Empir Software Eng (2013) 18:857–858
DOI 10.1007/s10664-012-9237-5

EDITORIAL

Guest editorial: reverse engineering

Martin Pinzger · Giuliano Antoniol

Published online: 30 November 2012

Reverse engineering aims at obtaining high-level representations of software systems from existing low-level artifacts, such as binaries, source code, execution traces or historical information. Reverse engineering methods and technologies play an important role in many software engineering tasks and quite often are the only way to get an understanding of large and complex software systems.

Given for example the task to understand certain aspects of the test code in large software systems, such as Eclipse. How would you figure out the answers to the following questions:

- How are the tests in Eclipse organized and where is the test code located?
- What is tested by a unit test, test plug-in, and a whole test suite—what is not?
- What was the reason for running these tests?
- What influences the test execution environment?

The sheer mass and complexity of information to be handled by the developers working on such large software projects is huge. Eclipse size is in the millions of lines of code and ten thousands of classes. Consider for example the Eclipse 3.3 release, just the `org.eclipse.jdt.ui` component contains 47,243 methods. Looking for the answers by manually browsing the code is an option, but obviously may not be practically feasible or at least not the most efficient one. Automatically extracting the required information from the implementation, configuration files, historical information and bug reports, and presenting them to the developers in a compact and easy to understand way is the preferred one. Achieving that, is one of the main goals of reverse engineering. This special issue consists of five distinguished papers in the research area of reverse engineering, that are briefly introduced in the following.

M. Pinzger (✉)
Software Engineering Research Group, Delft University of Technology, Delft, The Netherlands
e-mail: m.pinzger@tudelft.nl

G. Antoniol
SOCCER Lab. DGIGL, École Polytechnique de Montréal, Québec, Canada
e-mail: antoniol@ieee.org

The paper "What your Plug-in Test Suites Really Test: An Integration Perspective on Test Suite Understanding" deals with the challenge of testing loosely coupled software systems built with a plug-in-based architecture. Although developers use extensive automated test suites, they provide little insights which of the many possible configurations are actually tested. To remedy this problem, the authors study its nature by interviewing 25 professional software engineers working on Eclipse. Based on these findings, they introduce five architectural views that provide an extensibility perspective on plug-in-based systems and their test suites which they evaluate with three open source systems.

During the evolution of software systems, changes lead to portions of source code with unrelated responsibilities being grouped together. To measure this phenomena, we typically use structural coupling and cohesion metrics. The paper "Using Structural and Semantic Measures to Improve Software Modularization" argues that considering only structural aspects to measure such unrelatedness is not sufficient to improve the modularization of a software system. They present an approach using structural and semantic measures to decompose a package into a set of smaller, more cohesive packages. The findings of their evaluation shows, that the approach obtains a more cohesive modularization and, regarding responsibility, also a more meaningful one.

Related to coupling, the paper "Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software" presents an approach to assess the impact of changes. The basic question is: "Given that I change the source code in this file what other files will be affected by my change?" The presented approach combines conceptual (the extent to which domain concepts and software artifacts are related to each other) with logical coupling (the extent to which software artifacts were co-changed in the past). The authors evaluate their approach with data about co-changes obtained from the repositories of five open source systems. The results show that the combined approach significantly improves the accuracy when compared to traditional approaches that only use one source of information.

Along this line of research, the paper "On the Impact of Software Evolution on Software Clustering" investigates the usefulness of such historical information for automated software clustering. The paper reports on several clustering experiments with the data of ten open source Java software projects. For each project, the authors compute and compare the quality of the clusters obtained with static structural source code dependencies, co-change dependencies (alias logical coupling), and a combination of them. The results show the best accuracy is obtained by the combination of both data sources.

The rich information stored in software repositories is also subject to the paper "Studying Re-opened Bugs in Open Source Software". It presents a study of bug reports obtained from an issue tracker to answer the question: "What indicates whether a bug report will be re-opened?" In general, developers want to prevent re-opening a bug report because that means extra costs for fixing it. In a study with three large open source projects, the authors show that the comment and last status of a bug report before it is closed are the most significant indicators for re-opening. According to their findings, it pays off to check them before setting the status of a bug report to fixed and closed.