On Software Odysseys and How to Prevent Them

Simone Scalabrino
University of Molise, Pesche (IS), Italy
Email: simone.scalabrino@unimol.it
Advisor: Rocco Oliveto

***Abstract*—Acting on a software system developed by someone else may be difficult. Performing any kind of maintenance task requires knowledge about many parts of the system. Therefore, program comprehension plays a lead role in software maintenance, above all when new resources are added to a project. At the same time, acquiring full knowledge about big code-bases can be utopian, because it requires a big effort if no sufficient documentation is provided. In this paper I present TIRESIAS, an approach able to suggest a subset of important software artifacts which are good entry points for newcomers. The suggested artifacts can be used in order to acquire knowledge about the system in an initial stage. TIRESIAS uses a knowledge graph to model the references among source code artifacts and to find (i) the artifacts that lead to acquire the widest knowledge about the system and (ii) the most important artifacts that are worth keeping in mind. The approach is validated through a case study conducted on a software system and three professional software developers.**

***Keywords*-software maintenance; program comprehension; recommender systems**

I. INTRODUCTION

Understanding code means traveling through it, from class to class, going through methods and, again, moving to other classes. Such a behavior is encouraged by IDEs, which allow developers to easily jump from one artifact to another. The goal of the journey is to gain enough knowledge about the code to perform a specific maintenance task. In OOP the problem of localization [1] makes code navigation less effective. Nevertheless, such a practice is still of paramount importance when other sources of information such as models and documentation are lacking.

Previous work on concept location [2] [3] [4] [5] focused on the selection of code artifacts relevant to a specific topic. While reducing the effort of newcomers, allowing them to concentrate only in artifacts related to a specific task, concept location just helps to detect *what* is relevant, without suggesting *how* to browse it. Indeed, some important questions remain unanswered: which code artifacts are worth to understand at the very beginning? Where should a developer start from in order to explore most of the code without being forced to abort the original journey and to start another one? In short: how to avoid code odysseys? Program comprehension is a trial-and-error process, in which developers read code, follow links to other software artifacts and they build a mental (cognition) model of the system [6]. Starting from the wrong source code artifact could delay the creation of such a model and, thus, slow down the entire comprehension process.

The main contribution of this paper is TIRESIAS, a new approach for recommending code artifacts from which a newcomer should start analyzing a code-base. Unlike previously defined approaches [7] [8] [9], TIRESIAS can suggest both methods and classes and it does not just indicate *central* artifacts, but also artifacts which help developers gain the widest knowledge about the system or a part of it.

II. APPROACH

TIRESIAS aims at recommending a subset of significant code artifacts to a newcomer. Such a subset should allow the developer to (i) gain knowledge about the core parts of the system and (ii) start from parts of the code that help them to acquire a good amount of knowledge about the system. TIRESIAS consists (i) in the creation of a *knowledge graph*, containing all the source code artifacts of a system, (ii) in the computation of two scores, *knowledge quality* and *knowledge coverage*, and (iii) in the recommendation of artifacts with high *knowledge quality* or *knowledge coverage*.

A. Knowledge graph

The knowledge graph models all the references from parts of the source code to other parts. The knowledge graph is a directed graph in which each vertex represents a class or a method in the source code. If a source code artifact A contains any reference to a source code artifact B, the graph contains an edge from A to B. More specifically, an edge from a vertex A to a vertex B is created under the following conditions: (i) method A uses method/class B; (ii) class B contains method A and A refers to any part of B (e.g., a field); (iii) method A overrides method B and calls it (using `super`); (iv) class/interface A extends/implements class/interface B; (v) class A contains fields with type B. All the edges in the knowledge graph model a possible jump made feasible by any IDE clicking on the name of the artifact. For example, if a method x contains a local variable n of type K, it is possible, in any IDE, to use the function "Go to class definition" on K in order to jump to K from x. The knowledge graph contains heterogeneous artifacts, i.e., methods and classes. Such artifacts are treated in a different way: the body is considered only for methods and not for classes. In other words, if TIRESIAS recommends reading a method, it means that the developer should read its body; if, instead, it recommends a class, it means that the developer should understand the class behavior as a black box.

B. Recommendation of source code artifacts

In order to recommend the best artifacts to a newcomer, two aspects are kept into account: *knowledge quality* and *knowledge coverage* of artifacts. The first one aims at measuring how important is to understand and remember a source code artifact in order to easily explore other portions of code; the second one measures how much knowledge a newcomer can acquire if (s)he starts to analyze a source code artifact browsing code following forward links to other artifacts and possibly going back.

Knowledge quality (KQ) is estimated computing the centrality of the artifacts in the knowledge graph, obtained using the PageRank algorithm [10]. Given a directed graph, such an algorithm assigns to each vertex a score which represents the probability that a developer ends in it while randomly browsing code. The higher the probability, the higher the centrality. Therefore, it is likely that a newcomer needs such knowledge in order to understand many parts of the source code. Such a metric was previously used in this context [7] [8].

On the other hand, the *Knowledge coverage* (KC) is a completely novel metric, and it is computed as:

$$KC(v) = \frac{1}{2} \sum_{p \in n(v)} PKC(p)$$

where PKC indicates the partial knowledge coverage, computed as:

$$PKC(v) = KQ(v) + \frac{1}{2} \sum_{p \in n(v)} PKC(p)$$

In both the formulas, $n(v)$ returns the set of vertices directly connected to the target vertex v on the graph; furthermore, to avoid loops, $n(v)$ has a memory and it never returns the same vertex twice, so that each vertex is taken into account only once. The higher the distance between the vertex v of which we want to compute the knowledge coverage and a given vertex p connected to v on the graph, the lower the weight of the knowledge quality of the vertex in the computation (in a exponential way). The hypothesis is that the farther the vertex p, the lower the probability that the developer focuses on it when (s)he starts from v. Such an effect reflects what happens in code browsing.

The artifact recommendation is made from two ranked lists containing source code artifacts: the first one is ordered by knowledge quality, while the second one is ordered by knowledge coverage. In the scenario in which the newcomer uses TIRESIAS, (s)he starts from the first list to look at some important artifacts, while (s)he uses the second list to select source code artifact from which it is worth starting to deeply analyze code. TIRESIAS was implemented in a prototype tool as a plugin for IntelliJ IDEA. It is worth noting that TIRESIAS can be used either on the whole project or on specific packages: in such a mode, it supports the scenario in which a user is only interested in a part of the system to complete a specific task.

III. CASE STUDY

In order to evaluate the effectiveness of TIRESIAS, I conducted a preliminary case study. The *goal* of the study is to understand whether it is possible to suggest artifacts which are good entry points for newcomers using the TIRESIAS and whether both the metrics are useful. The *context* of the study is composed by (i) MyUnimol, an Android app consisting of about 100 Java classes and more than 16 KLOC, and (ii) the three main contributors of the project.

I asked each participant to assign a score between 1 and 4 to 20 source code artifacts, 10 recommended by TIRESIAS (5 with the highest knowledge coverage and 5 with the highest knowledge quality) and 10 not recommended by the approach (5 with low knowledge coverage and 5 with low knowledge quality). The score would indicate the probability that they suggested to a newcomer to concentrate on a specific artifact. The median score assigned to each artifact by the participants was used as ground truth. I considered an artifact as "suggested by developers" if the median score was greater than 2.5 and as "not suggested by developers" otherwise. In order to evaluate the approach, I computed precision, measured as $\frac{|Sugg_B|}{|Sugg_T|}$, and recall, measured as $\frac{|Sugg_B|}{|Sugg_D|}$, where $Sugg_D$ is the set of the artifacts suggested by developers, $Sugg_T$ is the set of artifacts suggested by TIRESIAS and $Sugg_B = Sugg_T \cap Sugg_D$.

TIRESIAS achieves an overall precision of 0.4 and an overall recall of 0.67. More specifically, using only knowledge quality, TIRESIAS achieves 0.20 precision and 0.50 recall while using only knowledge coverage, it achieves 0.75 precision and 0.75 recall. The results suggest that developers prefer recommendations coming from the novel metric, *knowledge coverage*, which is promising and worth to investigate more in depth.

IV. RELATED WORK

Previous work addressed the problem of the selection of central classes in software projects. Steidl *et al.* [7] compared different scoring algorithms to rank artifacts by their centrality in order to suggest them to newcomers. To compute the scores, the authors use a dependency graph which contains only classes. Şora [8] used PageRank [10] to identify key classes in a software system. With the same goal, Ding *et al.* [9] used H-index to rank classes by centrality. The main differences between the proposed approach and the state-of-the-art are (i) the concept of *knowledge coverage* and (ii) the fact that TIRESIAS can recommend both classes and methods.

Other work treated the problem of introducing new resources to a running project: Cubranic *et al.* [11] proposed Hipikat, a tool which helps developers to access the group memory of a project; Canfora *et al.* [12] devised an approach which recommends appropriate mentors to newcomers.

V. FUTURE WORK

Future work will be addressed at expanding the study and at evaluating TIRESIAS with a controlled experiment, in order to understand whether newcomers find real benefits in using its recommendations.

REFERENCES

[1] A. Dunsmore, M. Roper, and M. Wood, "Object-oriented inspection in the face of delocalisation," in *Proceedings of the 22nd International Conference on Software Engineering*, 2000, pp. 467–476.

[2] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004, pp. 214–223.

[3] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *Proceedings of the 15th International Conference on Program Comprehension*, 2007, pp. 37–48.

[4] B. Cleary, C. Exton, J. Buckley, and M. English, "An empirical analysis of information retrieval based concept location techniques in software comprehension," *Empirical Software Engineering*, vol. 14, no. 1, pp. 93–130, 2009.

[5] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[6] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[7] D. Steidl, B. Hummel, and E. Juergens, "Using network analysis for recommendation of central software classes," in *19th Working Conference on Reverse Engineering*, 2012, pp. 93–102.

[8] I. Şora, "A pagerank based recommender system for identifying key classes in software systems," in *10th Jubilee International Symposium on Applied Computational Intelligence and Informatics*, 2015, pp. 495–500.

[9] Y. Ding, B. Li, and P. He, "An improved approach to identifying key classes in weighted software network," *Mathematical Problems in Engineering*, vol. 2016, 2016.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[11] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: a project memory for software development," *Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.

[12] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?" in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, 2012, p. 44.