Generating UML Models with Inferred Types from Pharo Code

Jan Blizničenko, Robert Pergl

Generating UML Models with Inferred Types from Pharo Code

Jan Blizničenko
Dept. of Software Engineering
Faculty of Information Technology,
Czech Technical University in Prague
Prague, Czech republic
jan.bliznicenko@fit.cvut.cz

Robert Pergl
Dept. of Software Engineering
Faculty of Information Technology,
Czech Technical University in Prague
Prague, Czech republic
robert.pergl@fit.cvut.cz

Abstract

Generating structural UML models from Pharo code poses several problems that need to be addressed. A structure of classes with variables and methods has to be gathered, types of instance variables have to be found for associations and the model has to be importable into common modeling tools. This paper brings a review of current solutions to these and similar problems and it presents our ongoing effort towards this goal – our current solution we developed so far.

CCS Concepts • **Software and its engineering → Software reverse engineering**;

Keywords type inference, metalinks, uml, model generating

ACM Reference Format:
Jan Blizničenko and Robert Pergl. 2019. Generating UML Models with Inferred Types from Pharo Code. In *Proceedings of International Workshop on Smalltalk Technologies 2019 (IWST '19), Cologne, Germany,* 9 pages.

1 Introduction & Motivation

Pharo, being a modern implementation of the Smalltalk programming language follows the traditional style of coding using class browsers and various inspection tools. While this brings an excellent focus and navigability during the programming, seeing the "big picture" and exploring big code bases may be challenging.

UML [17] is considered today's industrial standard notation for software systems modeling and it offers multiple types of diagrams depicting various aspects of a software system. Software engineering best practices advise to model the system and then implement it [1]. Model-driven approaches [20, 29] go even further and bring a rigorous and often automated or semi-automated way of transforming conceptual models into implementation models and code.

However, without providing concrete statistics, we may state that their adoption in practice is not excitingly high and software artifacts often come with missing documentation and models, or the models get outdated in the course of time.

This is why it becomes interesting to look into the possibility to (at least partially) reconstruct UML diagrams from existing code bases. Such approach obviously cannot deliver conceptual-level diagrams any more, but seeing the code visualized in a diagrammatic way can help understand a system at hand.

Indeed, there are tools available for Java, C++, C# and other languages that provide UML reverse-engineering functionality, typically to reconstruct UML Class Diagrams. To reconstruct types of attributes, methods signatures and to reconstruct associations between classes, these tools use type information of instance variables which is available in these statically-typed languages. In dynamically typed languages – such as Pharo – there is no such information present in the code.

1.1 Type Inference Techniques

To reconstruct data types in code without explicit type definitions, type inference may be used [14]. There are two main kinds of type inference: the most usual one is the *static type inference* and the other one is the *run-time (also called dynamic) type inference*. Type inference might result in false positives (a class is marked as a possible type, although it is not) or false negatives (a class is ruled out although it is one of possible types).

Static inference uses information provided in the code itself and these are several examples of typical principles used, which illustrate the pitfalls:

- Wherever new instance of a class is created, then assigned to an instance variable, we know that the class of which instance has been created is one of possible types of the variable. If there is a line with `aVariable := MyClass new`, the algorithm can guess

IWST '19, August 27–29, 2019, Cologne, Germany

that `aVariable` is of type `MyClass` or any of its superclasses. However, even if the class was `SmallInteger`, the proper type of variable does not have to be the `SmallInteger`. It might also be any of its superclasses – it might be `Integer`, or generally any `Number`. In case of classes with no common superclass, `Object` as the only general superclass is used.

- A variable might only contain instances of classes that understand methods sent to the variable (unless reflection is used). A chance of eliminating false positives rises with the amount of messages being sent to the variable (otherwise there might be many unrelated classes implementing same messages too). Once we have code `aVariable addSomething; addThat; build`, we know that `aVariable` can only be instance of classes implementing all of such named methods – probably some builder in this case. The only downside to this rule is a case where object might understand a message without having such methods implemented – usually because `doesNotUnderstand:` is implemented.
- Wherever an instance variable is returned in method, its type is used as a return type of the method. Once we find `^aVariable`, the return type of the method has to be the same one as the type of the variable.

If a static analysis of a variable, method parameter or return type identifies a single class, it is usually the one that should act as type in Java and UML. Whenever several classes are marked as possible, we might either use their common superclass as the single type (sometimes, it might be `Object`). It means that in a statically typed language, such as Java, the identified class or any of its superclasses will become the type of the variable and the return type. The problem of static analysis is that it contains many false positives which need to be sorted out by a human analyzer.

The second way of type inference is the runtime one. The runtime type inference is based on recording every type that was actually contained in a variable (or in a method parameter or the returned value) once the code has been executed. It means the class of an object actually assigned/-passed/returned. This is, however, not always possible [10]. It requires a thoroughly tested code or a code that can be manually executed, including lots of possible use cases. The reason is that we can record a data type only for variables actually used during runtime. We need to record the class of the object assigned to the variable. If just an `Integer` is passed as an argument during the execution, we might miss that a `Float` or a `Fraction` might be passed as well, so we get false negatives.

Regardless of the problem of finding the single correct datatype in case one exists, there are cases where finding just a single class (data type) is not possible at all. For example, enumeration methods in collections, such as `do:`, `collect:`, `select:`, etc. accept blocks and symbols as an argument, which have no other common superclass than `Object`, so there is no other option in UML than to offer `Object` as the data type – we cannot provide a list of possible types. If there were interfaces in Smalltalk, `Block` and `Symbol` would have the same interface, but there is no explicit interface concept as is in Java.

In this paper aims to present our ongoing work on generating structural UML models from Smalltalk code.

2 Problem Description

The long term goal of our work is generating UML Class Diagrams from Smalltalk code, now focusing specifically on generating structural UML models from Pharo code. It is important to distinguish between UML models and UML Class Diagrams. UML Class Diagram is a visual representation based on structural parts of UML model. In this paper, by UML model, we mean the structural part of the model that provides base for Class Diagrams. The resulting model should be importable into commercial modeling tools such as Enterprise Architect [22]. To achieve that, the following questions need to be addressed:

- How to represent UML models in Pharo?
- How to generate a UML model with class structure, operations (methods), properties (variables) etc. from a Pharo code?
- How to find types of instance variables, method parameters and return types?
- How to transfer/import the generated model to Enterprise Architect?
- How could be such model used for creation of UML Class Diagrams with as many automation as possible?

2.1 Existing Solutions & Related Work

Our work consists of multiple problems and problem of linking everything together from the Pharo code to UML Class Diagram in a tool similar to Enterprise Architect. Here we present the overview of relevant related work.

2.2 Existing UML Generators

For Pharo and also other dynamically typed languages, there are already tools for generating UML Class Diagrams or diagrams similar to UML.

2.2.1 Python

In Python, there are multiple tools for generating UML, like Pyreverse [12],
PyNSource [5] or Lumpy [8], although some are not actively developed or supported anymore.

These tools create a UML model with Class Diagrams, or diagrams very similar to UML Class Diagrams. Lumpy does not generate method arguments nor data types, but

Pyreverse, as the most advanced of all, does, however because of type inferring limitations, only some basic or easily-inferred types of instance variables are extracted, while the rest are marked as NoneType (Python's equivalent of null used where Pyreverse does not know what type the variable has). An example of an output of Pyreverse can be seen in figure 1. Also, no method argument types are inferred at all.

PyNSource does not show any data types at all, as seen in figure 2.

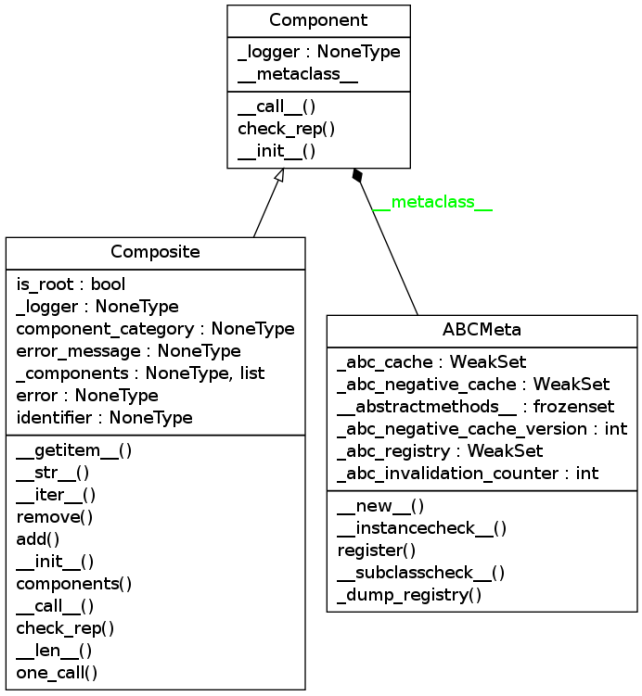

Figure 1. Example of output from Pyreverse [12]

2.2.2 Ruby

Ruby has a smaller community then Python and the amount of tools for UML is smaller, too. One of such tools is Umlify [21], which is able to generate UML diagrams without providing any datatypes or any information about method arguments (just names are generated, as seen in figure 3), similarly to, for example, Lumpy. There are also multiple UML generators for Ruby's most popular web application framework – Ruby on Rails – for which most of the tools offer to generate data types of variables. It is possible because Ruby on Rails has strict naming rules where a name of a variable always has to be derivative of the class of its content, including information about multiplicity. That is not the case of non-Rails Ruby applicationa, just as is it not the case of Pharo or most of Smalltalk implementations, although some naming conventions are in place.

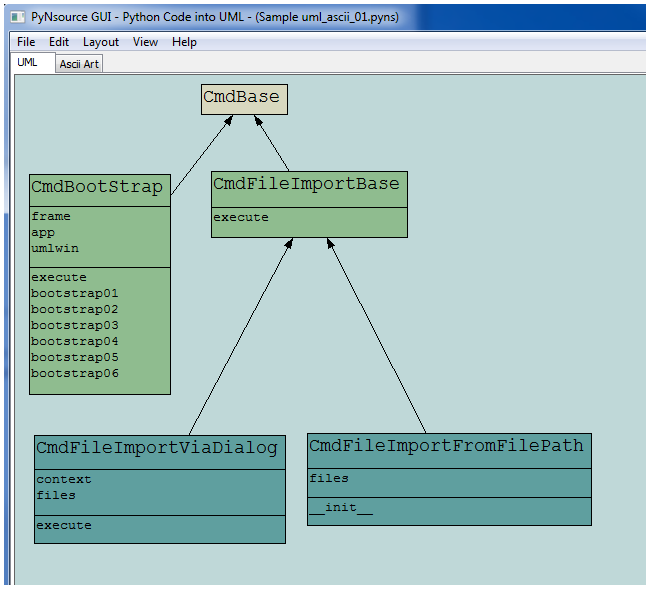

Figure 2. Example of output from PyNSource [4]

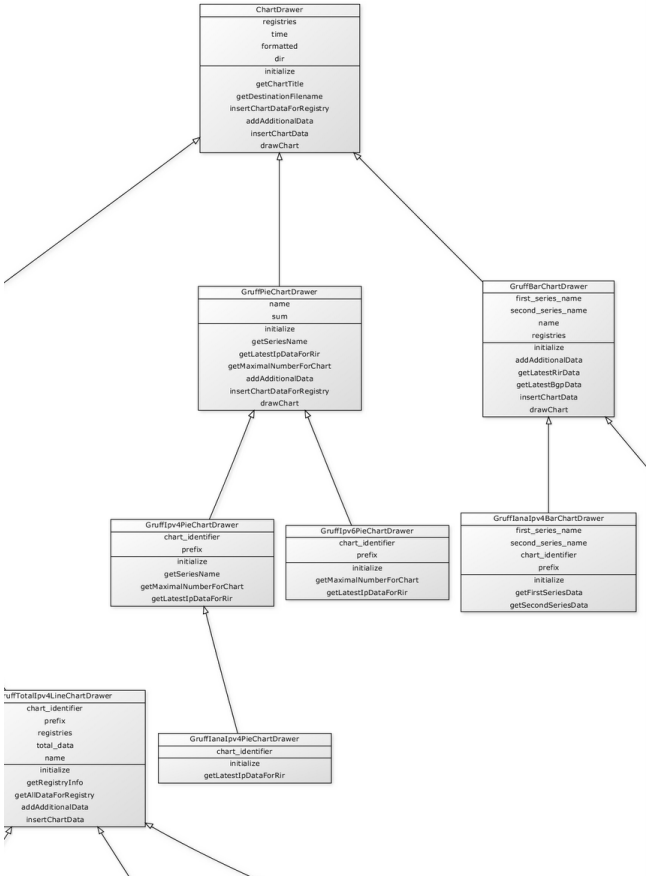

Figure 3. Example of output from Umlify [27]

2.3 Moose

Moose is the closest example of a tool aligned to our goals. It does not generate a UML model, but a FAMIX model, which is quite similar to UML and even has multiple advantages over UML [7, 13]. However, FAMIX is not a solution in our case, as general knowledge of the modeling notation and commercial tools support is important for us.

Moose leverages the Roassal visualisation library [2, 3] to show parts of the model as diagrams similar to UML. To create associations, it uses a static type inferrer RoelTyper. However, RoelTyper is not able to find any method argument types and return types. Also, most instance variable types cannot be found just by static type inference, therefore there are just very few associations between classes created.

2.4 Type Inference in Pharo

Lots of work has been already done in the field of static typing inference in Pharo, too. There are 3 major tools for static type inference in Pharo - RoelTyper, RBRefactoryTyper and J2Inferer – and few experiments on run-time type inference.

These tools are mainly being used to augment other tools in Pharo, such as refactoring tools [28]

2.4.1 RoelTyper

RoelTyper is a tool and a set of algorithms by Roel Wuyts. It uses several heuristics algorithms and it aims to be as fast as possible, claiming to process all code in the whole default VisualWorks image in less than 1 minute and a half.

Since the focus is on speed, it does not find as many types as other tools and it also does not try to find types of content in a collection. For example, if there is an `Array` of instances of `String`, RoelTyper correctly finds that the variable contains an instance of `Array`, but it does not find what is inside the array, which is crucial for real usage in UML generating.

Also, it can find only types of instance variables, no types of arguments of methods, nor returned values.

2.4.2 RBRefactoryTyper

RBRefactoryTyper, as the name suggests, is used for refactoring. A user may pick a method and variable(s) used in the method, ask to move the method and RBRefactoryTyper offers possible classes to which the method may be moved to. It offers those classes, whose instances can be in the variable, because it might mean that the method working with an instance of that class could be, in some cases, moved to that class itself to improve encapsulation.

RBRefactoryTyper analyses AST (abstract syntax tree) and it is usually slower than RoelTyper's heuristics, but offers more features than RoelTyper. Based on testing on several real packages, RBRefactoryTyper guessed more types with a better precision[1].

[1]By precision is meant how many possible classes the tool offered – if they both offer the correct class, the less other classes offered, the better.

Despite these achievements, RBRefactoryTyper cannot infer types of method arguments, nor returned values of methods.

2.4.3 J2Inferer

J2Inferer is a prototype tool in Pharo by Pablo Tesone that aims to infer types not just of instance (or class-side instance) variables, but also of method arguments and return types [24]. As such, it represents a promising approach, however the tool was unfinished in time of concluding our research and could not yet infer method arguments and return types, although multiple issues of J2Inferer had been fixed.

2.5 More Work on Type Inference

Improving static type inference results in Pharo has been a goal of several works [6, 11, 15, 19, 23] that propose new algorithms and improvements of current ones.

2.5.1 Runtime Type Inference

Run-type type inference has been researched both generally and for several specific languages [6, 9, 18]. These languages include Pharo, as described in [16, 26], both proposing use of AST modifications, or specifically Metalinks, to record data types.

Metalinks allow to enhance AST in a way that before, after or instead any node in AST something else happens. It can be used to find values assigned to variables, returned values and many more usages.

2.5.2 Creating UML Models in Pharo

In Pharo, there is a possiblity to use Graphviz – an open-source graph visualization software – to render UML Class Diagrams. These are not UML models that may be exported, just drawn diagrams.

There is also the OpenPonk modeling tool [25] – a metamodeling platform and a modeling workbench that offers UML Class Diagram modeling and provides a full UML metamodel generated from UML specifications. The metamodel is designed to be usable for creating custom UML models even without the OpenPonk tool itself.

3 Our Solution

Our solution is based on existing solutions and toold and it consists of separate parts addressing each problem and then linking all parts together. The following subsections are each dedicated to a single question presented in section 2.

3.1 Representing a UML Model in Pharo

For representing a UML model in Pharo, the UML metamodel for OpenPonk offers exactly what we need. It allows to create any UML model which can be either opened in OpenPonk itself directly in Pharo, or there is an exporter into a XMI-format file. More on exporting will be discussed

in the subsection dedicated to transfering model from Pharo to Enterprise Architect. The UML metamodel is independent on OpenPonk tool itself, which means we can use it alone without OpenPonk and even loading the tool into the image is not required.

3.2 Generating the Main UML Model Structure

UML models for class diagrams share lots of similarities with the FAMIX model, and Moose has its Smalltalk importer into FAMIX. It creates all the packages, classes, methods, instance variables and it offers an option to use RoelTyper for finding some types of instance variables to create associations between classes. We created a custom implementation of UML generator loosely inspired by Moose importer.

First of all, a collection of packages that should be imported and analysed is defined. We call these packages *focused packages*, because these are the packages we are interested in. Once the collection of focus packages is provided, UML packages are created from them and separated into inner packages by tags. Then classes are created without any content, followed by creating generalizations for all the classes.

Next, instance variables (and class-side instance variables and shared class variables) are added. During adding the instance variable, provided instance of class we call `TypesManager` is asked for type of the variable. Detail about `TypesManager` are discussed in subsection 3.3, for now let's treat it as a black box that gives us either a class representing the correct data type or just class `Object` if none was found. Adding the class first before adding variables to it effectively prevents deadlock in case of circular dependency between classes (through their variables).

Whenever `TypesManager` returns a class from any package that is not focused (other package than one of those initially given as input), such a class has not yet been added to the UML model before. In that case, the class is added and then used as the data type (and will not be added again once `TypesManager` returns it next time). The classes from not-focused packages, that are being used just for types, do not have any variables or methods added by default (can be redefined by subclasses of the generator). However, in many cases, we do not want to add variable just as a UML property, but instead we prefer creating an association between those two classes. In case we know the type of the variable (it is not `Object`) and the class is a part of our focused packages, an association is created between the class that contains the variable and the class that should be type of that variable. If we do not know the type or the type class is not from focused packages, we just add the variable as a UML property.

As the last step, methods are added along with their parameters and their types are being requested one by one whenever needed first time, along with return types.

These steps result in a complete UML model of the class structure of the focused Pharo packages. The following section describes options we use for finding data types.

3.3 Finding Types

The UML generator has its `TypesManager` that may be asked for type of an item (variable, parameter or return type). `TypesManager` remembers the already found types and asks type inferrers for types not yet asked for.

3.3.1 Type Inferrer Adaptors

As described in previous chapters, there are multiple existing solutions and tools for type inference, along with several research results that could potentially improve existing tools.

During our main work on the project, J2Inferer was not functional at the time, so we used the other two static type inference tools during our preliminary experiments - RoelTyper and RBRefactoryTyper – where both had their advantages which varied over each analyzed package. There were variables found by RoelTyper and not by RBRefactoryTyper and other variables found by the RBRefactoryTyper but not by RoelTyper. In addition to static type inference, our goal is to use realtime (dynamic) type inference, as well.

For that, we created abstract class TyperAdaptor, designed to be subclassed, where each subclass is class of adaptor for some type inference tool, being static or dynamic.

Since each tool (represented now by the adaptor) may provide type for some variables that no other tools are able to find, and therefore complement each other, we use several adaptors (several inference tools/techniques) over a single package. For that goal, we created adaptor combiners with same interface as adaptors that contain several adaptors in an ordered collection, where in case one did not find any type (i.e. the type is `Object`), the next one is asked for the type of the same item (variable, parameter or return type).

3.3.2 Types Manager

A common case is that the type inferrer offers several possible classes as types, either because all are truly possible, or because some are false positives.

Whenever this situation happens, a choice has to be made, which single type do we want to use and that is the task of Types Manager. Types manager that asks its adaptor or adaptor combiner for collection of possible types is an instance of class `TypesManager` or its subclass. Each subclass may redefine what kind of solution will be used in situations where several possible types are provided.

The most trivial one would just return `Object` in such a case, because we do not know which of the given possible types is the correct one. We did not create such implementation of `TypesManager`, because its only advantage is minor performance boost over the other possibilities and the precision of results is more important than negligible performance increase.

Just a little more advanced, yet a lot more useful, is the one that attempts to find common superclass of all possible types (classes). This one we implemented as the default one and it

is used in almost all cases where type inference has to be fully automatic. It takes the first possible type (class) provided as an initial resulting type and than finds common superclass with each other possible type and stores it as the current resulting type. In case `SmallInteger` and `LargeInteger` are possible types, `Integer` is the resulting type. In case it is `SmallInteger`, `LargeInteger` and `Fraction`, the resulting type is `Number`. In case it could be also `Character` in addition to those 3 numeric types, `Object` is the answer, because it is the closest common superclass.

Finally, for cases where the amount of found types and their precision is more important than full automation, we created `TypesManager` implementation with graphical user interface that displays

- all the possible types from which the user may choose,
- in case of finding a type of a variable it shows all references of the variable (same as when we ask for references of the variable in the Browser),
- once the users picks a type from the list, it displays a list of all superclasses of it (in case we do not want the type to be e.g. `Integer`, but any `Number`),
- a text field with hints where the user may type any class present in the system

3.3.3 Types of Elements Inside Collections

For the case of collections, RBRefactoryTyper provides us possible types of both the collection and the content. RoelTyper, on the other hand, does not offer types of items inside the collection, so we just know that type is set, but we do not know set of what items, which is usually more important than the type of the collection.

In case we use RBRefactoryTyper and find that the variable is being assigned, for example, a set of strings, there are three main possibilities to represent that in UML:

- simply just as a `Set`, ignoring the inner type,
- denoting the type as `Set<String>`, however, this is an unofficial notation that is not explicitly supported by the UML standard and meaning of such notation is not specified and therefore is not understood by most tools, or
- marking the type as `String` with multiplicity `(0..*)` and defining its elements as unique and unordered (which can be considered as definition of a set).

Note that even `String` is a subclass of `Collection`, as well as `Matrix` any many others, but in many such cases, we never wish to display it as `Character(0..*){ordered}`, but simply as `String`. Therefore, we have a constant set of collection subclasses that are displayed directly instead of as a collection. Also there is a collection of subclasses which should never be used directly, such as `SmallInteger`, in which case `Integer` – `SmallInteger`'s superclass – should be used.

3.3.4 Real-time Type Inference

Our implementation of real-time type inference has been inspired by existing solutions [16, 26] and uses Metalinks for enhancing AST by types recording.

We created a `RealTimeTyper` with API that allows to

- start recording of a package (or another package in addition to previous ones),
- stop recording of packages,
- clear recorded types and
- get all possible types of an item (instance variable, method parameter or return type).

The important part is to actually run the code for which we record types, otherwise nothing will be recorded. Whenever method is executed, all relevant information is saved. A single assignment to variable or a single execution of a method results in a single recorded type. If several types could be there, all possible types have to be actually used during the execution to be recorded – this is the main downside of real-time type inference. We can run the code by trying to use application manually (possibly with several use-cases), executing examples or running tests. Whenever a class from recorded package is used by some other (a not recorded) package, even running that not recorded package may improve results of our package in focus (because its code gets executed). After the code is executed, the recording can be stopped (and should be, because recording types reduces performance of the code).

To actually start recording types, Metalinks get attached to the following nodes of all methods of all defined classes and their classes (for class-side methods and variables) inside the package:

- before the AST root of the method itself to record method arguments,
- after each variable assignment node for instance variables to record assignments to record contents of instance variables,
- before each return node to record returned types and
- after the AST root of the method itself in case there is not explicit return as last node to record implicitly returned self (note that this could be found even statically).

The type inferrer remembers all its installed links to be able to remove them. Otherwise, the links could be removed only by recompiling all methods of all classes in the package.

Simply saving the class of the object is not enough, because we want to be able to record classes of elements inside collections. Therefore the type inferrer checks whether the object is a collection and if it is, it records both the collection type and the elements type.

3.3.5 Comparison of Type Inference Techniques

To evaluate characteristics of static vs real-time (dynamic) type inference, we introduce results of applied type inferrers

RoelTyper, RBRefactoryTyper and our RealTimeTyper, along with combination of all these using serial combiner described with adaptors that uses one type inferrer as a backup for another whenever the previous one did not find any type.

In case of static type inferrers, we simply applied each on whole package. In case of the run-time type inferrer, we started logging of types, executed all related tests and examples, then stopped logging.

Given the focus on amount of found types, we did not measure amount of time or memory consumption of each approach.

Table 1 contains the total amount of instance variables, class-side instance variables, methods (possible return types), and method parameters in the package and amount of those with at least partially guessed type. By partially guessed is meant any more information than `Object`, which usually means unknown type, including found types of collection without found types of elements in the collection. The table contains packages where most of them include tests, some do not offer tests, but provide us with executable with examples, `OSWindow-Core` has no examples and very low amount of tests compared to size of the package and `Tool-Diff` and `Traits` have no tests or examples at all in both cases.

As we can see in the table, the real-time type inference provides much more types than static type inference in most of the cases. Package `OSWindow-Core` contains only few tests, therefore real-time type inference provides only marginally better results than static type inferrers. Package `Tool-Diff` has no tests or examples at all, thus no types have been found by real-time type inference. Package `Tests` has no tests or examples either, yet it is being constantly used in the system, even during the type inferrence process itself, giving real-time type inferrer better results than static type inferrers by small amount.

Using both the static type inference and the real-time one combined provides the best results – even in worst case it has results equal to best type infererrer out of the three.

3.4 Transferring the Model to Enterprise Architect

For transferring the model to Enterprise Architect and other tools, we use the XMI format. XMI is a standardized way for transferring UML models that is supported by major UML modeling tools. Each tool has some minor deviations to the standard, such as handling of structure of packages, or requiring certain format of IDs of model elements (Enterprise Architect), especially the return types. Subclasses of UML generator for each target modeling tool can handle these differences.

To export the UML model into a XMI file, there is a XMI exporter for these models available for OpenPonk's UML package that we use without any modification. After that, we use the Enterprise Architect's integrated XMI importer to import the whole model into Enterprise Architect.

3.4.1 Enterprise Architect and Return Types

There are two ways to model the return type of a method in UML.

The first option is to use simple type property of the method itself, which cannot hold information about both a collection (whether it even is a collection) and the type of elements inside the collection – just a single type without any additional info. We consider this a major issue, therefore use the second option.

The second option is to add a return parameter, which can have the same type information as any variable or method argument, including multiplicities. However, in class diagrams, Enterprise Architect displays such return arguments along with all other arguments of a method instead of being the return type of the method.

3.5 Creating UML Class Diagrams

As automatic generating diagrams from models presents several challenges, it is out of scope of our current research. Instead, an empty diagram has to be created by the user, all related classes have to be selected and dragged onto the diagram. Enterprise Architect offers automatic layouting which is of great help compared to the need of manual layouting in some other tools.

4 Discussion and Conclusions

A lot of work has been done in the Pharo community, as well as other communities on types inference. As there is no single method and tool that provides satisfactory results, we based our solution on integrating the best possible pieces and trying to get near the right balance between automation of the process and precision of results.

4.1 Limitations of Our Solution and Future Work

In spite of proving itself very usable for us already, there are limitations in our solution that represent opportunities for future work.

4.1.1 Automatic Generation of Diagrams

Diagrams provide various views on a model, which becomes very important in case of big models of several hundereds or thousands of classes. As explained, right now we generate a model and a corresponding diagram is layouted by Enterprise Architect. As a future work, a technique of generating diagram views (semi)-automatically is an interesting (and supposedly very hard) topic.

4.1.2 UML Representation of Traits

Although UML directly supports interfaces, it does not support traits. Designing a solution how to represent traits in UML would make it possible to further enrich the models. Currently, we ignore traits in our solution.

Package name	Total	RoelTyper	RBRefactoryT.	Real-time	Combined
Zinc-HTTP	2369	76 (3.2 %)	88 (3.7 %)	1565 (66.1 %)	1584 (66.9 %)
Athens-Cairo	1146	34 (3.0 %)	31 (2.7 %)	498 (43.5 %)	513 (44.8 %)
GT-Playground	264	3 (1.1 %)	5 (1.9 %)	58 (22.0 %)	60 (22.7 %)
Ombu	430	21 (4.9 %)	26 (6.0 %)	255 (59.3 %)	263 (61.2 %)
OSWindow-Core	1675	57 (3.4 %)	50 (3.0 %)	64 (3.8 %)	125 (7.5 %)
Tool-Diff	559	34 (6.1 %)	29 (5.2 %)	0 (0.0 %)	34 (6.1 %)
Traits	1823	14 (0.8 %)	13 (0.7 %)	17 (0.9 %)	30 (1.6 %)

Table 1. Comparison of amount of inferred types by each type inferrer

4.1.3 Metaclasses and Class-side Methods and Variables

Another topic, we have not fully addressed, is how to treat metaclasses and how to represent a situation where a variable contains a class. First of all, UML has a support for marking methods and variables as static. However, there is effectively nothing static about class-side methods and variables in Pharo thanks to its full object-orientation (classes are full-blown objects). For example, class `Integer` is an instance of the `Integer` class, which is an instance of `Metaclass`, etc.

So, the question is, should we represent class-side methods and variables as static ones in the class itself? Or should we have not only `Integer`, but also `Integer class` as separate UML classes and put those methods and variables there? This problem needs further research and debate and probably would depend on use of the resulting UML diagram and expected knowledge of Smalltalk-based languages of the person reading the diagram.

Another problem is that, unlike in Java, any variable may contain a class and a class may be passed as a parameter of a method or being returned. The question in such situation is, which data type should it be represented by. Our current solution uses `Class` as the data type in this case. This, however, may not be completely precise, because usually it is not an arbitraty class, but a specific class or its subclasses. E.g. marking the type as `Class` is not precise whenever a variable can contain class `Announcement` or any of its subclass – it should be `Announcement class` instead. The downside of such representation of type is that when we add `Announcement class` as a UML class, it becomes separate to class `Announcement`, so we have to pick whether:

- `Announcement` will have only instance-side variables and methods and `Announcement class` will contain class-side variables and methods,
- `Announcement` will have both instance-side and class-side ones marked as static and `Announcement class` will not have methods and variables, or
- class-side methods and variables will be duplicate in both `Announcement` and `Announcement class`.

Furthermore, in Pharo, there are instance variables, class-side instance variables and shared class variables. As stated in the previous section, UML distinguishes instance variables and static instance variables and we can possibly split a class and a metaclass, such as the shown `Announcement` and `Announcement class`. For now, we stick to having all variables in a single class, such as `Announcement`, where both class-side instance variables and shared class variables are treated as the same thing – marked as static variables, although neither is exactly static and we would like to improve representation of such variables, if that is even fully possible in UML.

4.1.4 Collection Types in UML

As stated in the previous section, for representing collections as data types, our solution uses multiplicities along with ordered/unordered and unique properties. For example, `Set` is unordered unique and `OrderedCollection` is ordered not-unique. The problem is that many collections share these properties, such as `OrderedCollection` and `Array`, or `Set` and `IdentitySet` and such way of modeling collections is insufficient for `Dictionary`, where types of both keys and values might be important. We have not attended to this problem so far and it will be focus of our future work.

4.1.5 GUI Limitations

Our GUI is now rather prototypal and a way too simplistic for use on large projects. Currently, it works only for instance variables and it needs to be modified for method arguments and return values, as well.

Also for instance variables, there are several ways to improve it. Instead of two lists – one of all classes from type inferrer(s) and the second of all superclasses of the selected class in first list, we would like to explore possiblity to show these classes as a tree combining information from both lists.

Next, there is currently no way to select type of collection in case variable contains collection of some elements. We can just mark it is a collection, not knowing whether it is `Set`, `OrderedCollection` or `Dictionary`.

Apart from these, there are some user-experience improvements that would be nice, such as remembering the position for windows opening.

Currently, we have no GUI for settings such as picking what packages to pick, which type inferrers to use with

which adaptors, to which modeling tools to generate, what should be the name of the XMI output file etc. As of now, the user has to configure these by code.

We would also like to provide GUI for our run-time type inferrer.

4.1.6 Displaying Return Types in EA

As stated in description of solution of transferring model to Enterprise Architect, there are two ways to represent the return type of a method – using the type property of the method itself, or using a separate return parameter, each with its downsides – the first lacks the important information about collections, the second displays itself incorrectly in Enterprise Architect as another argument instead of just return type.

4.1.7 Selecting Final Type from Options Offered by Type Inferrers

We also plan to improve the way of selecting the final type from options offered by each type inferrer. For example, in some cases, we can assume 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.

4.1.8 Improving Type Inference Tools

Our type inference results are only as good as the undelying type inference tools such as RoelTyper and RBRefactory-Typer. Therefore, if these tools further improve, our results will, too. Also, another tools may be integrated to further improve the results, such as the most current version of J2Inferer, which did not work at the time we did most of the work presented in this paper, but may do a good job now.

Acknowledgments

The presented research was sponsored by Tomcat® computer GmbH.

References

[1] Scott W Ambler. 1998. *Process patterns: building large-scale systems using object technology*. Cambridge university press.

[2] Vanessa Pena Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. 2013. Agile visualization with Roassal. *Deep Into Pharo* (2013), 209–239.

[3] Alexandre Bergel. 2016. *Agile Visualization*. Lulu. com.

[4] Andy Bulka. 2019. Pynsource Screenshots. Online. http://www.andypatterns.com/index.php/products/pynsource/screenshots

[5] Andy Bulka. 2019. PyNSource [software]. Online. https://github.com/abulka/pynsource

[6] Ravi Chugh, Ranjit Jhala, and Sorin Lerner. [n. d.]. Type Inference with Run-time Logs (Work in Progress).

[7] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. 1999. Why FAMIX and not UML. In *Proceedings of UML'99*, Vol. 1723.

[8] Allen Downey. 2007. Lumpy: UML in Python [software]. Online. http://www.greenteapress.com/thinkpython/swampy/lumpy.html

[9] Michael Furr, Jong-hoon David An, Jeffrey S Foster, and Michael Hicks. 2009. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 1859–1866.

[10] Henricus Johannes Maria Meijer and Oludare Victor Obasanjo. 2011. Efficient data access via runtime type inference. US Patent 7,970,730.

[11] Nevena Milojkovic, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Exploiting type hints in method argument names to improve lightweight type inference. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 77–87.

[12] Nussel Nakamura. 2014. Exploring Pyreverse. Online. https://pythonhosted.org/theape/documentation/developer/explorations/explore_graphs/explore_pyreverse.html

[13] Oscar Nierstrasz. 2012. Agile software assessment with Moose. *ACM SIGSOFT Software engineering notes* 37, 3 (2012), 1–5.

[14] John Plevyak and Andrew A Chien. 1994. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices* 29, 10 (1994), 324–340.

[15] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. 2009. Fast type reconstruction for dynamically typed programming languages. In *ACM Sigplan Notices*, Vol. 44. ACM, 69–78.

[16] Pascal Rapicault, Mireille Blay-Fornarino, Stéphane Ducasse, and Anne-Marie Dery. 1998. Dynamic type inference to support object-oriented reengineering in Smalltalk. In *ECOOP Workshops*. 76–77.

[17] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *Unified modeling language reference manual, the*. Pearson Higher Education.

[18] Michael Salib. 2004. *Starkiller: A static type inferencer and compiler for Python*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[19] Daniel Schweizer. 2000. Exporting MOOSE Models to Rational Rose UML. (2000).

[20] Shane Sendall and Wojtek Kozaczynski. 2003. Model transformation: The heart and soul of model-driven software development. *IEEE software* 20, 5 (2003), 42–45.

[21] Michael Sokol. 2011. Umlify [software]. Online. https://github.com/mikaa123/umlify

[22] Geoffrey Sparks. 2009. Enterprise architect user guide. (2009).

[23] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. 2014. 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*. ACM, 133–142.

[24] Pablo Tesone. 2015. Type inference in Pharo. Online. https://github.com/tesonep/j2Inferer/blob/master/presentation/main.pdf

[25] Peter Uhnák and Robert Pergl. 2016. The OpenPonk modeling platform.. In *IWST*. 14.

[26] Peter Uhnák and Robert Pergl. 2017. Ad-hoc Runtime Object Structure Visualizations with MetaLinks. In *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*. ACM, 7.

[27] unknown author. 2011. Ruby: umlify - generate UML model out of the source code. Online. http://krystianekb.blogspot.com/2011/06/ruby-umlify-generate-uml-model-out-of.html

[28] Martin Unterholzner. 2014. Improving refactoring tools in Smalltalk using static type inference. *Science of Computer Programming* 96 (2014), 70–83.

[29] François Vernadat. 2002. UEML: towards a unified enterprise modelling language. *International Journal of Production Research* 40, 17 (2002), 4309–4321.