Towards Generating Software Systems From Legacy Code: Gathering Associations From Smalltalk Software

Jan Blizničenko and Robert Pergl

Faculty of Information Technology, Czech Technical University in Prague,
Thákurova 9, Prague, 16000, Czech Republic
{jan.bliznicenko,robert.pergl}@fit.cvut.cz

Abstract. Generating programming code out of models is a useful approach towards maintainable software systems, yet legacy systems have often no models at all. There are multiple ways to partially automate or aid the process of generating models out of legacy code, yet there is one kind of object-oriented programming languages substantially harder to transform – dynamically typed languages. While statically typed languages enforce programmers to explicitly state the data types of various elements, dynamically typed languages do not, presenting significant difficulties when gathering associations between classes. This paper presents an ongoing effort towards dealing with gathering associations by combining various type inference techniques and tools and how the authors aim to use UML models as a transition form between origin and destination programming languages, producing UML models as useful byproducts. Pharo – Smalltalk-based dynamically typed programming language is used as a case study.

Keywords: software models, generating, reverse-engineering, type inference

1 Introduction & Motivation

Modeling software systems before implementation has been recommended for several decades [1, 3] and general model-driven development as well as Normalized Systems theory [7] advises automated or semi-automated ways of generating programming code out of models [20, 27]. There are several kinds of models, like Unified Modeling Language – UML [18] – for various aspects of the software, BPMN for business processes or Normalized Systems models allowing building modular and well-maintainable software. Furthermore, such models are being used to automatically generate programming code and even whole systems.

Unfortunately, there are still too many recent or legacy software systems of various proportions without any models and diagrams at all. Such systems can be found in any programming language and that is why there are tools for automating the other way around - generating UML-like diagrams from the code itself. Generated models and diagrams are not as useful and clear as hand-made

conceptual-level diagrams, yet even the generated models and diagrams provide a general overview that even well-commented text files with code lack.

With the ability to generate UML models and generate code out of these models, the long-term goal is to find a way to transform legacy software systems into modern technologies with as much automation as possible.

To generate UML models, there are several tools available for statically typed programming languages like C++, Java, C#, and many others. These tools are able to at least provide UML Class Diagram-like diagrams and some of them even generate actual UML models by specification. Gathering associations and data types does not pose any significant problem for such languages, because all types are explicitly stated in the code itself.

Unlike statically typed languages, dynamically typed ones, like Python, Ruby, or Smalltalk, do not enforce types of variables, method arguments, and return values and therefore do not explicitly state data types in the code. For example, whenever a variable is introduced, there are no checks whether a programmer assigns an integer, a date, or a person object to the variable. To generate associations out of attributes and to put types into method signatures in UML structural models, type inference techniques are needed to extract data types out of the code.

In this paper, the term *association* is used for binary associations defined in OMG UML [6], including aggregations and compositions, but not including generalizations.

Smalltalk is a dynamically typed language used for over 40 years in various industrial applications and has several implementations as well as Smalltalk-based languages, like Pharo, that is used as a case study for all aspects of our work.

With the ability to generate UML models from legacy software systems created in discontinued object-oriented programming languages and the ability to generate code of recent programming languages, the process of transforming legacy software systems to current technology could be automated or, at least, improved.

1.1 Type Inference

Type inference is a process of finding data types in a code that does not specify them. It can be used in both statically and dynamically typed languages.

Statically Typed Languages In statically typed languages, the data type information of variables, method parameters, and return types must be known during compilation (i.e., before the code is actually executed). Certain languages or their recent versions allow not to specify types in cases where the type can be found in other parts of code. For example, a compiler will assign the data type to the variable based on the first object assigned to it and disallow objects of other types to be assigned there as well. This kind of inference can be (and actually has to be) done completely in compile time and does not present any new options or functionality to the programmer, it is just a matter of convenience.

Dynamically Typed Languages In this paper, by type inference, it is meant the kind used for dynamically typed languages. In dynamically typed languages, data types are usually not provided by the programmer and instead of checking and verifying data types, only behavior of the object during execution is important. This is very convenient for fast prototyping, but poses a challenge for analysis of the code, as it is much harder for both human analysts and analysis tools to determine what kinds of objects could be assigned to the variable, returned by the method or provided as parameters of the method. To achieve that, analysis tools (and possibly integrated development environments – IDE) rely on multiple algorithms and heuristics used in type inference [15]. There are two general ways to do that: *static type inference* is usually used where *run-time – or dynamic – type inference* is another option for certain cases.

Static Type Inference As the name suggest, static type inference does not need the code to be actually executed or even fully executable. It relies on information in the code itself and there are several techniques, algorithms, and principles to gather these types. For example, wherever a new instance of a class is created, then assigned to an instance variable (attribute), the class of which the instance has been created is one of the possible types of the variable. However, even in this case, where the class surely is a possible type, there is no indication whether even its superclass and its other subclasses might be used as well.

Run-time Type Inference The runtime type inference relies on running the application, executing tests, or otherwise executing the code. Once the code is executed, any actually used type is being recorded. Unfortunately, there are several problems limiting the usefulness or even preventing the possibility to use it [10]. Main issue is with applications that can no longer be executed and fully used and or not even very thoroughly tested. The reason is that anything about parts of the code that have not been used cannot be recorded, although rest of the application has been executed. Additionally, if the method could return both an integer and a floating-point real number, but returns only integers during the analyzed execution, its type could be wrongly assumed to be an integer, although it could be any number.

1.2 Problem Description

This work attempts to present answers to the following questions:

1. How to improve the generating of UML associations from Pharo code, possibly using and improving upon existing type inference tools and techniques in Pharo?
2. How to integrate these results into an existing UML and Java code generation earlier prototype?

Regarding the usage of type inference, the focus is on the amount and quality of results rather than time efficiency.

2 Previous Results & Related Work

This chapter describes the past work of the authors, related state-of-the-art work, and the current type inference implementation in Pharo.

2.1 Past Work of the Authors

Authors entire work consists of multiple problems and the problem of linking everything together from the Pharo code to UML Class Diagram and then into a new code in different programming languages. This paper is a follow-up of a previous one describing the technical details of the main ideas and early implementation [2], while this one focuses on further achievements on generating associations between classes using type inference based on combining and improving upon currently known ways.

2.2 Related Work & Current State-of-the-Art

Normalized Systems theory [7] suggests building software from 5 basic elements. The system should be modified on the model level only, disapproving any modifications on the generated system as a whole. If such modifications are done, there are various harmful combinatorial effects impairing the modularity and the ability to reverse-engineer such changes back into the model. In practice, unless Normalized Systems theory or strict model-driven development is followed, such problems are very common and require tools for generating models out of the code itself.

There are various tools for generating UML models and diagrams for statically typed languages as well as models similar to UML or partial UML models for dynamically typed languages, including Pharo.

For example, Pyreverse [12], PyNSource [4], and Lumpy [9] are diagram generating tools for Python and Umlify [21] for Ruby. Lumpy, PyNSource and Umlify do not show any data types or associations, while Pyreverse, as the most advanced of all, does. However, because of type inference limitations, only some basic or easily inferred associations and types of instance variables are extracted and method arguments and return types have not type information at all, as can be seen in Figure 1.

Special case is the popular web application framework Ruby on Rails with very strict naming rules that allows both the framework itself and related tools to find all important data types and associations.

In the case of Pharo, a code analysis tool Moose has several features, generating FAMIX [8, 13] models being the most relevant. It is able to display very few associations thanks to the type inference tool RoelTyper. Similarly to Pyreverse, no method argument types and return types could be found using it. Although it provides great ideas, it cannot be accepted as a solution, as general knowledge of the modeling notation, commercial tools support, and the existence of code generating tools is important as well.

OpenPonk modeling tool [24] implemented in Pharo offers UML Class Diagram modeling and provides a full UML metamodel generated from UML specifications, although no means of generating it.

2.3 Type Inference in Pharo

Pharo community offers 3 static type inference tools – RoelTyper, RBRefactoryTyper, and J2Inferer – used, for example, by refactoring tools [26]. It is important for the community and is subject to several works even just for Pharo itself [22, 11, 19, 5, 16].

Out of these three, only J2Inferer [23], although being much slower, provides inference of method arguments and return types [23] and only RBRefactoryTyper is able to find the types of contents of collections, while others just mark the type as, for example, Set, without any indication what elements count be inside.

Run-type (also called dynamic) type inference has been the subject of research for a long time as well [5, 28], including specifically Pharo [17, 25].

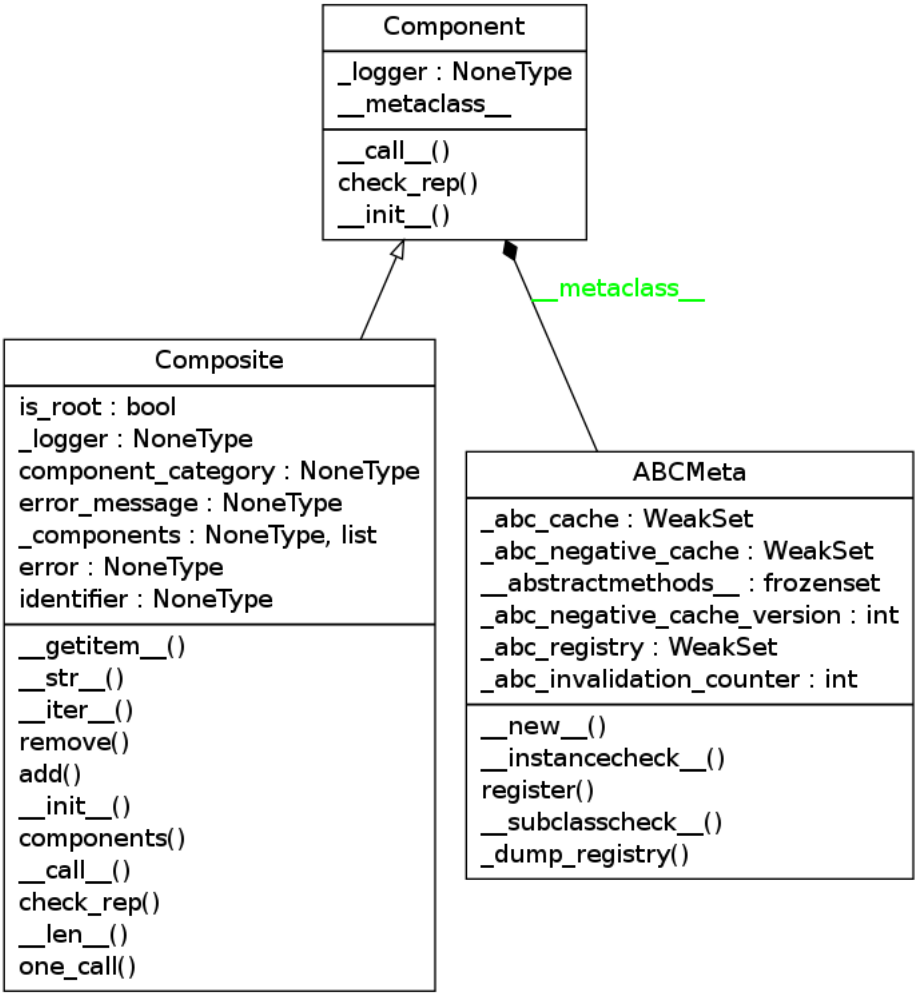

Fig. 1: Pyreverse [12] output with unknown types marked as NoneType

3 The Solution

As stated earlier, Pharo is used as the source platform in the case study. Since the long-term goal is transforming its code into a statically typed programming languages, Java has been chosen for its lasting popularity and general support. It would be possible to attempt to directly generate Java classes with some content from Pharo itself, however, the authors do not aim at full automation of code transformation, but rather at providing the best information and simplifying the lives of human analysts. Therefore, another step is taken in between.

Since there already is a partial UML model structure generator [2], it can be used to create UML models and class diagrams first, improve it, especially the type inference part to create as many associations as possible, and finally use existing tools for generating Java code out of UML.

3.1 Generating Associations

To generate associations, type inference is required. With that as a requirement, type inference can be used not only for associations, but for all possible items with data types, like instance variables, method attributes, and method return types. Actually, it is up to the UML user whether she or he creates an association with another class or just marks the data type of a simple UML class attribute. In this use case, associations are preferred between classes inside the packages that are being analyzed and generated from, and simple attributes in other cases, like `String` or `Date`, although these are classes as well.

Their results may come in the form of a list of possible classes (data types) where only one could be used in both UML and Java code. The correct one might be one of them or any of their common superclasses. Choosing the right one may be done either by human analysts or automatically – depending on how much and how precise the results are needed compared to time and effort. A simple user interface has been created to provide analysts with the found possibilities along with a code browser and information about superclasses. In case of automatic decisions, a quite simple search of the common superclass is used, with possible improvements for future work. For example, in Figure 2, where `SmallInteger` and `LargeInteger` are both provided as possible, their closest common superclass – `Integer` – is marked as the result. If `Float` was added to the set of possibilities, `Number` would be the result.

Despite all that, in some cases, no singular type could ever be found. For example, it is completely legitimate to provide either an instance of class BlockClosure or Symbol as an argument in several cases, without them having any common superclass except Object – the superclass of all classes in both Pharo and Java.

3.2 Real-time Inferer

Because the success rate of static type inference is quite limited, an own real-time type inferrer has been implemented, inspired by previous solutions [17, 25].

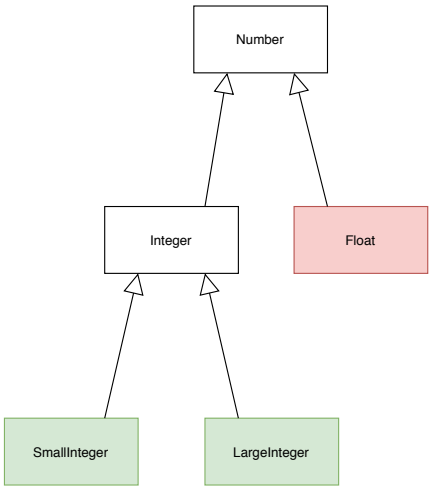

Fig. 2: Simplified class structure example

First, it collects the types of a package (or multiple packages), then tests and examples and use cases are executed to run the code and collect those types. Finally, the recording of data types is stopped and the inferrer with its results is passed inside its own adaptor to the main tool that does not even need to know how those results were collected.

The real-time type inferrer relies on the code being actually executed. Methods that are never executed could have no information about its types, as well as instance variables having no information if nothing has been assigned. On the other hand, there might be multiple classes, because any object could be present in the case of dynamically typed languages. For example, an integer, a float, a date, or any other in the very same variable, even during a single execution of a single method. Furthermore, only a single class, or even just its subclasses, might be recorded as a type, even if some other could be there as well: in a case where only one alternative usage has been executed but not the other. An example of such structure is illustrated in the figure 2, where `Integer` subclasses are correctly marked as possible types, while Float is not.

As a result, the amount and quality of found types depends on the amount and quality of tests, examples or ability to manually use the application to its greatest extent. In some cases, the real-time inferrer found several times more types than all static ones combined, while in other cases it found completely nothing as there was nothing that could have been executed.

3.3 Inferer Chaining

All three described static type inference tools in Pharo have been tested – RoelTyper, RBRefactoryTyper, and J2Inferer – along with the custom real-time solution and compared their results. Adaptor system has been created with adaptors for each of them, that allows to switch them easily during both testing and final UML generating. Some attributes were found by one of them, other attributes by other ones, and J2Inferer added some type information about methods. As a result, choosing any single of them did not provide satisfactory results. To take the best results of each alternative, all of them are being chained:

All inferrers are being asked one by one until a sufficient answer is provided. For example, whenever RBRefactoryTyper did not find any sufficient answer, J2Inferer is asked. As sufficient are marked such outputs that provide either one type or multiple ones with a common superclass other than `Object`.

4 Results and Comparison

To evaluate the results of each type inferrer, along with all of them chained, each inference option has been used on several packages in Pharo with automatic evaluation of possible provided options by each one.

4.1 Inference Output Comparison

To compare the results, instance variable types are inferred to create associations, the amount of these is compared, then these counts are added to the amount of found method attribute types and method return types. For simplification purposes, instance variables, method attributes, and return types are further often referred just as "items".

In both cases, for each package used for testing, following steps were taken:

1. picked several core packages of Pharo
2. used each type inferrer separately and all of them chained
3. set automatic way of picking types
4. in the case of real-time type inference, executed all available tests and executable examples
5. counted items with sufficient results (single found type other than Object)

Since the focus is on the amount of results, rather than time, their efficiency has not yet been compared, although it is clear that the combined approach takes longer time than any single one.

Comparison of found instance variables, and therefore possible UML associations, can be found in Table 1. The table has an entry for each package, the total amount of instance variables, the amount of them sufficiently answered by each type inferrer, including the chained one, and percentage of the total amount. This is the percentage of possible associations that could be found and created in the UML model.

Table 1: Comparison of the amount of inferred variable types by each type inferrer

Package name	Total	Roel	RB	J2Inferer	Real-time	Chained
TraitsV2	19	14 (74 %)	16 (84 %)	9 (47 %)	10 (53 %)	16 (84 %)
Tool-Diff	56	33 (59 %)	28 (50 %)	16 (29 %)	41 (73 %)	44 (79 %)
XML-Parser	337	210 (62 %)	207 (61 %)	129 (38 %)	244 (72 %)	264 (78 %)
Zinc-HTTP	167	77 (46 %)	88 (53 %)	48 (29 %)	117 (70 %)	132 (79 %)
Athens-Cairo	222	34 (15 %)	30 (14 %)	22 (10 %)	37 (17 %)	47 (21 %)
OSWindow-Core	440	63 (14 %)	59 (13 %)	39 (9 %)	38 (9 %)	82 (19 %)
LibGit-Core	257	10 (4 %)	15 (6 %)	8 (3 %)	0 (0 %)	15 (6 %)

Table 2: Comparison of the amount of all inferred types by each type inferrer

Package name	Total	Roel	RB	J2Inferer	Real-time	Chained
TraitsV2	744	14 (2 %)	16 (2 %)	136 (18 %)	314 (42 %)	386 (52 %)
Tool-Diff	564	33 (6 %)	28 (5 %)	85 (15 %)	307 (54 %)	342 (61 %)
XML-Parser	6659	210 (3 %)	207 (3 %)	1247 (19 %)	4769 (72 %)	4878 (73 %)
Zinc-HTTP	2364	77 (3 %)	88 (4 %)	312 (13 %)	1590 (67 %)	1648 (70 %)
Athens-Cairo	2064	34 (2 %)	30 (2 %)	92 (5 %)	615 (30 %)	656 (32 %)
OSWindow-Core	1783	63 (4 %)	59 (3 %)	176 (10 %)	175 (10 %)	342 (19 %)
LibGit-Core	3363	10 (<1 %)	15 (<1 %)	451 (13 %)	18 (<1 %)	476 (14 %)

Table 2 has the same structure, but contains method-related types (method arguments and return types) along with instance and shared variables.

From these tables, it is clear that the results vary greatly between each package and each inferrer. In some cases, static type inferrers were way more successful than real-time one, while in other cases it was the other way around. Interesting is the comparison between Zinc-HTTP and LibGit-Core in the first table, where real-time inferrer has substantially greater results in the case of Zinc-HTTP while having absolutely no results in the case of the LibGit-Core.

In the second table, where method-related types are added to the numbers, RoelTyper and RBRefactoryTyper have substantially worse results, because these are not able to find method-related types at all. Results of J2Inferer and real-time inferrer are worse too (except for LibGit-Core package), but by a much smaller margin, especially in the case of the real-time inferrer.

Most importantly, yet as expected, using both static and real-time inference combined provides the best results – even in the worst case it has results equal to the best type inferrer out of the four.

In the end, it is possible to create UML models with associations, create UML class diagrams and generate a basic structure of Java code with classes, empty methods, and attributes, mostly with proper data types. This part is yet still in progress.

Table 3: Relation to Test Code Coverage

Package name	Code coverage	Examples	Real-time	Chained
TraitsV2	51 %	0	314 (42 %)	386 (52 %)
Tool-Diff	0 %	3	307 (54 %)	342 (61 %)
XML-Parser	80,5 %	0	4769 (72 %)	4878 (73 %)
Zinc-HTTP	52 %	3	1590 (67 %)	1648 (70 %)
Athens-Cairo	3,6 %	35	615 (30 %)	656 (32 %)
OSWindow-Core	4,5 %	11	175 (10 %)	342 (19 %)
LibGit-Core	0 %	0	18 (<1 %)	476 (14 %)

4.2 Package Difference Reasons

As stated previously, real-time type inference logs only the types of actually executed code with as much variance as possible. To get the best results, as many tests, examples, and use-case scenarios have to be executed as possible. Such differences between packages might therefore be related to the amount of such tests, examples, and use-case scenarios provided. To verify this hypothesis, code coverage was calculated along with counting amount of executable examples used for logging and is presented in table fig:coverage.

Although no reliable outcome could be generalized with such a small amount of packages, it indicates that with increasing amount of examples and especially code coverage, real-time type inference results might improve too. There are still some exceptions, like Tool-Diff, that has no tests and only 3 examples, yet these examples are so thorough that result in a quite high amount of found items. Although LibGit-Core has no tests and no examples, 18 types have been found, because LibGit is constantly being used by background processes.

5 Discussion and Conclusions

Type inference is important for dynamically typed languages for both analysis and refactoring tools. Because of that, lots of research and engineering work has been put into it and several existing tools. There is no single one consistently better than any other, therefore the proposed approach of combining existing solutions along with the custom inferrer provides the best results in cases where the amount of results is more important than speed. Although there might be cases where the result does not improve upon all specific existing algorithms or tools, using the best result of all guarantees that the result is at least as good as the best one, while the initial results presented in tables 1 and 2 suggest that most of the cases get actual improvements.

Because of that, it is possible to continue efforts towards providing ways for analysts and researchers to generate UML and finally statically typed modern code out of legacy systems created in dynamically typed languages.

5.1 Programming Languages

Although no legacy applications are implemented in the Pharo itself, Pharo shares all main similarities with various Smalltalk implementations where several important legacy systems exist. For example, there are tools to generate VisualWorks code from Pharo and vice versa [14], making this solution for Pharo applicable to at least some other important Smalltalk-based languages and Smalltalk implementations as well. Although the main ideas of the article could be applicable to Python, Ruby, and other popular dynamically typed languages, practical implementation possibilities have yet to be analyzed, especially regarding real-time type logging mechanisms.

5.2 Generated Models Quality

While the generation of classes, attributes, and operations is mostly good enough to understand their use and the amount of found data types has been increased, models lacking even as little as every tenth association could not be very reliable regarding the overall architecture. With that in mind, most of the future work is still related to the ability to find associations and data types both automatically and with human input.

5.3 Future Work

Despite the advancements, there are several ways to build upon the work or improve what already exists.

Generating UML Class Diagrams The proposed solution is able to create the full UML model and generate Java code using existing code generators, yet generating UML class diagrams out of models is a much more complex task. In the case of large applications with several hundreds or thousands of classes, creating a single diagram for the whole model would be near to useless. Models for each subpackage could be created, but many programming languages have a very shallow package structure or structure so deep that each diagram would consist of just a few classes without a general overview of the related parts of the system.

User Interface Although this paper compares purely automatically gained results, some human interaction would be required in some cases, for which a user interface and proper tooling would need to be created. While a very simple GUI (Graphical User Interface) for choosing the data types manually from multiple options has already been created, most of the work has to be done through code. The authors aim to improve the situation by creating a complete user interface.

Selecting Type from Options Provided by Type inferrers The authors plan to improve the way of selecting the final type from the options offered by each type inferrer as well. For example, in some cases, it could be assumed that a class from the same package has a greatly higher chance of being the correct type than from a completely unrelated package. This assumption, however, may be wrong in other cases and needs further analysis.

Improving Type Inference Tools Although type inference results have been greatly improved, there are always many ways to get even better results. At this time, only existing type inferrers are used as black boxes, yet these could be improved using known algorithms and rules not yet implemented in Pharo.

6 Acknowledgements

The presented research was sponsored by Tomcat® computer GmbH.

References

1. Ambler, S.W.: Process patterns: building large-scale systems using object technology. Cambridge university press (1998)
2. Bliznicenko, J., Pergl, R.: Generating UML Models with Inferred Types from Pharo Code. Not yet published (2019). URL http://esug.github.io/2019-Conference/articles/2019-08-26-IWST19.zip
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synthesis lectures on software engineering **3**(1), 1–207 (2017)
4. Bulka, A.: PyNSource [software]. Online (2019). URL https://github.com/abulka/pynsource
5. Chugh, R., Jhala, R., Lerner, S.: Type Inference with Run-time Logs (Work in Progress)
6. Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., Tolbert, D.: Unified Modeling Language (UML) Version 2.5.1. Standard, Object Management Group (OMG) (2017). URL https://www.omg.org/spec/UML/2.5.1
7. De Bruyn, P., Mannaert, H., Verelst, J., Huysmans, P.: Enabling Normalized Systems in Practice–Exploring a Modeling Approach. Business & Information Systems Engineering **60**(1), 55–67 (2018)
8. Demeyer, S., Ducasse, S., Tichelaar, S.: Why FAMIX and not UML. In: Proceedings of UML'99, vol. 1723 (1999)
9. Downey, A.: Lumpy: UML in Python [software]. Online (2007). URL http://www.greenteapress.com/thinkpython/swampy/lumpy.html
10. Meijer, H.J.M., Obasanjo, O.V.: Efficient data access via runtime type inference (2011). US Patent 7,970,730
11. Milojkovic, N., Ghafari, M., Nierstrasz, O.: Exploiting type hints in method argument names to improve lightweight type inference. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 77–87. IEEE (2017)

12. Nakamura, N.: Exploring Pyreverse. Online (2014). URL https://pythonhosted.org/theape/documentation/developer/explorations/explore_graphs/explore_pyreverse.html
13. Nierstrasz, O.: Agile software assessment with Moose. ACM SIGSOFT Software engineering notes **37**(3), 1–5 (2012)
14. Object Profile: Pharo2VW [software]. Online (2018). URL https://github.com/ObjectProfile/Pharo2VW
15. Plevyak, J., Chien, A.A.: Precise concrete type inference for object-oriented languages. ACM SIGPLAN Notices **29**(10), 324–340 (1994)
16. Pluquet, F., Marot, A., Wuyts, R.: Fast type reconstruction for dynamically typed programming languages. In: ACM Sigplan Notices, vol. 44, pp. 69–78. ACM (2009)
17. Rapicault, P., Blay-Fornarino, M., Ducasse, S., Dery, A.M.: Dynamic type inference to support object-oriented reengineering in Smalltalk. In: ECOOP Workshops, pp. 76–77 (1998)
18. Rumbaugh, J., Jacobson, I., Booch, G.: Unified modeling language reference manual, the. Pearson Higher Education (2004)
19. Schweizer, D.: Exporting MOOSE Models to Rational Rose UML (2000)
20. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE software **20**(5), 42–45 (2003)
21. Sokol, M.: Umlify [software]. Online (2011). URL https://github.com/mikaa123/umlify
22. Spasojević, B., Lungu, M., Nierstrasz, O.: Mining the ecosystem to improve type inference for dynamically typed languages. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, pp. 133–142. ACM (2014)
23. Tesone, P.: Type inference in Pharo. Online (2015). URL https://github.com/tesonep/j2Inferer/blob/master/presentation/main.pdf
24. Uhnák, P., Pergl, R.: The OpenPonk modeling platform. In: IWST, p. 14 (2016)
25. Uhnák, P., Pergl, R.: Ad-hoc Runtime Object Structure Visualizations with MetaLinks. In: Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies, p. 7. ACM (2017)
26. Unterholzner, M.: Improving refactoring tools in Smalltalk using static type inference. Science of Computer Programming **96**, 70–83 (2014)
27. Vernadat, F.: UEML: towards a unified enterprise modelling language. International Journal of Production Research **40**(17), 4309–4321 (2002)
28. Xu, Z., Zhang, X., Chen, L., Pei, K., Xu, B.: Python probabilistic type inference with natural language support. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 607–618 (2016)