Knowledge-Based Systems 21 (2008) 813–825

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Improving program comprehension by combining code understanding with comment understanding

Bradley L. Vinz *, Letha H. Etzkorn

The University of Alabama in Huntsville, Huntsville, AL 35899, USA

ARTICLE INFO

Article history:
Received 31 October 2007
Accepted 28 March 2008
Available online 4 April 2008

Keywords:
Program comprehension
Code understanding
Comment understanding
Knowledge-based system
Heuristic approaches

ABSTRACT

Existing source code-based program comprehension approaches analyze either the code itself or the comments/identifiers but not both. In this research, we combine code understanding with comment and identifier understanding. This synergistic approach allows much deeper understanding of source code than is possible using either code or comments/identifiers alone. Our approach also allows comparing comments to their associated code to see whether they match or belong to the code. Our combined approach implements both our heuristic code understanding and the comment/identifier understanding within the same knowledge base inferencing engine. This inferencing engine is the same used by an earlier well tested, mature comment/identifier-based program understanding approach.

Published by Elsevier B.V.

1. Introduction

Program comprehension refers to any activity that uses dynamic or static methods in order to extract properties of a program so as to achieve a better understanding of the software [4,5]. Development of automated program understanding tools is a prolific, on-going research area [12,16,18,21,26]. Such tools are important for several reasons: first, software maintenance personnel are often not the original developers of the software. Next, personnel turnover can result in programmers having to take over unfamiliar software, sometimes in the middle of project development. Furthermore, code inspections and code reviews also require individuals to deal with unfamiliar software. In these cases, automated methodologies can make program comprehension easier and more complete, and thus can substantially reduce the amount of time required to understand the code.

Automated program understanding tools are particularly important when using object-oriented (OO) software [5], where the learning curve required to use an OO framework can be extensive. OO software is characterized by code scattering: class implementations are commonly constructed in files separate from their respective class definitions. Similarly, classes often inherit from parent classes located in distinct files. Code scattering combined with other advanced OO language features, such as function overloading and runtime polymorphism (virtual functions), can contribute to the difficulty in understanding complex software frameworks.

Early work on program comprehension focused primarily on analyzing the code itself. Research often included comparing data flow and/or control flow graphs derived from analysis of a source program to an a priori library of known constructs [12,13,20]. However, much research in this golden age of code understanding (mid-1980s to mid-1990s) focused on formal non-heuristic program comprehension which was shown by Woods and Yang [27] to be NP-hard. For this reasons, heuristic approaches acquired new importance [27]. One such heuristic approach was that of Harandi and Ning [9]. In their heuristic concept recognition approach, lower level events such as statement, condition, loop, search, sort, etc., were combined within a knowledge base to form higher level events.

More recently (1996 to present), a substantial amount of research in program comprehension has focused on applying information retrieval techniques to descriptive identifier names (function names and variable names) and comments in computer software [4,5,7,15,18]. Methods that exploit such information are sometimes referred to as informal tokens-based approaches [6,8]. The information encoded in comments and identifiers potentially offer valuable clues in obtaining higher degrees of program comprehension, in both clarity and depth.

One major advantage of tokens-based approaches over pure code analyzers is that they have the potential to capture pertinent domain knowledge not encoded in the code events [3]. The reason is that code and informal tokens are defined at different levels of abstraction [3]. Code-based program understanding approaches

* Corresponding author. Tel.: +1 703 369 6422; fax: +1 256 824 6239.
E-mail addresses: bvinz@verizon.net (B.L. Vinz), letzkorn@cs.uah.edu (L.H. Etzkorn).

0950-7051/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.knosys.2008.03.033

return programming-oriented concepts such as searches, numerical integration, sorts, etc. However, human-oriented concepts such as acquire target and reserve airplane seat are not directly related to the code itself. This information appears in the informal tokens such as comments and identifiers, but does not directly appear in the code [24].

However, any approach based on comments alone has the potential problem that comments may not be kept up to date as the code is modified during a maintenance activity. The domain knowledge encoded in the comments may no longer match the associated code. Also, not all code is commented and very often good identifier names are not used: in those situations, comment and identifier understanding is clearly not useful. Similarly, any approach based on code alone is totally ignoring the domain level (as opposed to code level) information that is present in comments [3]. In general, it should be clear that any approach that analyzes only the code or, alternately, only the comments/identifiers is using only half the available information, and thus is providing only a partial view of the software.

Thus, in order to achieve a complete view of software, it is necessary to examine both the code and the informal tokens [24]. As a side benefit, with a combined approach, it is also possible to map the code to the informal tokens, and determine to a certain degree whether the comments match the code. This valuable capability has application to software maintenance and documentation quality assurance. For example, the software analyst should be suspicious of the quality of the internal software documentation if the comments, in general, failed to match the associated program code. This situation could arise if the source file contained very few comments relative to the amount of code, or if the comments were not kept up to date when the code was modified. Conversely, if the comments matched the code everywhere, the software analyst should be suspicious that the abstraction levels of the comments are far too low. For example, the following comment is very uninformative and provides minimal value to the understanding process:

(Example 1) i++; //increment i

In contrast, the following comment is at a higher abstraction level, which could result in a better understanding of the associated code:

(Example 2) i++; //examine the next customer

In Example 1, comment-to-code matching might work well, but the comment is useless. In Example 2, the comment-to-code matching might not be very useful, but both the code and the comment provide useful information to a combined approach.

Our combined code and comment understanding approach merges the different levels of abstraction to form a complete view of program knowledge. The merged effect is an increased level of understanding of the program under analysis.

In our research, we have extended Etzkorn's PATRicia (Program Analysis Tool for Reuse) system [4,5], a mature natural language knowledge-based (KB) program understanding engine which analyzes the semantic aspects of source code contained in comments and identifiers, to handle not only comments and identifiers but also heuristic analysis of the code itself. In this paper, we describe how we use the same knowledge-base inferencing engine that was employed in the original PATRicia system to perform code understanding as well as comment understanding. We expanded this to perform comment-to-code matching using the knowledge base of our combined approach. Specifically, our research focuses on the following goals:

1. A heuristic approach to code understanding that improves on some earlier heuristic code understanding approaches (while still avoiding the NP-hard characteristics of earlier formal code understanding approaches).
2. A combined code and informal token (comment and identifier) approach that will provide a more complete view of the software than has been possible heretofore.
3. An automated analyzer to determine whether the given comments actually match the associated code, and to determine the degree to which they match.

Section 2 provides background information on program understanding approaches with specific attention to code-based and informal tokens-based approaches (primarily information retrieval approaches). Section 2.1 describes our research approach that combines these approaches. The use of the combined approach is illustrated on open source computer software for our proof of concept, described in Section 4. Section 5 introduces a simple code-to-comment match metric that can be used on source code to determine how well comments match the code. Section 6 provides some conclusions drawn from our research. Section 7 suggests future research directions.

2. Program comprehension approaches

Program understanding consists of all activities by which knowledge is gained about a program. It is the task of building mental models of the program for various abstraction levels, ranging from the models of the code itself to models that represent the application domain.

As noted above, program comprehension approaches include code understanding approaches and comment/identifier understanding approaches. A common systematic approach to code understanding is to take a program and construct a high-level representation of it by analyzing the program code's structure [21]. The golden age of code understanding approaches date from the 1980s and 1990s. Comment understanding approaches [3,6], account for higher levels of abstraction derived from program comments and identifier names. Starting in the late 1990s, comment and identifier understanding came to be treated as an information retrieval task [5,6,16].

In the Section 2.1, we discuss some formal approaches to code understanding and why several of these approaches are NP-hard. In Sections 2.2 and 2.3, we discuss some comment and identifier approaches, particularly the more recent information retrieval approaches.

2.1. Code understanding approaches

Code understanding approaches include Rich and Wills [20], Kozaczynski and Ning [13], Woods and Yang [27], and Harandi and Ning [10]. Tjortjis et al. [23] analyzed existing program understanding systems, and divided the types of program understanding systems into formal, rigorous, semi-formal, systematic, and ad hoc. Tjortjis characterized the work of Rich and Wills, Harandi and Ning, and Kozaczynski and Ning as "rigorous."

2.1.1. Woods and Yang code understanding approach

Woods and Yang [27] analyzed the complexity of various program understanding systems. They defined the "simple program understanding problem (SPUP)," which consisted of dividing the source code into a series of blocks, each represented as a graph, and comparing each block to a library of program plan templates represented as graphs. They then proved that simple program understanding problem is NP-hard by using a reduction from the