Extracting Code Examples from Unit Test Cases

Mohammad Ghafari
DeepSE Group @ DEIB – Politecnico di Milano
mohammad.ghafari@polimi.it

Abstract—Understanding how to properly use APIs of large libraries is difficult, error prone, and time consuming. Software developers resort to study code examples to learn APIs. Several approaches have been proposed to mine these examples, but the sources from which they mine examples as well as their mining approaches hamper their applicability in some practical scenarios. Unit test cases seem to be an additional source of significant API examples [2], which may overcome the aforementioned difficulties [1]. Synthesizing meaningful examples from tests not only improves the applicability of current code recommendation systems, but also facilitates providing up to date API examples to augment documentation. However, mining examples of API use from unit tests is a non-trivial task and arises several research challenges summarized in this paper.

Different structure of test code from client code inspires the need to adopt a specific mining approach to obtain examples from tests. This paper draws three main research questions (RQ1-RQ3) coupled with a number of potential solutions that constitute my research agenda to this end.

RQ1: Classifying Test Types. Not all unit tests represent correct API usages. In practice, some tests may be in charge of checking a negative behavior like an exceptional condition. Distinguishing negative tests is a fundamental requirement for extracting meaningful examples. Nonetheless, this distinction is sometimes blurred and difficult to automate. Actually, a negative test does not necessarily raise an exception; it may, for example, invoke a method which returns null to indicate that no data is available. My intuition is that a technique for automatically classifying tests may also be based on mining test comments and the messages provided by assert methods. In addition, if necessary, I may exploit a CrowdSourcing platform where a qualified crowd is asked to classify tests.

RQ2: Identifying Meaningful Examples. Unit test cases are commonly structured in three parts setup, execution, and oracle. The setup part instantiates the class to test and puts it into a particular state, then execution includes all the method invocations on the object under test, and finally oracle validates the state of affected objects through some assertions. Each sequence of non-assert statements (setup and execution) followed by assert statements represents a sub scenario.

Test code comprehension and analysis is not easy. On the one hand, all sequences of test code do not always represent a useful example. For instance, the above test includes two sub scenarios, add and remove, each of which separately illustrates an API usage. To synthesize meaningful examples, these sub scenarios must be extracted from the given scenario (insert and then immediately remove), which instead only serves a testing purpose. However, automated identification of practically meaningful sequences of APIs participating in a test is non-trivial. On the other hand, extraction of the code contributing to such usage examples from a given test is non-trivial because they usually share the same program entities. For example, both add and remove need the same setup part (lines 1 and 2), although, they follow different executions and respectively. So, existing slicing techniques cannot be adopted directly to extract examples from test code, and there is a need to adapt a technique which also considers the test structure during slicing. Unfortunately, this is not easy. It is not discernible, for example, whether a method invocation belongs to the setup or execution parts of a test. Even in case of the oracle part that is often associated with assert methods, there may be an invocation (e.g., line 4) to provide access to class states to be verified, but it can be confused with method invocations belonging to the execution part. In fact, despite clear logical differentiation of test parts each having its own purpose, distinguishing them is hard even for developers. This is probably why existing approaches to link production code and tests are only successful to find the class under test but not the tested method.

RQ3: Resolving Test Dependencies. Unit tests often need to instantiate auxiliary or dependent classes to fulfill the preconditions needed to exercise the test. To minimize the side effects resulting from other objects being called from the unit under test, unit tests usually use stubs and mock objects to simplify the setup procedure and focus testing efforts on a single unit under test. These kinds of dependencies indeed belong to testing environment and should be replaced in the synthesized examples with a corresponding program entity with which the tested unit should collaborate in practice. Besides, many tests may belong to the same API, thus perhaps produce repetitive examples that hinder the representativeness of the synthesized examples. That means there is a need to apply a clustering technique to discard redundant ones and build a set of representative examples.

REFERENCES