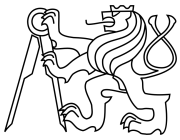
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Code Completion for Pharo System
Student:	Lukáš Komárek
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2017/18

Instructions

- Acquaint yourself with the current state of code completion in the Pharo environment.
- Perform a review of the current state-of-the-art of the topic.
- Formulate a proposal of a better code completer for Pharo focused on flexibility, openness and suitable ratio of effectiveness/accuracy.
- Implement a solution prototype and test it.
- Document your solution.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague November 7, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Code Completion for Pharo System

Lukáš Komárek

Supervisor: Ing. Robert Pergl, Ph.D.

May 15, 2017

Acknowledgements

Many thanks to everyone who has supported me while creating this thesis. Many thanks especially to Robert Pergl, who was always there ready to help with any problem and to support me while writing this thesis. I also appreciate Peter Uhnák for his time he spent consulting me and trying to help me figure out some of the problems I faced. I would also like to thank everyone in the Pharo community who has shared their knowledge and experience about code completion in my survey.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 15, 2017 .....

Czech Technical University in Prague
Faculty of Information Technology
© 2017 Lukáš Komárek. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Komárek, Lukáš. *Code Completion for Pharo System.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Automatické doplňování kódu je funkce, v rámci které se program snaží předpovědět, co uživatel chce napsat, a poskytuje tomuto uživateli schopnost nechat program doplnit části kódu za něj. Automatické doplňování kódu je jednou z těch funkcí ve Pharu, které potřebují zásadním způsobem vylepšit. Současná implementace automatického doplňování kódu je velice naivní a z tohoto důvodu není moc nápomocná uźivateli.

V této práci bych chtěl zanalyzovat jak automatické doplňování kódu funguje obecně a jak vypadá a funguje v jiných integrovaných vývojových prostředí. Dalším krokem bude zjistit, jak Pharo pracuje s automatickým doplňováním kódu a pokusit se tuto funkci nějak vylepšit, což by mohlo vést ke tvoření kompromisů mezi rychlostí a přesností návrhů, které funkce bude nabízet na libovolný vstup.

Řešení, ke kterému jsem se dopracoval je napsat nové automatické doplňování kódu ve Pharu. Bohužel jsem neměl dostatek času k implementaci více než velice ořezaného základu či kostry toho, jak si představuji, že by mělo automatické doplňování kódu ve Pharu fungovat. Vynasnažil jsem se napsat tuto práci takovým způsobem, aby každý s alespoň základními dovednostmi v programování mohl pokračovat v tomto projektu i bez hlubších znalostí o tom jak funguje Pharo.

Klíčová slova Doplňování kódu, Pharo, Smalltalk

Abstract

Code completion is a feature in which a program tries to predict what a user wants to type and gives this user the ability to let the program complete some parts of that code for him. Code completion is one of the features in Pharo that need improving very badly. The current implementation of code completion is very naive and therefore is not very helpful to the user.

In this thesis I am to analyze how code completion works in general and how it looks like and works in other integrated development environments (IDE). The next step is to get an idea about how Pharo deals with code completion and try to figure out a way to make it better, which could lead to compromising between speed and precision of proposals that the feature is going to offer on any input.

The solution I found was to rewrite code completion in Pharo from scratch. Unfortunately, I didn't have enough time to implement more than a very trimmed scaffolding of how I believe code completion in Pharo should work. I tried to write this thesis in a detailed way, so that after reading basically anyone with at least minor coding skills can continue this project even without deep knowledge of how Pharo works.

Keywords Code completion, Pharo, Smalltalk

Contents

Introduction

Code completion is a feature in which a program tries to predict what a user wants to type and gives this user the ability to let the program complete some parts of what he wants to type. When I was presented the option to write about code completion for Pharo system, I was at first focusing on the little window that offers you names of classes or methods that you may want to write when you start to type. But text expansion for example could also be recognized as a form of code completion since you only need to write some keyword and tell the program you are using to expand it into some larger piece of text. These text expanding keywords are also often referred to as snippets. In this thesis I will take a look at code completion algorithms, which are algorithms that decide which predictions and in what order to offer to the user, but this won't take much of my time because as I will explain then they didn't really serve any purpose or hold much value for my thesis.

Now what is Pharo? At the official Pharo web site, you'll read that *"Pharo is a pure object-oriented programming language and a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one)."*[1] At the web site pharoByExample.org you'll read *"Pharo is a modern open-source development environment for the classic Smalltalk-80 programming language. Despite being the first purely object-oriented language and environment, Smalltalk is in many ways still far ahead of its successors in promoting a vision of an environment where everything is an object, and anything can changed at run-time."*[2] I believe these explanations are sufficient to at least understand the most basic concept of Pharo. When I first heard everything in Pharo is an object I didn't know what to expect. Or better yet I always expected something, but it has surprised me again and again in its simplicity. More to what Pharo is and a little bit of how it works follows in

[1] [5] PHARO COMMUNITY. Homepage. *Pharo.org* [online]. [cit. 2016-11-25]. Accessible at: http://pharo.org

[2] [4] PHARO COMMUNITY. About this book. *PharoByExample.org* [online]. [cit. 2016-11-25]. Accessible at: http://pharobyexample.org

the second chapter.

Code completion is one of those features in Pharo that need to be improved very badly, because the current implementation is known to be naive and is therefore not very helpful to the user. In fact it is even worse than naive. The current implementation does not even work correctly with the way Smalltalk syntax works.

In this thesis I am going to analyze how code completion works in general and how it is implemented in other integrated development environments (IDEs). I will also need to get an idea about how Pharo deals with code completion and try to figure out a way to make it better or to find out that it would be best to write it all from scratch. This is not going to lead to compromising between speed and precision of proposals that the feature is going to offer to the user on any given input because this need for compromising usually occurs because when we try to make an algorithm more precise, we give it more information and more information causes the algorithm to take more time to process it. But as I have already mentioned code completion in Pharo is a broken system and finding improvement in this case would mean proposing a solution that would work, hence no comparison is necessary.

Chapter 1

Goal and Methodology

The main goal of this thesis is to find out how to improve code completion in Pharo. That includes acquainting myself with how the current code completion is implemented, what the state of the art of code completion is and formulating a proposal of what and how to implement the new code completion system by either improving the current one or building one from scratch. Another part of the goal will be to try to get as much implemented as possible. Documenting the solution should be a huge part of it so that everyone could understand how to continue the project. Testing should be an important part of the project. Because Pharo is the only environment this work is going to take place in, there is no need for to use any other technologies or languages. Any inspiration from other environments is welcome though.

Unit tests should be made through out the whole system. Integration and component interface tests should be done whereever it would be possible. The reason is that there is no efficient way to implement system tests. If there will be all of the lower level tests though it won't matter much. The system tests will eventually be done by users as they will want to try the new code completion system. The Pharo community is so active that the feedback will only take a few days.

After creating the functional system, UML diagrams should be created and stored somewhere so that it will be possible for anyone to get acquainted with the system. However, it is not possible to finish the project in the given time, which means this is merely a suggestion on what should be done should someone continue this project.

Chapter 2

About Pharo and Code Completion

To start writing about improving code completion in Pharo, I should first clarify a few things. I should explain what Pharo actually is and how does code completion fit into all of it. This chapter focuses on getting the reader to understand the basic concept of all the parts of this thesis so that hopefully even a reader (writing reader, I actually mean programmer) with no previous knowledge about Pharo or Smalltalk will be able to undestand what I will be writing about in the following chapters.

2.1 What is Pharo

First I would like to present the definition of Pharo offered by pharobyexample.org. *"Pharo is a modern open-source development environment for the classic Smalltalk-80 programming language. Despite being the first purely object-oriented language and environment, Smalltalk is in many ways still far ahead of its successors in promoting a vision of an environment where everything is an object, and anything can change at run-time."*[3]

I highly recommend to read about the history of Smalltalk as many programming languages and concepts were influenced by Smalltalk. I would like to mention at least that if someone talks about Smalltalk the understanding usually is that she/he means Smalltalk-80, which is the first version of Smalltalk made publicly available. The first versions were named after the year they were created in, so I guess I don't have to say more than that the first version was called Smalltalk-71, the second version Smalltalk-72 etc. As time went by, a few implementations of Smalltalk programming language and environment emerged and created small but loyal and dedicated communities

[3][4] PHARO fCOMMUNITY. About this book. *PharoByExample.org* [online]. [cit. 2016-11-25]. Accessible at: http://pharobyexample.org

around themselves. Sqeak is one of the Smalltalk environment implementations created by the Smalltalk-80 creators Dan Ingalls and Alan Kay. Pharo was created in 2008 as a fork of the Sqeak implementation and got its first version (Pharo 1.0) released in 2010. Smalltalk-80, Sqeak and Pharo are all open source implementations.

I would characterize Pharo as an untyped programming language and a powerfull IDE combined with a just-in-time interpreting virtual machine where everything is an object. Let's take this sentence apart to understand it better.

You could probably guess that Pharo is developed in Pharo. But the part where it gets really exciting is, that every pice of code in Pharo is open for you to view and even to edit. In other words the IDE is running some of the code there is and at the same time you can be rewriting it. The way this happens is that your code gets interpreted into byte code either when you execute it manualy or save it in the image as a method for example. If you for instance decide to change some method, that the core of Pharo uses often - like a few times a second, you can. If you cause an error in this method though, you pretty much just cause the whole virtual machine to freeze because it will call an error apon an error apon an error etc. and you will have to restart it without the ability to save all the code you wrote from last time you saved the image. Fortunately the image saves temporary changes as well, which you can later in case of such a breakdown reload. So just thinking about it gets you to a realization that the IDE is a big debugger where you write code, run it and are able to debug everything in the process. You can easily change anything you want about your image.

Now to the part where I said everything is an object. The basic idea is that everything you see and do is an object. You have the most basic one called *ProtoObject*. Of this object all the other objects should be a subclass of. The basic object that everyone derives from is called *Object* which is also a subclass of *ProtoObject*. All the GUI elements, all the windows, every class there is, is an object in this little world of Pharo. Because the language is untyped (not strongly typed) you cannot tell what you are going to hold in a variable unless you initialize it yourself. You cannot be certain of what other programmers might send to your class in parameters. Everything will interpret even if no method with a given name exists and you are calling it. Everything might seem to work until you execute such piece of code and you get an error.

There are tuns of great packages (or libraries if you will), which help you to code way quicker. There is also an enormous amount of features you can use like the *Spotter* or the *Finder* which again can make your life so much easier. While the *Spotter* lets you look up any class or method, the *Finder* lets you do a lot of stuff like write an example of what you would like to have as input and what you would like to have as output. The *Finder* then looks through some methods for a solution to this problem you presented and proposes what method you could use.

The best part of Pharo as a programming language is yet to be said. The whole syntax can be explained on the back side of a postcard. It is really amazing how little you have to know about a programming language like Smalltalk to get started right away. The more of a shock it was to me to find out how the current implementation of code completion was designed. But I will get back to that later.

2.2 About Syntax

The following table copied from [1] explains what there is to know about syntax of this language.

Syntax	What it represents
startPoint	a variable name
Transcript	a global variable name
self	pseudo-variable
1	decimal integer
2r101	binary integer
1.5	floating point number
2.4e7	exponential notation
$a	the character ‘a’
'Hello'	the string “Hello”
#Hello	the symbol #Hello
#(1 2 3)	a literal array
{1. 2. 1+2}	a dynamic array
"a comment"	a comment
| x y |	declaration of variables x and y
x := 1	assign 1 to x
[x + y]	a block that evaluates to x+y
<primitive: 1>	virtual machine primitive or annotation
3 factorial	unary message
3+4	binary messages
2 raisedTo: 6 modulo: 10	keyword message
↑ true	return the value true
Transcript show: 'hello'. Transcript cr	expression separator (.)
Transcript show: 'hello'; cr	message cascade (;)

Figure 2.1: Syntax in a table.

What would become the biggest struggle of my work is the method section (the fifth section). More specifically unary and keyword messages. The way the syntax works is that you can write:

```
Dictionary new at: keys first put: values last.
```

After writing "keys", I can either write a unary or a binary message. I cannot start writing a keyword message that I would like to sent to the "keys" receiver without adding parenthesis first. If I decide to write a part of a keyword message it has to be one continuing after the "at:" keyword message (in our case "at:put:" in deed is a keyword message) otherwise you won't get the code you wanted. It will still be syntactically correct, which is actually sometimes the worst part of writing code in Pharo, but it simply won't work as desired.

2.3 What is Code Completion

Code completion in a more general point of view is a feature in a text editor that let's the user complete certain pieces of text quicker. In IDEs code completion is usually implemented because letting the user complete some piece of code for him/her saves keystrokes the user would have to do, which saves time. It also saves time because it lets you write without unnecessary typos and as you don't have to think too hard about if you have written the name of some method correctly or not. This causes you to be able to think more ahead about what you are trying to express instead of what you are typing. While I understand that there are a few programmers, who are typing so fast, that they basically have little use for code completion in this sence, I also know that there is a majority of programmers, who don't type this fast and therefore are used to a feedback from code completion in their IDEs.

Programmers also use code completion to see what methods of a class they could use. This helps especially if the language is strongly typed. In the case of Pharo it tends to get more complicated because as I already mentioned it is a very complex problem to see which method parameters are used in which way. To track all of that could lead to a very time and space consuming and therefore inefficient solution. So what would be sufficient for a programmer to see when he looks for methods to use without knowing the name of this method or even if there is such a method? He would for example have to be able to see the implementation of such a method. But there might be more methods with the same name in different classes. In such a case, should the user be able to look at every one of these implementations or would one be enough?

In Pharo you can set up code completion to show you details of any suggestion. If there are multiple methods that could be applied in the programmers context, the suggestion menu will offer to let you browse all those methods. If there is only one it will let you see the implementation of that method. Have a look at pictures 2.2 and 2.3 to better understand how this works.

This far most of what I have said was more or less optimistic. What can code completion offer, how it can save time and so on. However, if the implementation is naive, code completion can get more and more time consuming and even in some cases loose its purpose. If this is the case in Pharo I will

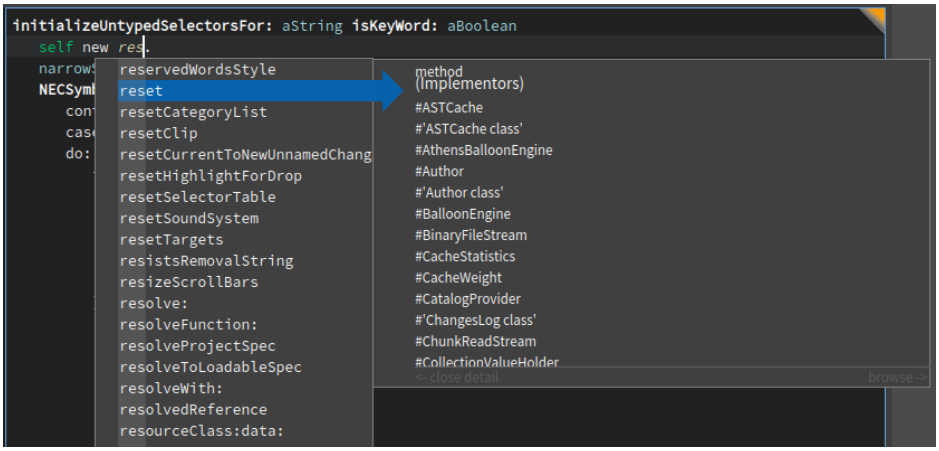


Figure 2.2: Method detail in suggestions menu.

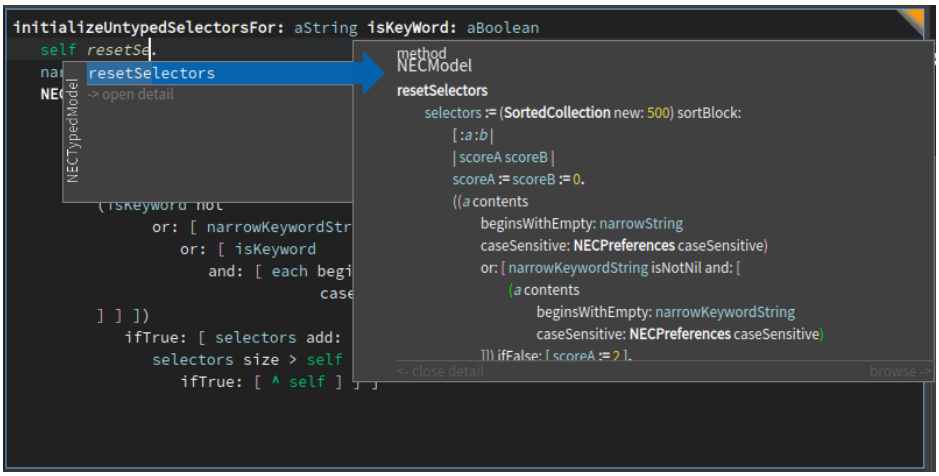


Figure 2.3: Method implementation in suggestions menu.

discuss later. Now I would like to concentrate on the general ideas behind it. Imagine you have implemented a code completion algorithm focusing on characters (meaning class and method names) instead of classes and methods. If you do that, you are very likely to ignore the syntax and you cannot prioritize between the suggestions good enough. You are very likely to prioritize alphabetically, because that is always the easiest way to order a list of strings. But is this a good thing? If you are looking for some method, you want to use, you sometimes have to stop coding for a while and switch to thinking about where to look for this method in the suggestions list. Though this little switch of thoughts might seem meaningless and quick it might break the programmers train of thoughts whitch can be very sensitive to this kind of thougt switching. I believe this is the problem one should face and always keep in mind when

designing or in my case redesigning code completion.

Other than simply completing a word you started or even completing open parenthesis with closed ones, there sometimes are code snippets that let you write some keyword and replace it with some larger prepared piece of code. In Pharo for example this could be that

```
itf
```

would be expandable to

```
ifTrue: [  ] ifFalse: [  ]
```

with the cursor set between the first pair of parenthesis. These kinds of code completions should be customizable for the programmer. Every programmer writes code in a little different way and that means that every programmer would prefer other code snippets or even other formatting of these inserted pieces of code. So in my opinion it is essential to a good code completion to grant the programmer this kind of freedom.

In this chapter I have focused on getting together everything I've read and learnd about code completion and a little about how it looks like in Pharo. I have especially been inspired by the articles [2], [7] and [8]. I highly recommend to read these if you're interested in how to implement a good code completion into an IDE.

2.4 Evaluating Code Completion

In order to get an idea about the improvement of code completion, one would have to set a benchmark which would be able to compare them. But how would one calculate a value of an algorithm?

The answer I found is very practical because it seems quite easy to set up. Romain Robbes presented it in his doctoral disertation [7] as he wrote about change and software and dedicated a chapter to the topic of evaluating recommendations for code completion.

He presented that a good way to evaluate a code completion (in this case I mean code completion algorithm) is to write a program that would simulate a programmer writing code. Then it would look at the suggestions presented and calculate on which index in the suggestions list the desired suggestion is. With the following equation he shows how he would calculate the precision of one entry.

The higher the rank of this evaluation would be, the better. *"For each prefix length we compute a grade G_i, where i is the prefix length, in the following way:*

$$G_i = \frac{\sum_{j=1}^{10} \frac{result(i,j)}{j}}{attempts(i)} \tag{2.1}$$

Where resul ts(i, j) represents the number of correct matches at index j for prefix length i, and a t temp ts(i) the number of time the benchmark was run for prefix length i. Hence the grade improves when the indices of the correct match improves. A hypothetical algorithm having an accuracy of 100% for a given prefix length would have a grade of 1 for that prefix length. Based on this grade we compute the total score of the completion algorithm, using the following formula which gives greater weight to shorter prefixes:

$$S = \frac{\sum_{i=1}^{7} \frac{G_{i+1}}{i}}{\sum_{k=1}^{7} \frac{1}{k}} \times 100 \tag{2.2}$$

The numerator is the sum of the actual grades for prefixes 2 to 8, with weights, while the denominator in the formula corresponds to a perfect score (1) for each prefix. Thus a hypothetical algorithm always placing the correct match in the first position, for any prefix length, would get a score of 1. The score is then multiplied by 100 to ease reading."[4]

I believe this is a great way to evaluate code completion especially for its simplicity. The problem though is that it takes some time to acquire a sufficiently large code base to compute this rank of code completion algorithm on. In the case of this thesis I have decided to evoid building such a code base because it would shorten the time I would like to spend focusing on other chapters of this thesis. I believe though that it doesn't really matter that much because if my task is to figure out how to improve code completion, I only have to figure out a way to change the current implementation to make it work accordingly to the way the syntax of this language works. More to this later.

2.5 State of the Art of Code Completion

I already quoted the doctoral dissertation [7], talking about evaluating code completion algorithms. This paper offers much more than just that. Romain Robbes also tests several code completion algorithms using large code bases from one huge project consisting of almost 200,000 method calls which means this got him almost 200,000 tests for each code completion algorithm and multiple smaller code bases he got from other students and used for this purpose.

The best part of this is that he also tests against the code completion algorithms written in Sqeak, not only in Eclipse for Java. This is really great for me because as I already mentioned Pharo emerged as a fork of Sqeak and is also a Smalltalk-80 language and environment implementation. He mentions

[4] [7] ROBBES, Romain. *Of Change and Software.* Lugano: 2008. Doctoral dissertation. University of Lugano, Faculty of Informatics.

that in Sqeak there are about 3000 classes, 57,000 methods and 33,000 unique method names.

The problem with Smalltalk is that it is an untyped language, so you can almost never be sure what classes are going to be the possible receivers of some method the code completion is just trying to guess from the prefix you're typing. Thanks to this dissertation I don't have to guess too hard which approach would be better than some other.

Of all the code completion algorithms presented, one struck me as unusually well suited for this purpose. It is the algorithm he called "Per-Session Vocabulary". The intuition behind this idea he defines as follows: *"Programmers have an evolving vocabulary representing their working set. However it changes quickly when they change tasks. In that case they reuse and modify an older vocabulary. It is possible to find that vocabulary when considering the class which is currently changed."*[5]

He describes the algorithm the following way: *"This algorithm fully uses the change information we provide. In this algorithm, a vocabulary (i.e., a set of dated entries) is maintained for each development session in the history. A session is a sequence of dated changes separated by at most an hour. If a new change occurs with a delay superior to an hour, a new session is started. In addition to a vocabulary, each session contains a list of classes which were changed (or had methods changed) during it.*

When looking for a completion, the class of the current method is looked up. The vocabulary most relevant to that class is the sum of the vocabularies of all the sessions in which the class was modified. These sessions are prioritized over the other."[5]

I believe the one hour session could be arguable. I mean, shouldn't it depend on how quickly one is coding (perhaps saying one session should be every n keystrokes or every m modified methods), or maybe simply left completely to ones preferences? I believe it shouldn't just be some hidden value. Otherwise according to the results this algorithm was one of the best valued ones. If the author would implement even more known code completion algorithms, there would be some pretty complex ones at the top of the list (sorted by how they scored using this evaluation). Let's have a look at them.

The next algorithm I am going to mention will be an algorithm based on the emph"k nearest neighbors algorithm" [3]. This algorithm is presented in the article *"Learning from Examples to Improve Code Completion Systems"* [2]. The authors have implemented three code completion algorithms and compared them to each other. The first one is a "frequency based code completion", the second one is an "association rule based code completion" and the third one is the "best matching neighbors code completion". They have implemented this in Eclipse for java and tested how the algorithms would per-

[5] [7] ROBBES, Romain. *Of Change and Software.* Lugano: 2008. Doctoral dissertation. University of Lugano, Faculty of Informatics.

form compared to each other. They have found that the "association rule base code completion" better than the "frequency based code completion" and the "best matching neighbors code completion" did best. So it got implemented in Eclipse and was used until a better alternative should occure.

Another improvement came a few years later when Sebastian Proksch, Johannes Lerch and Mira Mezini found this code completion insufficient for several reasons and come up with yet another code completion system that they tested in comparison to the "best matching neighbors code completion". The results they have gotten were very plausible. They have contributed to advancing the state-of-the-art in the following way:

"(1) We extended the static analysis of the best-matching neighbor approach (BMN) and extracted more context information. We show that this indeed improves prediction quality by up to 3% at the cost of significantly increased model sizes by factor 2 and more.

(2) We introduced a new approach for intelligent code completion called pattern-based bayesian network (PBN), a new technique to infer intelligent code completions that enables to reduce model sizes via clustering. We introduced a clustering approach for PBN that enables to trade-off model size for prediction quality.

(3) We extended the state-of-the-art methodology for evaluating code completion systems. We perform comprehensive experiments to investigate the correlation between prediction quality and different model sizes. We show that clustering can decrease the model size by as much as 90% with only minor decrease of prediction quality. We also perform a comprehensive analysis of the effect of input data size on prediction quality, speed and model size. Our experiments show that prediction quality increases with increased input data and that both the model size and prediction speed scales better with the input data size for PBN compared to BMN."[6]

In my opinion these are very interesting ideas on how to implement a code copmletion system. What I want to demonstrate though is not which algorithm is at the moment the best to use for untyped programming languages or which should be tried to be applied to an untyped programming language. I don't even want to explain how they work. All this was to demonstrate one thing: an implementation of code completion in Pharo that would be easily customizable is required. It is essential to have one part that would store all the necessary configuration about code completion, one part with the context about the code it is working with and so on and of course one part that covers the algorithm and the way entries for the code completion are looked for and prioritized. This way it becomes way easier to replace some functionality.

[6] [6] PROKSCH Sebastian, Johannes LERCH and Mira MEZINI. *Intelligent Code Completion with Bayesian Networks* [article]. 2015. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 3.

Chapter 3

Analysis and Design

In this chapter I will first present the current implementation of code completion in Pharo. I am going to focus on the objective side of it and try to keep my toughts to myself. The reason for this is, that the other two sections I should describe what members of the Pharo community think about it and what point of view I am taking on this. In the second section I will show how members of the Pharo community feel about the code completion and I will do that by showing results of a survey I've made. I'll show questions I have asked and explain why I asked them. I will also try to explain why it could be that respondents answered the way they did. At this time I really want to stress out that the reader should keep an open mind about the answers. There are certainly more ways to interpret the answers than just one. In the third section I will finally explain what I think about the code completion, more specifically what problems I found while I was analyzing it.

3.1 Current Implementation of Code Completion

In this section I am going to explain how code completion is implemented in Pharo. I hope that after reading this, the reader will be able to orient much easier in the Pharo NECompletion package containing all the code completion classes. Though there are two code completion systems implemented, NECompletion and NOCompletion, the NOCompletion reuses the same model I am going to explain, only modifies a few classes. For example in NOCompletion there is the NOCController that is a subclass of NECController but it only overrides a few methods.

3.1.1 NECSymbol

The NECSymbol class, that has only static methods implemented, serves as an entry getter. With this methods you can get a list of strings that are names of either classes or methods in your Pharo image. The NECSymbol

class communicates with the Symbol class that has these getter methods implemented that communicate either with the SmalltalkImage class that stores all the global variables and a class is in fact a global variable in the image, or gets all the methods iterating through all the CompiledMethod instances that are in the image and saves the names into a collection. There are a lot of interesting methods in the CompiledMethod that never get used this way. There are methods that get you the source code of the CompiledMethod that you can analyze and try to get an idea about what it returns. Even better there is a method that checks whether it returns "self" or not and many more methods could get implemented here as well that could later be used by the new or improved code completion system.

3.1.2 NECPreferences

This class stores all the configuration of code completion in the image. The information about which controller is at the center of code completion is stored in a dictionary in the Smalltalk tools.

3.1.3 NECController

The NECController gets called when code completion starts. It is essential to how code completion works. It controls which methods get executed in what order. There are three essential methods:

The first is the "codeCompletionAround:textMorph:keyStroke:" method. This static method gets called when code completion starts. That means if "#codeCompletion" as a Symbol (which is kind of like a string in Pharo) is found in a dictionary in Smalltalk tools.

The second one is the "handleKeystrokeBefore:editor:" method. This instance method serves as a handler for keystrokes that are ment for the suggestions menu window such as pressing the down-arrow key or the escape key and so on.

The third one is the "handleKeystrokeAfter:editor:" method. This instance method does the following. If the cursor is at a completion position, it sends the NECContext instance method a message "narrowWith:" with the parameter being the word at the cursor. After the context gets updated this way the controller calls the suggestions menu to update it self as well. At last it checks if there are any suggestions and if not, the controller closes the menu.

3.1.4 NECContext

This is a class of which an instance is representing the context when running code completion. The instance takes care of parsing the edited code, computing the receiver, creating the model of code completion suggestions and so on.

Even though when writing Smalltalk code and writing a method name there can be in some cases two receivers depending on whether the programmer is continuing a keyword method or writing the name of a unary method, the NECContext instance cannot compute more than one receiver and is in general focussed on that it does not care whether or not there are any previous keyword messages already. The thing is writing code using this sort of code completion generates red colored code which signals in case of methods that those methods don't exist. This happens a lot and having to contain code in parenthesis before continuing to write another keyword would not only help the code completion algorithm to find the correct receiver but would as well help the programmer to keep track of his/her code.

3.1.5 NECModel

The NECModel instance serves as a container for entries. The NECModel is an abstract class and its subclasses are only a few. There is an empty model which serves as a model that gets created when there is nothing to complete, there is a typed model that gets created when the context found out what the receiver is and finally an untyped model that gets created in all the unmentioned cases. The NECModel is the most important one for the context and for the suggestions menu of Pharo code completion.

3.1.6 NECEntry

At last I am going to present NECEntry. It is an abstract class and the instance of any of its subclasses serves as a suggestion in the suggestions menu. There are several subclasses which are pretty reasonably named. There is a "NECClassVarEntry", a "NECGlobalEntry", a "NECInstVarEntry" and four other ones which are named analogously to those I have just named.

3.2 Survey for the Pharo Community

On the current state of code completion in Pharo I made a survey for members of the Pharo community. I didn't quite expect how quickly everyone responded. As I was writing about Pharo in the "What is Pharo" section of the second chapter, I mentioned loyal and dedicated communities that emerged around the Smalltalk language and environment implementations. This is exactly the way you can see very clearly how dedicated Smalltalk communities are. I'm personally not that engaged into Pharo, so for me it is not possible to keep pace with the speed at which new issues emerge and at which old issues develop. But this certainly is something to admire.

In total I got 33 responds.

3.2.1 "Do you think the current code completion in Pharo needs to be improved?"

The first question I focussed on what would probably determine whether Pharo developers even feel the need to improve code completion in Pharo.

List of options and responses:

Yes, it is necessary	20	60.6%
It would be nice	12	36.4%
I don't care	1	3.0%
No, I don't	0	0.0%

I believe that it is clear how developers feel about code completion in Pharo. I believe that a huge part in this lies with the problem that it doesn't work accordingly to the syntax of the language. It is sometimes difficult to evaluate something if you're not sure if there is a problem or if you're just not using it right. This as you can see just from looking at the responses is certainly not the case.

3.2.2 "Did it happen to you, that a method name or class name that you were looking for was missing among the suggested?"

The second question is focussed on how the current code completion selects entries, especially methods. Of course this is too precise of a question and I wanted to eliminate the probability of getting a "I don't understand the question" answer, so I decided to formulate it so that every developer could answer even without any knowledge about the implementation of code completion in Pharo or terminology of the syntax.

List of options and responses:

Yes, it happens often	6	18.2%
Yes, it did	19	57.6%
No, this never happend to me	5	15.2%
Other	3	9.0%

The three "Other" responses were:

- "I do not know"
- "don't remember; maybe"
- "there is old (deleted already) methods in completion"

The majority of Pharo developers responded that they feel like there is a problem with the entries selection in code completion in Pharo. What it meant to me at this point was that there would be a model that should be changed, that there is probably something else wrong with it than maybe some bad condition that ruins the finale selection of entries.

3.2.3 "Do you think it would be helpful to be able to customize the list of suggestions while it is suggesting?"

This and the following questions were meant to give me an idea about how Pharo developers feel about some "how should code completion work" details. With this question I was trying to find out if it would be helpful to for example select a receiver class of the method I am looking for. Looking at this question I realize now that I could have formulated it better.

List of options and responses:

Yes, it certainly would	7	21.2%
It would be nice	13	39.4%
I don't care	4	12.1%
It might be confusing	4	12.1%
I don't want that	2	6.1%
Other	3	9.1%

The three "Other" responses were:

- "Only if it helps me to go faster"
- "not clear for me what it means to customize the list of suggestions while it is suggesting"
- "I used to be able to examine the source code of the offered options easily to understand which were appropriate. I can't do that now. This is a major loss."

I believe that some of the Pharo developers were afraid to say that they would like that because one doesn't know if it will be helpfull unless one knows exactly how it would work or better yet unless one tries to use it. Still more than half of the participants said that they would like this option.

3.2.4 "Would you welcome the ability to use code expansion with often repeated code? (like "itf" → "ifTrue: [ ] ifFalse: [ ]")"

The fourth and fifth question were focussed on trying to get an idea from the users about using code expansion also known as code snippets.

List of options and responses:

Yes, I certainly would	9	27.3%
It would be nice	10	30.3%
I don't care	7	21.2%
No, I would not	5	15.2%
Other	2	6.0%

The three "Other" responses were:

- "Discoverability will determine how useful this is. Pharo is still a nightmare of hidden features for new users."

- "This would be nice. In addition, be able to tab to next fields."

We can see now that more than half of Pharo developers would like to have the possibility to work with code snippets. The main feature that would have to be implemented as well would have to be customizability of these snippets.

There is also a great idea in the second "Other" answer. The programmer poposes that the programmer should have the possibility to simply tab through all the places where he should edit some code inside of this expanded code. This should in my opinion also be implemented when selecting a keyword in code completion. Parameters are separated by parts of the keyword method name and it always takes more time getting over those pieces of text than simply pressing tab or some key used for this purpose. This feature is fully working in the Visual Studio 2015 versions and in the newer ones when using a code snippet in C# code.

3.2.5 "Could you write some examples of these code expanding snippets you would consider useful if they were in Pharo?"

This question I wrote to demonstrate the importance of making such a feature as customizable as it could get. It is also important to let the user choose the formatting of his expanded code as the automated formatter is in most cases suboptimal and renders the code unreadable.

List of responses:

- "itf
  → ifTrue: ifFalse:
  it
  → it
  ifTrue:
  but the list should be built with the community."

- "A very cool heuristic we use in our dev environment is the following: suppose I type 'oc' then lookup classes having an O and C in caps, for example OrderedCollection and expand using this information."

- "adding elements to collections; calling super initialize; printing to transcript; "

- "Should be configurable (like Templates in Eclipse)"

- "do:, at:put:, "

- "Can't think of any right now - but important would be an interface for people to define their own. I may want *my* "itf" to be formatted over two lines, and someone else another way. Provide a central location

to upload definitions for the community to browse (and maybe vote) to determine what is popular for defaults inclusion in Pharo. It doesn't need a web interface. All interface could be in Pharo."

- "do:, at:put:, "
- "iter → .. do: [:each | <CURSOR>]
  ts → Transcript show: '<CURSOR>'
  fori → 1 to: <CURSOR> do: [:i ..]"
- "d: → do: [: each | .]
  wt → [.] whileTrue: [ ]
  wf → [.] whileFalse: [ ]
  super → super xxx. (where xxx is the current method with the given arguments)
  in the first 3, '.' shows where the cursor should be left"
- "Write a default "self fail" when saving an empty test method in a test class, logging shortcuts, breakpoint shortcuts, ..."

The first response I got to this question was a suggestion that the list could be a static one and it should be built with the Pharo community. But with the second response you could already see that the ways you could use code completion are too many. One could even argue they are changing as you code depending on what you're coding at the moment, but let's focus on the customizability for the moment. While the third proposition is still a list of propositions what contributers in the Pharo community would come up with the fourth one presents a solution that I had in mind while writing this question. As I have never used Eclipse, I was very glad to be able to see how this could look like and be admired by the users of this feature.

In the sixth answer came the proposition to vote for defaults inclusions in Pharo. These would be some most often used code expansion snippets and as such I would call them the most usefull ones. In the other responses you can notice the need for being able to place the cursor, so you can assume this wouldn't be as simple as placing code, but could get very complex very fast. The only way to avoid this complexity boom would be to perhaps take into concideration one problem at a time. In simpler terms: don't try to solve all problems at once.

3.2.6 "Do you think method names implemented in doesNotUnderstand: method should be recognized by the new code completion?"

To understand how I thought of the sixth question is actually not that difficult. As I was studying how Pharo works and maybe more importantly how it is used by members of the Pharo community, I noticed that a method called

"doesNotUnderstand", which is a method that is called when you send a message to a receiver that is not implemented, gets sometimes used in a completely different way. Imagine you wrote a class and you want to store many variables in this instance. Or even better you want to automatically generate variables and their getters and setters as well and use them in the way you are used to if you would really implement those methods. And there it is this method that get's called every time you send a message to this class. A simple solution to your problem would be to have a dictionary in your instance and to look for this variable in this dictionary inside of this "doesNotUnderstand" method. I mean it works, right?.. The thing is that if you decide this should be a legitimate solution for this kind of problems you make code completion for this kind of methods very very difficult. Even reading code of these classes becomes a nightmare and believe me that for a newcomer it is already hard enough to read code of a complex project even without this unnecessary behaviour. So at this point I just reminded myself that I am doing this primarily to help the Pharo community and therefore I decided to at least figure out how many of the Pharo community members know about this and how many of them would say an ideal code completion should investigate these methods.

List of options and responses:

Yes, they certainly should	3	9.1%
It would be nice	13	39.4%
I don't care	9	27.3%
No, they should not	4	12.1%
Other	4	12.1%

The three "Other" responses were:

- "We do not care they represent 0.001 of the cases it is much more important to have a solid completion first"
- "Do you mean recognising symbols that look like method names e.g. #ifTrue:ifFalse: ? If so, YES!"
- "sounds really hard to do well, but would be nice"
- "I don't understand the question"

The answers were just about as I expected them to be. 13 people said it would be nice to have the code completion analyze these methods as well, but I have got the impression that most people don't really care about it or even if they use it, they believe this is not the way to write clean code. Don't get me wrong. I am the last person to preach about clean code, but even I see this is not ideal to have getters implemented in the "doesNotUnderstand" method.

I also believe that they realise that analyzing these methods for code copmletion is not something that could be done quickly. The way this would have to work then would be to analyze the code already saved to the image first and then change these data accordingly with every other change made to the

image. Often these getters though are implemented using dictionaries which you usually have the chance to inspect at runtime but while writing code. One of the answers mentions using code completion in the way that it would recognize symbols and save them. This would be one way to do that but I believe that this would as well be a suboptimal solution.

3.2.7 "If you have any advice, recommendations or anything you would like to share, please do so here."

The last question was aiming at getting any last advice that I could use trying to figure out a way to do this right.

List of responses:

- "focus on the main cases and make it strong."
- "Good luck and thank you for wanting to improve Pharo."
- "Another idea is to be able to complete keywords.
  Example:
  If I wrote ' aDictionary at: 'key'‘
  then I write an 'i', I would like to see 'at:ifPrensent', 'at:ifAbsent:', 'at:ifAbsentPut:', 'at:ifPresent:ifAbsent:'
  And if I press enter it would complete the #at: message instead of writing the whole method."
- "Make it faster. Like 10x or more. Especially on large images. Also, make the Finder faster when looking in source code. There used to be a reverse index implementation by Camillo Bruni a few years back. This will also improve the user experience a lot."
- "Certainly there's a lot of work to do in the heuristic of what to show first in the code completion suggestions."
- "I'm more concerned about usability of the code completion, i.e., does the overlay disappear fast enough; can I read enough of long similar selectors (width of overlay); could the cursor jump to the first argument position after insertion; I don't want to have to press escape every time I want to navigate the code with the keyboard (not hitting escape will make the code completion capture the key strokes)."
- "Templates like in Eclipse with cursor and argument positioning would be nice https://www.youtube.com/watch?v=zqm4CB1BX6Y"
- "Think about change management: don't change how things work until you have considered how this might affect existing users. Ensure that changes and new features are DISCOVERABLE, and that you create a rich set of help pages in the image. Explain there, or in a clearly linked

page, the philosophy behind your code, how it works, and how a user could easily add their own extensions.
This is an important area of the UX for Pharo, so good luck with your work, and do continue to consult with the community to ensure that your hard work finds a large and pleased audience!"

- "Method name completion should very precise inside the debugger, inspector or playground where the exact type of the receiver is known.
If I have an instance of Foo in the debugger the system should offer me the methods in Foo (and methods from its superclasses)."

- "First it should be nice that the code completion is triggered in each ST code editor (playground, debugger, nautilus,...), because sometimes, there is even no suggestion..."

- "In order to improve accuracy, low cost type inference for dynamic-type language needs to be studied"

- "Mostly works fine, but misses some cases, and the shortcuts would be convenient. One thing that doesn't happen at all is where you have 'oog foo: bar bl' should look for completions for 'bar' that start with 'bl' and also completions for 'oog' that start with 'foo:bl'"

- "About the does not understand, I understand you may want to autocomplete not understood messages to make DSLs. However, this would introduce a lot of noise I think for non-DSL methods, besides the complexity of recognizing valid messages in DNUs. For DSLs what I think would make sense is to have a declarative way to declare the syntax, and be able to plug-in in the auto-completion mechanism to change the suggestions depending on the context."

The suggestions are various, so let's go through the not already mentioned ideas.

The suggestion to make it faster was mentioned. The thing is that making it faster should come after thinking of a heurestic for the code completion algorithm. Focusing on making it faster now could result in a huge waste of time as I'll have to work on changing code completion probably from the ground. As there are so many issues with the current code completion implementation, this is impossible for me to deal with right now. Never the less this should not at all be forgotten - the aspect of time efficiency.

Then there is the aspect of usability of code completion as one of the respondents mentions in the other response. I think this is an important issue to pursue, but these are things that should be changing accordingly to the way we use code completion in the editors. Every user expects something of the menu and of the completion process so someone in the Pharo community should monitor these needs and think of a way this should change in time. I

believe that even if I would make the window larger or "improve" it in another way only a few programmers would really be satisfied.

It is indeed very important to think about change management and about the end users. Making the features easily discoverable and configurable is also very important. Even if you'd make the greatest feature and no one would know about it, no one could use it which would mean it would have once again been a tremendous waste of time.

The following answer has a great point. The completion should change according to the context and even to the editor you're writing in. In the debugger, you already have all the information about types you need to make the method completion as precise as it gets.

Not working code completion in some of the editors should be reported to the pharo developer mailing list and again there should be someone responsible for this area in my opinion. Even if this someone would only take notes of these troubles with code completion and leave the improvements to someone else. That would make corrections to this area much easier and the study and analysis of this field much less complex and time consuming.

For the unexpected complexity of studying code completion in Pharo further study of the automatic deduction of data type (type inference) is required that I will not have the time to follow up on in this thesis. At this point I realized that this thesis cannot represent the ultimate fix of code completion in Pharo but will have to be merely a guide for Pharo developers, who'll want to improve it in the future.

3.3 Problems With the Current Implementation

I believe I have actually already mentioned all the problems I see with the current implementation of code completion that I could think of. But it is always a great idea to conclude what has been said and thought of before proceding to the next step. I am also going to introduce some great features that are missing, that should be implemented in the new improved code completion system.

3.3.1 Not Supporting the Syntax

The most serious problem of all is to me that the current implementation does not support the syntax of the language. When writing

```
Dictionary new at: #key if
```

you should get two sorts of suggestions. There should be suggestions that pop up in the current code completion bound to the #key symbol but these suggestions should be only unary messages. Keyword messages pop up as well but these are suggestions bound again to the receiver #key and not to the instance of Dictionary. The correct way, the way this should work, would be that

the code completion would offer you keyword suggestions like “at:ifAbsent:”, “at:ifAbsentPut:”, “at:ifPresent:” and “at:ifPresent:ifAbsent:”. Though the code completion system offers you “at:ifAbsent:” it only does so because it is implemented in “SequenceableCollection” which is a superclass of the class “Symbol”. So not only the desired suggestion is missing but even if it was in the suggestion menu it would complete the whole keyword interrupting the workflow of the programmer.

3.3.2 Bad Design

After reading the section about current implementation you are probably thinking that the previous problem could be easily fixed. That might not be true. Maybe it is my lack of experience with Pharo, maybe it is my lack of experience as a programmer, but I found the whole design of how it is implemented very confusing.

One thing is that there is no way to clearly see which instance of which class holds what attributes. That the controller holds an instance of the model, of the suggestions menu, of the editor and of the context made sence to me at first. But the more I was thinking about it and going through all the methods exploring which method of which instance is getting called in which order it quickly became a very complex problem. For example if the context holds the instance of the model, why does the controller has to hold it too? Is it really necessary? And why is it that the receiver gets stored twice, once in the context and then again in the model? And most importantly if I decide to change the model where do I have to make the changes? The answer to the last question would be: pretty much everywhere. Again if you would like to implement a new code completion algorithm, you would have to study which methods do you have to override and which can you keep to get called in the already implemented NEC superclass.

The way this should have been avoided in the first place would be to keep in mind the separation of units so that the units would have only limited knowledge about other units and how they work. This is known as Law of Demeter. The way this could work would for example be if there were abstract classes implemented which would work as a guideline to help you understand the basic responsibilities of that class. I will get back to this in the next chapter where I’ll focus on proposing such a design.

Another big mistake was to implement the essential getter for the user favoured controller into the Smalltalk tools. This is a minor problem though and gets fixed very easily as I have done it in the image enclosed on the CD.

3.3.3 Inability to Test Functionality

The worst thing that could happen is when you work on a project and implement or alter some piece of code only to find out that there is no easy way to

test this functionality. The coder basically has only some feedback or none at all to know if what she/he has written even works or in what cases it should work. This is a nightmare for any programmer and the way tests are written for the current code completion in Pharo only prove that they in most cases even have little purpose or no purpose other than to check for some variable and if it's "nil" or not. Sometimes I felt like I am loosing my mind because I couldn't understand how to use some methods or what they do because there were no comments and even the test methods if there even were any were written this way.

3.3.4 Speed of using Keyword Messages

There should be a way to skip over parameters of a keyword. This should probably work using spans of text areas through which you should be able to iterate and implement them one at a time. For this purpose there could be the tabulator key dedicated as a way to iterate through these spans and to signal that the coder has finished writing the keyword there could be the enter key to close these spans up and to push the cursor behind the last span.

3.3.5 Lack of Customizability in the Suggestions Menu

This might be in a wrong section in the thesis, however I am going to present this point here anyway. In my opinion there should be customizable options built in the suggestions menu. What I mean by that is the ability to add a receiver in order to quickly look for a possible method to call or for a possible class to choose and so on. However this is such a minor feature request that not so many Pharo users would welcome as I would, which means I am only adding this as a code completion problem to this section because I believe it would save much time not having to look around in other Pharo tools for basics like this. In the current suggestions menu there is already the possibility to take a look at the source code while you are looking through suggestions. This would be only one step further to the powerful code completion system.

3.3.6 Lack of Code Expanding Keywords

Code expanding keywords have nothing to do with keyword methods. What I mean by code expanding keywords is a set of words that would get included in the suggestions menu for the coder to expand in a predefined piece of code. Understand that predefined would not only mean the text but also the formatting of the text and even where the cursor would be or better yet to which places would the cursor get after for example hitting tabulator or some for this purpose dedicated key. I believe that this would be the greatest boost to the speed of coding especially because of it would eliminate interrupting the programmer's workflow.

3.3.7 Too Slow for Fast Coders

The amount of programmers who write too fast for the code completion to catch up is getting greater and greater. However, to say that code completion is of little or no use to these programmers would be a mistake. Even these programmers could use code completion even on a slow machine, for instance if they could rely on completing certain often used pieces of code without having to check with the suggestions menu first. This would speed up the progress they are making now and could eliminate typos they might not catch at first without the code completion.

3.3.8 Order of Suggestions

Another problem is that in case of NECompletion the suggestions are ordered alphabetically except of those suggestions that directly start with the string that is getting completed. This is always a very suboptimal solution as all of the studies in the articles I have mentioned clearly show.

3.3.9 Getters Implemented via doesNotUnderstand Methods

I believe this is not an issue code completion should focus on in Pharo. If a programmer wants to implement getters this way, and I believe that in rare cases it could even be justifiable, he shouldn't be encouraged by code completion to do this more often than necessary. This being said it certainly would be nice to implement but one must be carefull and beware of overuse of this possibility.

3.3.10 Searching with Capital Letters

Another big relief for the programmer which is not present in the current code completion is the ability to get to the right suggestion using letters which are capital letters in the suggestion string the programmer is looking for. For example if the programmer would be looking for "DummySystemProgressItem", it should be enough to write "dspi" or "DummyItem" or perhaps "dumsysprog" to get this suggestion somewhere to the top of the list. If there were many more suggestions with the same or similar long prefix, this could be the way a programmer could get the desired suggestion to the top of the suggestion list a lot faster. This might take longer to compute and so fast coders could prefer not using this feature. For this reason this feature should be configurable in the configuration class and for that matter of course in the Pharo settings as well.

3.3.11 Width of the Suggestions Menu

Last but not least there is one more thing I need to mention. The fixed width of the window with the suggestion menu is very enoying. Especially if I am looking for some methods with similar names that have the same or similar and very long prefix. Then the only thing you can do if you really want to see the whole names of the suggestions is to select the morph (the graphical object - the actual window) and widen it this way. As you can probably guess this is not an easy fix and certainly not a sufficient solution.

Chapter 4

Realisation

I have presented Pharo and tried to show the basics of how it works. I have described how the current code completion is designed and implemented and done some research in the field of code completing systems and algorithms. Already during this process I have found out that this is not going to be relevant for this thesis because of more troubling issues with the design and the inability to navigate through the current code, which causes great problems when trying to fix it. Finally I have made a summery of what is wrong with the current design and implementation. Through this process I myself have actually been focussed on what I am going to do to make the new code completion system better.

I have slowly came to realize that there is much more work to be done than I can manage in the time I can possibly spend on one Bachelor's thesis and that this is going to be more of a summary of the main ideas that I came to think of during the process of creating it, so that the ones who will come later to finally implement it won't have to go through the same painful process again. For this reason is this chapter going to be almost entirely about the concept instead of the implementation.

Most of what I have done can be seen in the smalltalkhub repository. Using the Monticello Browser you can download the code by inserting the following Monticello registration:
MCHttpRepository
location:
'http://smalltalkhub.com/mc/lukaskomarek/CodeCompletion_FIT_CTU/main'
user: ''
password: ''

4.1 What Have I Done

I have spent a long time thinking about what could be done to keep as much as possible of the current implementation. After a lot of struggle I came to realize

that this is the wrong approach in this particular case. I first need to think of the right way to build it and after I would have done that I need to think of a way to keep as much as possible of the old code without compromising the new system and its readability or functionality. Because I couldn't come up with a solution to almost any of my problems, I have decided to start from scratch and at least implement the scaffolding of the new code completion system. The idea behind this is that after reading what I have implemented anyone could understand the concept behind the design and continue in a similar way I would have if there was more time to spend on this thesis.

After reading the "Problems with the Current Code Completion" section of the previous chapter it quickly becomes clear that the models of the new system can be various and that each other code completion system might be supporting different models with different criteria. There are many difficulties I have had with changing the current implementation that I certainly wouldn't have had, if there would be an abstract model behind it that someone could alter to get different results and as a whole a different code completion system. You might think that the NECModel is an abstract class, which is true actually. However, what I am trying to say is that it is only abstract in the sence that it limits you way too much in what it can and cannot represent. To demonstrate this I am once again going to mention the unary and keyword messages. How would you add another receiver, another text that the user is expanding in case of the keyword message and how would you create and name the new model that would have to represent two types of models - a typed and an untyped one. To implement this with all the attributes currently present in the "abstract" NECModel would have been terribly painful.

I hope I have sufficiently explained what I am criticizing in the design of the current code completion system. The menu on the other hand does not need much correction to solve most of the problems I have presented earlier and that is why I have just copied it to the code completion scaffolding. Don't get me wrong, I believe that the menu should be reimplemented as well, it just isn't so important at the moment.

I believe that the design I have tried to describe in my project would also offer much more opportunities in the testing area of code completion. One of the biggest problems with code completion lies in the tests of code completion. Because this thesis is not about testing software I am going to assume that the reader knows how important it is. Even more so the larger the software gets.

4.1.1 CCConfiguration

This class is actually the same as the NECPreferences class. Nothing should change about the way that user preferences should be saved in some place.

4.1.2 CCContext and CCModel

The CCContext should be different from the NECContext though. This classes responsibility should be to collect all information about the where, what and how of the code completion. It should also (based on what it finds out) create its models that would represent some possible sets of suggestions. These models also hold information like what the possible receiver could be or how to find the suggestions in the current Pharo world. Methods creating the individual entries should be implemented in the CCAlgorithm though.

4.1.3 CCAlgorithm and CCEntry

The CCAlgorithm is a class that was not present in the NEC design of code completion system. The responsibilities of an instance of this class would be to analyze the context and most importantly its models and according to these models create a set of suggestions. This class is abstract because there are countless ways to impelement such an algorithm and it shouldn't really matter to the rest of the code completion classes which algorithm class is being used. It would though matter which models or which context is given to this algorithm. The algorithms responsibility is also to order the list of suggestions in a way that the probably most relevant entries would be at the top of the list. These entries would be instances of unabstract subclasses of CCEntry. These instances should have all the necessary information about what to show to the user, about the way it would complete what the programmer wants to have implemented or completed and so on. These classes and the CCContext with the CCModel classes would be the most interesting ones in the whole code completion system. If you wanted to implement a new code completion algorithm you would only have to override some of these classes to get what you want and then tell the CCConfiguration which classes to use.

4.1.4 CCMenuMorph and CCDetailMorph

The CCAlgorithm instance, after it analyzes the context and creates its entries, is passed to the suggestions menu to present these suggestions to the user. The CCMenuMorph and the CCDetailMorph as I have already mentioned are at the moment copies of the NEC classes. The reason is that it is going to help a lot to change these a little bit and get started right away, not having to implement a whole morph from scratch. Of course even these classes need a lot of work and focus to get some serious improvements.

4.1.5 CCController

The CCController class is the most essential class to the whole system. This class implements and controls the code completion processes. This class should be implemented in a way so that changing almost any part of the system could

be done without having to implement a new controller as well. This would mean that even changing the code completion in a way that you add or modify an entry class would not cause the controller to throw any errors or to behave unexpectedly.

4.1.6 SpanCompletion

The SpanCompletion tag holds two classes, the SCCollection and the SCSpan. The collection class holds spans and the information about what span is currently being edited. This part of code completion should be managed by the controller as well but the part where changes should be made should be implemented in methods of the collection class. To give an example: the programmer would insert a keyword message and write the first parameter and would want to jump to the next one using a tabulator. He would press the tabulator and the controller would look for an instance of SCCollection and if there was one it would pass the tabulator with the editor to this instance. The instance would calculate where to jump with the cursor in the text editor and what code to select for the user to overwrite. So many optimizations come to mind when writing about these span completions. For example in case of completing a particular keyword the spans could be completed as symbols. In case of completing the

```
ifTrue:ifFalse:
```

keyword method, it would create spans resulting in having

```
ifTrue: #block1 ifFalse: #block2
```

inserted into the editor, where #block1 and #block2 would be the spans for the programmer to overwrite. The greatest advantage with this would be that the user would have a better idea about what to write. In case of the old implementation I have often thought some method takes other parameters than it actually takes. This slows down the progress of implementing the intended software and interrupts the workflow of the programmer so finding simple solutions like this one to problems like this is essential.

4.1.7 CodeSnippets

The CodeSnippet tag holds one class at the moment, the CSSnippet. This class represents a snippet that holds the name of the snippet and the content. I have not yet implemented the SCSpans and thought of a way to work with the position of the cursor. For more information look at the implementation and at the class comments. The same goes for all the other classes I have implemented. My efforts were to explain in all the comments as much as possible about the idea behind the concept.

4.2 What Now

The next logical step would be to start implementing classes of the CodeCompletion package. First the CCController, CCContext, CCModel, CCAlgorithm, CCEntry along with the CCConfiguration. Then adding and implementing the SpanCompletion classes and afterwards the CSSnippet class should come next. After this has been accomplished the last step should be to focus on the morph classes and on optimizing the way they interact with the user.

Because of realizing how difficult the testing of code completion system is it should be implemented tests first or at least with testing all the features of the implementation on the programmers mind. Separation of functionalities and class responsibilities should make testing those functionalities way easier.

4.3 Testing

I have in this project created one unit test to show how these should be implemented in the project. This test tests if the SCCollection calculates correctly the absolute value of the offset of a SCSpan in some text.

More unit tests should get implemented the more the project will grow. As the classes will start to interact with each other integration as well as component interface tests should get implemented also.

Conclusion

I have started this thesis with the intent to build a greater code completion in Pharo to satisfy the needs of members of the Pharo community. At first I spent a month analyzing the code and trying to figure out how and why everything works as it does. After this research I have concluded that it will be much easier to implement a new code completion. I have also started to read about different code completion algorithms. As I have proceeded with my research in how to replace the whole code completion system with another one I have found that it is not hard at all. The next logical step for me was to find out what problems did members of the Pharo community see in the current code completion system and what struggles have they had with it. I have decided to make a survey and send it to the pharo community mailing list. I would like to thank everyone in the Pharo community who has participated. In just a week or so I have gotten 33 responds which helped me a lot to see which problems were the most afflicting for the majority of Pharo developers.

As I have started to write about what I have accomplished to this point I have started to realize that implementing the new code completion system that would meet all the requirements that I have thought of would take too much time. That is why I have decided to focus on creating such a scaffolding that anyone could understand and continue this code completion system I would like to have implemented. This included creating comments for as much of the scaffolding as possible. I believe that I have achieved the last desired goal I have set for myself.

I have presented Pharo and the basics about how the syntax and the environment work. I have also proposed a way to evaluate code completion and presented a few code copmletion algorithms that would be great to have implemented and compared in Pharo. Then I have explained a bit about how code completion is implemented in Pharo. Then I have presented what I have learned about code completion in Pharo from the community members and after that I have made a summary of what is wrong with the implementation. There is a lot that came up in this section, which is great because that gave

me a lot of requirements to take in account while creating the design for the new code completion system.

The realisation is what I have described next. In this chapter I have described what I have implemented and after that I have described what would be the next step for anyone who would continue the project.

I believe that I have done everything I could with the knowledge, resources and time I had. I regret that I probably will not be able to continue working on this project because other students in the Pharo community will take over the topic of improving code completion in Pharo and I will not have enough time to work with them on this.

Bibliography

[1] BLACK, Andrew P., Stphane DUCASSE, Oscar NIERSTRASZ and Damien POLLET. *Pharo by example*. Switzerland: Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0.

[2] BURCH Marcel, Martin MONPERRUS and Mira MEZINI. *Learning from Examples to Improve Code Completion Systems* [online article]. 2009 [cit. 2016-11-25]. Accessible at: http://www.monperrus.net/martin/Learning-from-Examples-to-Improve-Code-Completion-Systems.pdf

[3] COVER T. and P. HART. *Nearest neighbor pattern classification.* [article]. 1967. IEEE Transactions on Information Theory.

[4] PHARO COMMUNITY. About this book. *PharoByExample.org* [online]. [cit. 2016-11-25]. Accessible at: http://pharobyexample.org

[5] PHARO COMMUNITY. Homepage. *Pharo.org* [online]. [cit. 2016-11-25]. Accessible at: http://pharo.org

[6] PROKSCH Sebastian, Johannes LERCH and Mira MEZINI. *Intelligent Code Completion with Bayesian Networks* [article]. 2015. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 3.

[7] ROBBES, Romain. *Of Change and Software.* Lugano: 2008. Doctoral dissertation. University of Lugano, Faculty of Informatics.

[8] SPASOJEVI Boris, Mircea LUNGU and Oscar NIERSTRASZ. *Overthrowing the Tyranny of Alphabetical Ordering in Documentation Systems* [online article]. 2015 [cit. 2016-10-25]. Accessible at: http://scg.unibe.ch/archive/papers/Spas14b.pdf

Appendix A

Pharo Virtual Machine Manual

To run the virtual machine you have to do this on a PC with Windows installed or you have to download the virtual machine for Mac OSX or GNU/Linux. To do that go to pharo.org/download.

When you open the content of the enclosed CD you find the readme file and the "CodeCompletion" folder. After you enter the folder you will see a lot of files but the only ones you will need are the CodeCompletion.image and the Pharo.exe files. You just drag the image file and drop in onto the executable and the virtual machine will start running this image all by itself. I highly recommend doing this after you copy this folder onto your machine you are want to run it on to eliminate unexpected behaviour.

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `CodeCompletion` ..... the directory with the VM and the image
  - `Pharo.exe` ..... virtual machine executable (VM)
  - `CodeCompletion.image` ..... image file
  - ..... other less important files