Pitekün: An Experimental Visual Tool to Assist Code Navigation and Code Understanding

Juraj Kubelka, Alexandre Bergel, Romain Robbes
Department of Computer Science (DCC), University of Chile
jkubelka, abergel, rrobbes@dcc.uchile.cl

***Abstract*—Studies show that software developers spend significantly more time navigating and understanding a codebase than actually writing code. Oddly, code navigation and comprehension are poorly supported by current programming environments.**

We present the main lines of the Pitekün programming environment to mitigate the gap between developer information needs and the current tool support. Pitekün uses three techniques to address the gap: (i) a spatial representation of the codebase, (ii) visual cues, and (iii) polymetric views. We conjecture that Pitekün improves navigation in the codebase and in answering the questions developers ask when learning the codebase.

I. Introduction

There are numerous studies on the interactions software engineers have with programming environments. These studies identify developer information needs when evolving a software and level of support that current tools offer [1]. Studies show that software developers spend significantly more time navigating and understanding codebase [2] than writing the actual codebase.

Research question. In our work, we take the identified issues of the current developer tools and we offer new insights regarding to graphic design. We propose to use spatial graphical representation of a codebase, combined with adequate source code metric to provide an accurate understanding of the codebase. In this experiment, we want to know if this approach facilitates the navigation and understanding of the codebase. The research question we address in this paper is:

> *Can Pitekün facilitate navigation and understanding of a codebase?*

Results. The Pitekün implementation is not ready for broader testing. Nevertheless, we identify interest in this solution and recognize that the techniques aid developers in navigating within a codebase and also in understanding it.

Outline. Section II describes developer information needs and weaknesses of current tools. Section III introduces the Pitekün. Section IV presents related work. We conclude with a summary in Section V.

II. Problem Definition

Developer questions. Numerous researches focus on developer information needs when evolving a software. Sillito *et al.* (herein designated as Sillito) present a number of observations about programmer interactions [1]. They define four question categories: (i) *Finding focus point* contains questions when a little is known about the codebase, (ii) *Expanding focus point* consists of questions when developers already have a relevant entity and explore the entity and other cooperating entities, (iii) *Understanding a subgraph* includes questions programmers ask when recognizing the behavior of the entities, and (iv) *Questions over groups of subgraphs* covers questions developers ask when contrasting the observed entities with the rest of the system or when developers identify change impact to the system.

Current tool support. Sillito defines two levels of tool support: *full* when development tool directly answer a particular question; *partial* otherwise. The following table summarizes the findings:

Questions Category	Full	Partial
Finding a focus point (5 questions)	3	2
Expanding a focus point (15 questions)	12	3
Understanding a subgraph (13 questions)	0	13
Questions over groups of subgraphs (11 questions)	0	11
Total	15 (34%)	29 (66%)

They conclude that there are 29 questions which do not have direct tool support. They highlight the lack of support in three areas where current tools should be improved: (i) *more refined and precise questions*, (ii) *context maintenance*, and (iii) *piecing information together*.

More refined and precise questions. Developers are often limited in how precise or refined their questions can be resulting in them asking generic questions. For example instead of asking *"Which classes have this class as fields?"* they ask *"Where is this class referenced?"* While the questions seem similar, the answers may be different. The class could be used as a field only in one class, but can be used in many other places as a local variable or a method argument. The developers then have to conduct additional explorations determining which entities are relevant to the original question.

Context maintenance. If development tools can not directly answer questions (*"Which classes have this class as fields?"*), developers are forced to ask another supported questions (*"Where is this class referenced?"*). However, these tools work in isolation and the context must be maintained by the developers. The developers have to remember the original question, the specific questions, and map them to the individual tool outputs. For example, the question *"How can data be passed to this point in the code?"* may involve exploration of several entities which could involve asking other questions,

e.g., "What data can we access from this object?", *"Where is this type referenced?"*, or *"Where is this method called?"* The developers are forced to remember mapping between each tool output and the corresponding question, and the original question.

Piecing information together. The imperfect tool support requires mentally piecing together information from multiple (often noisy) results. As the current tools usually work in isolation, this process is not supported. For example, to answer the original question *"How can data be passed to this point in the code?"* mentioned above, the answers to specific questions have to be pieced together mentally by developers.

Summary. In this section, we describe four question categories which consist of questions that developers have during software development. Subsequently, three reasons for insufficient tools support are presented. Those three weaknesses are our research.

III. Pitekün: An Experimental Tool

In this section we describe our tool Pitekün, an experimental research tool, which challenges the insufficient tool support presented before. Our approach combines three techniques: (i) spatial presentation, (ii) visual cues, and (iii) polymetric views.

Spatial presentation. Pitekün displays a source code in lightweight fragments that can be arranged freely anywhere on Pitekün's canvas. The canvas is infinite spatial interface. Our decision to use this technique is supported by two reasons. First, the lightweight fragments of a codebase offer greater freedom of arrangement of the codebase. A developer can arrange the parts of the codebase corresponding to a common context, *e.g.,* answers to a question, and the spatial cognition is utilized to the maximum. Second, the spatial presentation supports to observe larger part of the codebase at the same time. The developer can easily observe and compare the codebase. It improves the experience when a mental model of the codebase is created and the developer's understanding may be a more accurate.

Figure 1 illustrates an example of the spatial presentation. There are two groups of the code fragments for which a developer is interested in. The left group contains three methods where the left one calls the two methods on the right. The right group contains three other methods which are indented because the first method calls the next two methods. The freedom of the codebase arrangement according to a common context improves the code understanding and facilitates contrasting of different parts of the codebase.

Visual cues. Pitekün supports the orientation in the codebase by additional visual cues. Currently we display relevant information when the developer hovers the mouse cursor over an entity, *e.g.,* method, variable, or class name. It can improve orientation in the codebase and it may anticipate what could be useful information. This decision is supported by studies which identify that use of landmarks, *e.g.,* borders, paths, boundaries, and directional cues, can improve navigation performance [3].

Figure 2 illustrates an example of the visual cues. There are three methods in which a developer is interested. In order to understand how the variable "lbl" is manipulated, the developer

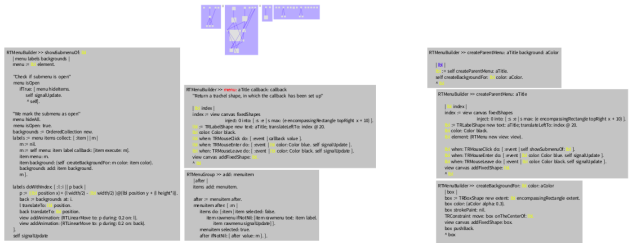

Fig. 1. Spatial presentation in Pitekün. It illustrates two groups of the code fragments in which a developer is interested. The left group (grey boxes) contains three methods, where the left one calls the two methods on the right. The right group (grey boxes) contains another three methods which are indented because the first method calls the next two methods. The blue boxes on the top represent a polymetric views which support overall program comprehension. In the particular case, the classes (blue boxes) are independent and do not share any superclass. It depicts that the majority of a behavior is concentrated in a one class (the biggest blue box).

Fig. 2. Visual cues in the Pitekün. When a user hovers over the each part of source code, relevant information is highlighted among the displayed codebase. This figure illustrates highlighted variable "lbl". If a developer wants to know how a given object is manipulated, this visualization helps him/her concentrate on a particular peaces of the code.

hovers above the variable and all the other occurrences are highlighted. This visual cue facilitates navigation in the code.

Polymetric views. Polymetric views [4] are a lightweight visual approach broadly used in the reverse engineering. It enables depicting of a codebase using various software metrics. It helps a reverse engineer form a mental picture of a system in which he/she works. While the polymetric views are primarily used in the reverse engineering, we experiment when the polymetric views can improve orientation in a codebase and look for which metric can be useful during a development process.

Figure 3 illustrates the current use of the polymetric views in the Pitekün. The larger grey boxes represent particular classes. The class layout is determined by class hierarchy relationship. In this particular case, the size of each class is determined as follows: width corresponds to the number

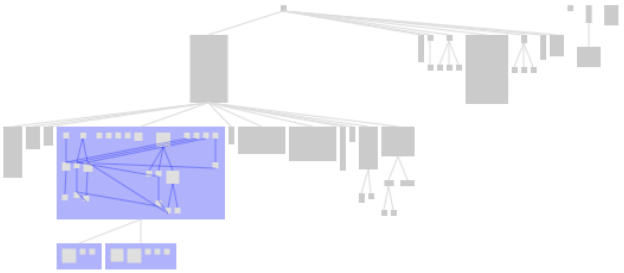

Fig. 3. Polymetric views in Pitekün. The larger grey boxes represent a particular classes in their class hierarchy. In this particular case, the size of each class is determined as follows: width corresponds to the number of members (instance variables), height corresponds to the number of implemented methods. If a developer clicks on a class, it is expanded and methods and methods' call relations are displayed. The size of each method illustrates size of the code.

of members (instance variables), height corresponds to the number of implemented methods. If a developer clicks on a class, it is expanded (the blue box) and methods and method calls are displayed. The size of the each method illustrates amount of the code.

Improving the Tool Gap. In Section II we discuss three weak areas of current tools: (i) more refined and precise questions, (ii) context maintenance, and (iii) piecing information together. Here we look closer at how Pitekün impacts each area.

Pitekün poorly targets the first issue — more refined and precise questions. It affects two other issues — context maintenance and piecing information together. Pitekün addresses those issues by leveraging spatial human memory. A developer arranges a relevant information close to each other and every relevant evidence is further highlighted by the visual cues that stress the reasons why those parts are related. For example when the developer hovers a mouse above a method definition Pitekün highlights all the method calls in other displayed methods or classes.

Summary. In this section, we present the Pitekün and three techniques that we apply during the development process: the spatial presentation that allows for displaying more information and it facilitates contrasting of the greater amount of a codebase; the visual cues that improve orientation in the codebase; polymetric views that offers comprehensive view of the codebase. To our knowledge there is no other tool that integrates all three features of Pitekün.

IV. Related Work

The projects CodeBubble [5] and Code Canvas [6] have a similar approach. They use spatial presentation and display pieces of a codebase in small fragments. Relo [7] and Gaucho [8] also use spatial presentation similar to UML diagrams. a typical clue between each fragment of code are oriented lines which indicate method calls and class hierarchies. Code Thumbnails [3] uses thumbnail images of a file which make any part of the file one click away. JASPER [9] provides views of a code fragments that can be spatially arranged and which are hyperlinked to their original codebase files. Both attempt to reduce the overhead when navigating through a codebase. Pitekün offers more information about a codebase and it aims to improve a comprehension of the codebase.

The polymetric views are used by various tools. For example, Hapao [10] supports reasoning about test coverage of a codebase, Rizel [11] facilitates discovering of an application slowdown. To our knowledge, there is no an application which uses polymetric views for daily development.

V. Conclusion

In this work, we discuss developer information needs and weaknesses of the current tools. The tool gaps are as follows: (i) the tools do not allow asking precise questions, (ii) they do not maintain context among information visible on a screen, (iii) they do not support piecing information together.

We then propose a new experimental tool, called Pitekün, which combines three techniques in order to mitigate the identified gaps: (i) spatial graphical presentation of a codebase, (ii) visual cues which facilitate identification of a relevant code, and (iii) polymetric views which offer additional information about the code and may offer "high-level" comprehensive view about the codebase.

The Pitekün may improve the developer experience as follows:

- Context maintenance: The developer can maintain context in Pitekün by arranging relevant information close to each other and thus exploiting spatial memory.
- Piecing information together: Pitekün may partially assist when the developer pieces information together by the spatial presentation of the codebase and by the additional visual cues which facilitate orientation in the codebase.

Future work. Pitekün is an experimental work in an alpha development stage. Nevertheless, we identify aspects which may improve a developer's experience. We continue to experiment with other visual clues and polymetric views which can improve orientation in a codebase. Consequently, we study how to use visual techniques for answering more refined and precise questions identified by Sillito.

ACKNOWLEDGMENT Juraj Kubelka is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PCHA/Doctorado Nacional/2013-63130188.