBlockMorph>>defaultArgs:>>defaultValue	SmallInteger	629
	ByteString	680
PhratchCommandHandler>>model:>>model	ScriptablePhratchMorph	139
	PhratchStackFrame	8

	PhratchSpriteMorph	3247
	PhratchStageMorph	10
ReporterWatcherBlockMorph>>showValue>>msg	SmallInteger	1
	Float	1
	ByteString	2
VariableBlockMorph>>receiver:>>receiver	CustomBlockDefinition	7
	PhratchSpriteMorph	8
	PhratchStageMorph	5
ListContentsBlockMorph>>showValue>>msg	PhratchListMorph	1
	Form	1
PhratchStackFrame>>expression:>>expression	SpecialBlockMorph	1642
	ChoiceOrExpressionArgMorph	3
	ByteSymbol	2
	ReporterWatcherBlockMorph	22
	CustomCommandBlockMorph	4
	ExpressionArgMorphWithMenu	7
	SetterBlockMorph	26
	ChoiceArgMorph	31
	CommandBlockMorph	1247
	ReporterBlockMorph	24
	Array	258
	CBlockMorph	96
	ExpressionArgMorph	477
	TimeBlockMorph	20
	ColorArgMorph	15
	SpriteArgMorph	3
	AttributeArgMorph	2
PhratchEvent>>name:argument:>>argument	SmallInteger	24
	KeyboardEvent	126
WatcherMorph>>translatedName>>param	SmallInteger	1
	ByteString	14
ColorBlockMorph>>defaultArgs:>>defaultValue	SmallInteger	16
	ByteString	4
PhratchFrameMorph>> watcherShowingFor:selectorAndArg:>>arg	SmallInteger	14
	Color	28
	ByteString	250
BooleanBlockMorph>>showValue>>msg	True	3
	False	7
	ByteString	20
TimeBlockMorph>>defaultArgs:>>defaultValue	SmallInteger	152
	ByteString	54
BooleanWatcherBlockMorph>>showValue>>msg	False	1
	ByteString	2
DoAsk>>model:>>model	ScriptablePhratchMorph	1
	PhratchStackFrame	4
	PhratchSpriteMorph	13
PhratchUpdatingStringMorph>> valueFromTargetOrNil>>result	Color	3474
	False	1861
	ByteSymbol	8400

	ByteString	21250
	Float	92854
	SmallInteger	29941
PhratchUpdatingStringMorph>>readFromTarget>>v	Color	3474
	False	1861
	ByteSymbol	8400
	ByteString	21250
	Float	92858
	SmallInteger	29941
PhratchUpdatingStringMorph>>readFromTarget>> lastValue	Color	1
	False	2
	ByteSymbol	2
	ByteString	37
	Float	1156
	SmallInteger	12711

Pangea

No potentially duck-typed variables found.