It's Duck (Typing) Season!

Nevena Milojković, Mohammad Ghafari, Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
{nevena, ghafari}@inf.unibe.ch
scg.unibe.ch

***Abstract*—Duck typing provides a way to reuse code and allow a developer to write more extensible code. At the same time, it scatters the implementation of a functionality over multiple classes and causes difficulties in program comprehension.**

The extent to which duck typing is used in real programs is not very well understood. We report on a preliminary study of the prevalence of duck typing in more than a thousand dynamically-typed open source software systems developed in Smalltalk.

Although a small portion of the call sites in these systems is duck-typed, in half of the analysed systems at least 20% of methods are duck-typed.

***Keywords*-duck typing, dynamically-typed languages, cross-hierarchy polymorphism**

I. Introduction

Duck typing was named after the "duck test", by James Whitcomb Riley: "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"[1]. Duck typing, also known as cross-hierarchy polymorphism, refers to the implementation of methods that have the same signature and belong to distinct classes that do not have a common superclass defining the method with the same signature [1]. It indicates that one variable may point to the objects whose types are unrelated, *i.e.,* do not have a common parent understanding the methods invoked on that variable. For example, any object that provides array-like operators such as indexing, *i.e.,* `[]`, can be used in any code where an array is expected. These additional methods are *duck-typed*. Duck typing is usually encountered in dynamically-typed programming languages, although it can be simulated also in statically-typed languages with the use of interfaces.

Duck typing leads to functionality being scattered throughout the code, which hinders program comprehension [2]. Combined with method overriding, it heavily aggravates program comprehension in dynamically-typed languages [3], [4]. For instance, when programmers read a piece of code, they typically need to make assumptions about the possible types of the variables used in it. Even though static type information would help a lot, duck typing introduces false positives in the result of type inference algorithms [5].

We now define terms that will be used in the rest of the paper. We say that a method is *duck-typed* if it has the same signature as another method, neither of which overrides a method with the same signature of a common parent. We define $intr(s)$ as the set of duck-typed methods with the same signature s. We call the cardinality of this set the *duck-typing degree* of signature s. A call-site is *duck-typed* if it is the call site of a duck-typed method. We define the degree of a duck-typed call site as the number of duck-typed methods corresponding to that call site.

In the light of the impact that duck typing has on program comprehension, we investigate the prevalence of duck typing in a large corpus of open source dynamically-typed software. We pose the following research questions:

- RQ1) How diffuse are *duck-typed* methods and what is the degree of the corresponding method signatures?
- RQ2) How widely present are *duck-typed* call sites and what is the degree of these call sites?

We analyse a corpus of more than 1000 software systems developed in Smalltalk. Our analysis is static, *i.e.,* we do not analyse the amount of duck typing at run time.

Structure of the Paper. We first define the terminology in section II. Next we introduce our experimental methodology and the analysis infrastructure in section III. In section IV we report on our findings, and discuss the threats to validity in section V. We present the related work in section VI before concluding the paper in section VII.

II. Terminology

In order to precisely define duck typing as we measure it, we introduce the following simple set-theoretical model.

$$sig_{cs} : CS \to S \quad (1) \qquad sup : C \to C \cup \{\bot\} \quad (4)$$
$$sig_{m} : M \to S \quad (2) \qquad mets : C \to \mathcal{P}(M) \quad (5)$$
$$def_{m} : M \to C \quad (3)$$

Given all source code of a system, C is the set of all classes that are defined locally in the system, M the set of all methods. S is the set of all signatures. *CS* is the set of all method call sites in the system.

Each method call site cs has the signature, $sig_{cs}(cs)$ (1). Each method m has a unique signature $sig_m(m)$ (2), and is defined in a unique class $c = def_m(m)$ (3). Class c either has a unique superclass $c' = sup(c)$ (4) or does not have a superclass, *i.e.,* $sup(c) = \bot$. Each class c has a set of methods that it defines $mets(c)$ (5).

We can now query the model to compute the metrics necessary to answer our research questions.

[1] https://en.wikipedia.org/wiki/Duck_test

$$grok(c, s) = s \in sig_m(mets(c)) \vee grok(sup(c), s) \quad (6)$$

$$intr(s) = \{m \in M |\ sig_m(m) = s \ \wedge \neg grok(sup(def_m(m)), s)\} \quad (7)$$

$$duck\text{-}typed(s) = |intr(s)| > 1 \quad (8)$$

$$duck\text{-}typed(cs) = |intr(sig_{cs}(cs))| > 1 \quad (9)$$

We use the function $grok(c, s)$ to determine whether the class c understands the signature s, either because it defines a method m such that $sig_m(m) = s$ or inherits it from a superclass (6). We define $grok(\perp, s) = false$, for any signature s. We also use the function $intr(s)$ to find all methods "introducing" the signature s, *i.e.,* methods m such that $sig_m(m) = s$ and not overriding the method with the same signature (7).

A signature s is *duck-typed* if there are at least two classes in the system defining methods that introduce signature s *i.e.,* signature s is not introduced by any superclass of these classes (8). Consequently, elements of the set $intr(s)$ are *duck-typed* methods and its cardinality represents the degree of the signature s.

A call site cs is *duck-typed* if its signature $sig_{cs}(cs)$ is duck-typed (9). The degree of call site cs in regard to duck typing is equal to $|intr(sig_{cs}(cs))|$.

III. Experimental Setup

We analysed more than one thousand projects in Smalltalk [6]. We chose Smalltalk as it is "pure" object-oriented dynamically-typed language. Since it was designed in the '80s, the practice in duck typing usage should be well established. Thus, the age of subject systems should not influence the results. We study the projects in the SqueakSource repository used to host the majority of open-source projects from both industry and academia. The projects in this representative repository are also used in an earlier study on polymorphism presence in the Smalltalk [7]. In order to exclude student and toy projects, we opt only for projects containing more than 50 classes. Out of 1850 cloned projects, we include 1,128 projects in this study. These projects contain 125,825 classes and 1,637,228 methods in total.

To analyse the code, we employed Ecco and Monticello as parsers [8]. We traverse the body of every class in the system, and collect the set of signatures that the class introduces: $\{s \in S | s \in sig_m(mets(c)) \wedge grok(sup(c), s) = false\}$. We then construct the set of methods in a system that have duck-typed signature, and measure the corresponding degree, *i.e.,* $|intr(s)|$ for a signature s. Afterward, we traverse the body of each method to collect the call sites cs for which $duck\text{-}typed(cs) = true$, as well as their degree, *i.e.,* $|intr(sig_{cs}(cs))|$.

IV. Experimental Results

Our study reveals that 99% of the inspected Smalltalk projects define and call duck-typed methods. We present a detailed explanation of our findings in the following section.

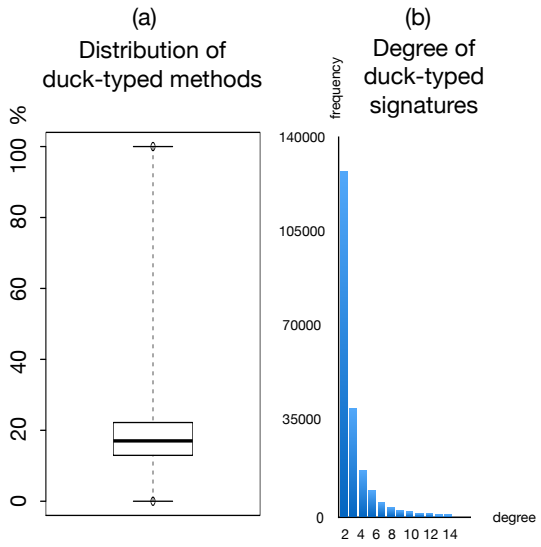

Fig. 1. Implementing duck-typed signatures in Smalltalk

A. Implementing duck typing

Figure 1 (a) shows the proportion of methods whose signatures are duck-typed. While there are outlier projects with more than 50% of methods being duck-typed, we generally observe that:

- for half of the projects, at least one out of six methods (17%) is duck-typed
- most of the analysed projects (about 75%) contain up to 22% of methods being duck-typed

We have also measured the degree of each of the duck-typed signatures, *i.e.,* $|intr(s)|$. The results are presented in Figure 1 (b). We can observe that:

- 75% of duck-typed signatures are implemented in up to five distinct hierarchies, *i.e.,* have a degree of up to five
- some outliers have more than 200 cross-hierarchy implementations

Based on this data, we can answer the first research question:

> In half of the projects more than one out of six methods are duck-typed, and 75% of the duck-typed methods have degree of up to five.

B. Using duck typing

Figure 2 (a) presents the proportion of the call sites that are duck-typed, *i.e.,* the proportion of ducked-type method calls. We establish that the average number of call sites in a project is 6500. In few projects, more than 18% of call sites are duck-typed which is surprising, but apart from this minority, we can state that:

- In half of the analysed projects, between 0.55% and 15% of call sites are duck-typed

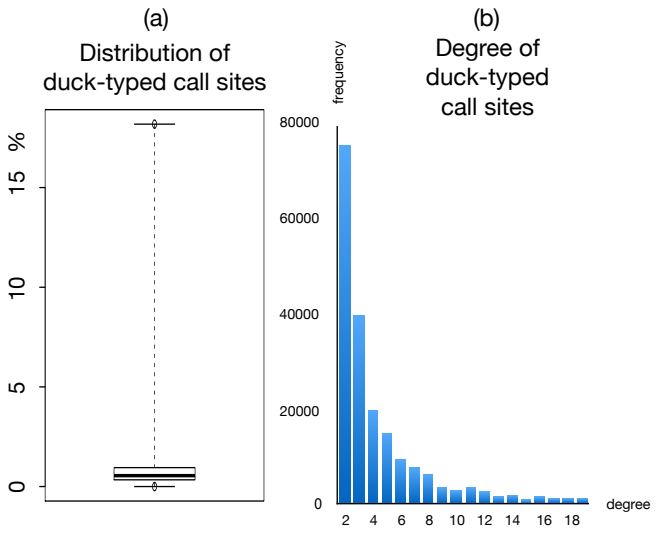

Fig. 2. Using duck-typed signatures in Smalltalk

- In around 20% of the analysed projects, between 1% and 15% of call-sites are duck-typed

We have also measured the degree of duck-typed call sites, *i.e.,* $|intr(sig_{cs}(cs))|$. Our findings are presented in Figure 2 (b). Disregarding the outliers with degree of more than 200, we observe that:

- 37% of duck-typed call sites have degree of two
- about 30% of duck-typed call sites have degree of three or four
- about 28% of call sites have degree varying from five to 17

We can now answer the second research question accordingly:

> All projects contain to some extent duck-typed call sites. 80% of them have up to 1% of duck-typed call sites. Degrees of these call-sites vary from two to more than 200.

C. Discussion

The average degree of a duck-typed call site is six. While the degree of duck typing only considers newly introduced duck-typed signatures, we do not explicitly explore subclasses overriding these methods. Since we do not consider subtype polymorphism in this study, this means that number of method candidates for these call sites may be even larger due to method overriding.

When analysing a call site that has commonly used method signature, a developer is often reluctant to explore all the method candidates statically, due to the possibly long list of selector implementors [4]. This forces her to run the code, if possible, in order to obtain the desired information. Static type information would probably eliminate the cost of running the software [9]. Yet, duck typing, together with subtype polymorphism are frequently labeled as one of the biggest obstacles for static type analysis and type inference in particular [5].

We deem important, as future work, to perform empirical studies that would explore in depth and quantify the actual impact that duck typing has on program comprehension. As static type information facilitates program analysis [10], we think it is necessary to explore ways to improve simple type inference algorithms, which suffer greatly in the presence of (cross-hierarchy) polymorphism.

Most of the analysed duck-typed signatures have degree of up to five, that is, five independent class hierarchies define the same method signature. IDE tools for dynamically-typed languages usually employ the same algorithm that we have used in this study [11], hence developers need to navigate to several method definitions in unrelated classes in order to understand the execution flow. We propose the need for tool support in the presence of duck typed methods specialised in displaying up to five method implementations. This would be useful, as developers would not have to open many navigation windows and pollute working space [12]. Consequently, it would decrease the number of navigation actions.

V. Threats to Validity

We have used a very simple algorithm based only on the method signature to detect duck typing. Our analysis may have over-estimated the actual presence of duck typing, and it may be that more precise analysis would provide different results. The best way to justify the results would be to use dynamic analysis. However, collecting run-time information for the studied corpus would be impractical due to the large number of analysed projects.

Due to the experimental setup, in our analysis we do not consider methods that are inherited and overridden from outside a project, *i.e.,* from libraries. This may produce false positives in the case of methods that appear to be duck-typed, but actually inherit from a common parent outside the studied project.

Since we analysed only open-source projects, we cannot generalise our findings to proprietary software systems. Finally, this study was dedicated to Smalltalk, and studying the presence of duck typing in other dynamically-typed languages may yield other insights due to different coding idioms.

VI. Related Work

Duck typing has been analysed on a set of 36 Python programs [13]. Based on the recorded run-types of variables, the authors state that most variables are monomorphic, *i.e.,* point to objects of only one type at run time. However, most of the rest of the variables do point to the objects of unrelated types, *i.e.,* duck typing is used at run time.

We are not aware of any large-scale study concerning the prevalence of duck typing. The most similar studies are those regarding polymorphism usage in object-oriented code.

A recent study analysed polymorphism usage in Smalltalk and Java, and found that polymorphism is more prevalent in

Smalltalk than in the Java corpus, and that it is omnipresent in both analysed corpora [7].

One of the first conducted studies is about the relationship between code maintainability and the depth of inheritance. Daly *et al.* [14] found that inheritance facilitates code maintenance: a system with three levels of inheritance is easier to understand than an equivalent system with no inheritance. Cartwright *et al.* confirmed the positive impact of inheritance on maintenance time [15], while Harrison *et al.* discovered that a system without inheritance is easier to maintain than the equivalent systems with more than two levels of inheritance [16].

Tempero *et al.* performed a study about the usage of inheritance in Java corpus [17]. Their work showed a high usage of inheritance and the variation in the use of inheritance between interfaces and classes. They also studied method overriding in Java open source projects [18]. Their study showed that most subclasses override at least one method and many classes only declare overriding methods.

VII. Conclusion

We performed an empirical study of duck typing on the corpus of open source systems written in Smalltalk. We found that nearly all projects define and use duck-typed methods.

We found that in half of the projects at least 17% of methods are duck-typed, and that most of the corresponding signatures have degree of up to five. These methods usually count for less than 1% of call sites, but there are projects with almost one fifth of duck-typed call sites, most of which have degree of up to 17.

In order to precisely measure duck typing usage, we intend to preform dynamic analysis on the part of the studied projects.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018). We also gratefully acknowledge the financial support of the Swiss Group for Object-Oriented Systems and Environments (CHOOSE) and the European Smalltalk User Group (ESUG). We thank David Röthlisberger, Romain Robbes, Mircea Lungu and Andrea Caracciolo for implementing the infrastructure used for analysis.

References

[1] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, 3rd ed. Pragmatic Bookshelf, 2009.

[2] A. Dunsmore, M. Roper, and M. Wood, "Object-oriented inspection in the face of delocalisation," in *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*. ACM Press, 2000, pp. 467–476.

[3] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Trans. Softw. Eng.*, vol. 34, pp. 434–451, Jul. 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1446226.1446241

[4] 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*, ser. PLATEAU '14. New York, NY, USA: ACM, 2014, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2688204.2688212

[5] N. Milojković and O. Nierstrasz, "Exploring cheap type inference heuristics in dynamically typed languages," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016*. New York, NY, USA: ACM, 2016, pp. 43–56. [Online]. Available: http://scg.unibe.ch/archive/papers/Milo16b.pdf

[6] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*. Reading, Mass.: Addison Wesley, May 1983. [Online]. Available: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[7] N. Milojković, A. Caracciolo, M. Lungu, O. Nierstrasz, D. Röthlisberger, and R. Robbes, "Polymorphism in the spotlight: Studying its prevalence in Java and Smalltalk," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 186–195, published. [Online]. Available: http://scg.unibe.ch/archive/papers/Milo15a.pdf

[8] R. Robbes, M. Lungu, and D. Roethlisberger, "How do developers react to API deprecation? The case of a Smalltalk ecosystem," in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, 2012, pp. 56:1 – 56:11. [Online]. Available: http://scg.unibe.ch/archive/papers/Rob12aAPIDeprecations.pdf

[9] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *IEEE Computer*, vol. 31, no. 3, pp. 23–30, Mar. 1998. [Online]. Available: http://www.cs.indiana.edu/classes/c102/read/Ousterhout.pdf

[10] S. Spiza and S. Hanenberg, "Type names without static type checking already improve the usability of APIs (as long as the type names are correct): An empirical study," in *Proceedings of the 13th International Conference on Modularity*, ser. MODULARITY '14. New York, NY, USA: ACM, 2014, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/2577080.2577098

[11] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009. [Online]. Available: http://pharobyexample.org

[12] D. Röthlisberger, O. Nierstrasz, and S. Ducasse, "Autumn leaves: Curing the window plague in IDEs," in *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 237–246. [Online]. Available: http://scg.unibe.ch/archive/papers/Roet09fAutumnLeaves.pdf

[13] B. Åkerblom and T. Wrigstad, "Measuring polymorphism in Python programs," in *Proceedings of the 11th Symposium on Dynamic Languages*, ser. DLS 2015. New York, NY, USA: ACM, 2015, pp. 114–128. [Online]. Available: http://doi.acm.org/10.1145/2816707.2816717

[14] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "Evaluating inheritance depth on the maintainability of object-oriented software," *Empirical Software Engineering*, vol. 1, no. 2, pp. 109–132, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF00368701

[15] M. Cartwright, "An empirical view of inheritance," *Information and Software Technology*, 1998.

[16] R. Harrison, S. Counsell, and R. Nithi, "Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems," *J. Syst. Softw.*, vol. 52, no. 2-3, pp. 173–179, Jun. 2000. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(99)00144-2

[17] E. Tempero, J. Noble, and H. Melton, "How do Java programs use inheritance? An empirical study of inheritance in Java software," in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ser. ECOOP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 667–691. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70592-5_28

[18] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, Dec. 2010, pp. 336 –345.