Automatically Recommending Test Code Examples to Inexperienced Developers

Raphael Pham
Software Engineering Group
Leibniz Universität Hannover
Hanover, Germany
Raphael.Pham@inf.uni-hannover.de

Yauheni Stoliar
Leibniz Universität Hannover
Hanover, Germany
Yauheni.Stoliar@se.uni-hannover.de

Kurt Schneider
Software Engineering Group
Leibniz Universität Hannover
Hanover, Germany
Kurt.Schneider@inf.uni-hannover.de

ABSTRACT

New graduates joining the software engineering workforce sometimes have trouble writing test code. Coming from university, they lack a hands-on approach to testing and have little experience with writing tests in a real-world setting.
Software companies resort to costly training camps or mentoring initiatives. Not overcoming this lack of testing skills early on can hinder the newcomer's professional progress in becoming a high-quality engineer.
Studying open source developers, we found that they rely on a project's pre-existing test code to learn how to write tests and adapt test code for their own use. We propose to strategically present useful and contextual test code examples from a project's test suite to newcomers in order to facilitate learning and test writing.
With an automatic suggestion mechanism for valuable test code, the newcomer is enabled to learn how senior developers write tests and copy it. Having access to suitable tests lowers the barrier for writing new tests.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Miscellaneous*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*

General Terms

Management, Human Factors

Keywords

Testing, Newcomers, Recommendation, Examples

1. INTRODUCTION

Computer Science students sometimes leave university with a dismissive and negative view on software testing [6]. Students reported to have problems in writing test code — the learning curve and technical barrier kept them from applying their theoretical knowledge in a real-world setting. This is problematic, as such inexperienced developers will soon apply for software engineering positions. Companies will have to take care of this gap in testing skills, however, providing training camps or mentoring initiatives is costly. Here, we see a need to support such inexperienced developers in adopting a good and healthy testing culture. The question arises: "How can novice developers be supported in writing more and better tests?"
We propose to automatically suggest test cases from the project's tests to newcomers so that they can adopt the presented testing techniques. In a previous study about the testing culture on a social coding site [7], we observed the role of existing tests for project newcomers. New contributors heavily relied on existing test code to write their own tests. Often, they searched for tests in the existing code base that would suit their needs and copied and adapted them. Being able to access tests lowered the barrier for writing tests: analyzing existing tests, users observed and learnt how other users wrote tests.
The inexperienced developer shall use the suggested test code example as a basis for writing her own test code. To this end, the most *suitable* test code must be selected among the set of written tests: First, the suggested test code must serve as a good basis for the new test, i.e. only a relatively small amount of changes need to be made. In this sense, the test code suggestion has to take into account the current changes to the production code. Second, the suggested test code shows features of the test framework that the newbie can learn and use.

2. RELATED WORK

The idea of using existing code examples as a learning aid is not new. In 2011, Barzilay et al. describe it as an "act of embedding a code segment from an example into a software system being developed" [2]. While staying on the theoretical level, Barzilay et al. suggest an overlay structure to example embedding as a part of code reuse and identify several categories, practices, and possible tool application areas. Although providing many vital examples and explanations [1], there is no mention of the peculiarities of example embedding for software testing.
The concept of example embedded programming spawned different tools: Edwards et al. present an eclipse plug-in which operates on a set of pre-written code examples, highlighting their occurrences in code [4]. These stand-alone

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2803202

examples are used in real-life code and allow for learning without having to leave the IDE. Similar to Edwards, we use tracing to determine suitable code and operate on previously created content, however this content does not need to be explicitly created for learning purposes — instead, we use content that is already existing (such as the existing test suite). Hummel et al. introduce "Code Conjurer", a tool that uses queries to search for code examples [5]. Again, "Code Conjurer" operates on a pool of pre-written examples, albeit provided by an external source. Brandt et al. present "Blueprint", an IDE plug-in that provides code examples on demand — including context information for it [3]. It mines regular websites and fetches the descriptions from around the code examples. Our search showed quite a few attempts to apply using example-centric programming. However only few of them mention the writing of software tests as a possible application area. In contrast to our approach, both "Code Conjurer" and "Blueprint" require a user written and explicit query and do not provide any insights into the matching decisions to the user. Targeting inexperienced developers — such as graduate students during onboarding — operating query-less and offering insights for learning is crucial. Additionally, we limit the search set for test code to the project's own test suite in order to guarantee applicability of the suggested test code. Direct and easy applicability of the test code is important as it reduces frustration and facilitates adoption of our approach among inexperienced developers. Lastly, our approach supplies the newcomer with a browsable list of test code examples that is automatically tailored to the changes that the newcomer has made. This, we hope, facilitates a learning effect.

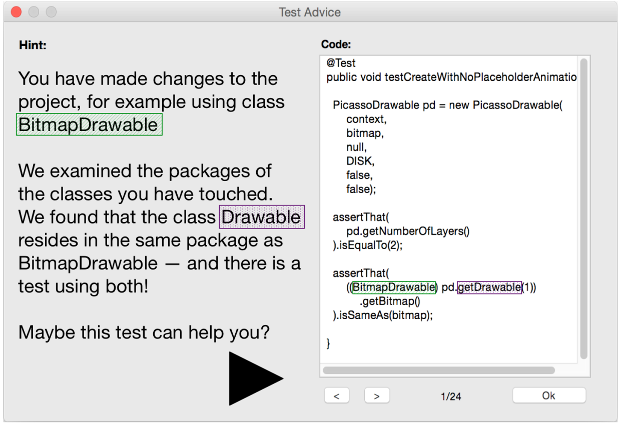
Qusef et al. describe a pragmatic approach to find correspondence between unit tests and the tested classes [8]. Although our matching approach was influenced by Qusef et al., their idea is quite the opposite. It traces contents of tests to a specific classes, thusly improving the refactoring process. However Qusef et al. do not expand their approach to finding test code from source code.

3. OUR APPROACH

Our approach focusses on a newly hired graduate with *some* experience in coding and *little* experience in writing tests. Different challenges arise when trying to suggest suitable test code examples: How do we automatically understand the changes made to the production code — in order to find a suitable example of test code? How do we know which test case is suited to serve as a basis for the currently needed test?

3.1 Test Recommender

In an ongoing effort, we have implemented our approach as an Eclipse IDE Plugin, called *Test Recommender* (see Fig. 1). The use case for Test Recommender (TR) is as follows: A new graduate makes changes to production code and when she is done, she triggers Test Recommender by the push of a button in her IDE. TR analyzes the changes made by the newcomer. Next, Test Recommender searches through the set of existing tests and looks for any test code that is similar to what the newcomer might need in this situation: Is there any test that tests something remotely similar to what the newcomer needs (according to her code changes)? The Test Recommender window pops up and suggests a list of suitable tests, ascending from best suited to marginally suited. Tests that did not match the Test Recommender's heuristic will not be included in that list. Lastly, the user browses through the set of presented tests, choses one and uses it to write her test.



You have made changes to the project, for example using class BitmapDrawable

We examined the packages of the classes you have touched. We found that the class Drawable resides in the same package as BitmapDrawable — and there is a test using both!

Maybe this test can help you?

```
@Test
public void testCreateWithNoPlaceholderAnimatio

  PicassoDrawable pd = new PicassoDrawable(
      context,
      bitmap,
      null,
      DISK,
      false,
      false);

  assertThat(
      pd.getNumberOfLayers()
  ).isEqualTo(2);

  assertThat(
      ((BitmapDrawable) pd.getDrawable(1))
          .getBitmap()
  ).isSameAs(bitmap);

}
```

Figure 1: Suggesting a suitable example of test code in the IDE of a newcomer.

3.2 Helping Newcomers

The main view of Test Recommender features two panes. The left pane is informative and explains why this test has been recommended. For the newcomer, this facilitates an understanding of why a particular test has been shown to her. Also, it can give her an idea of where to find other tests and how to approach the search for tests in general.

The right pane shows the recommended test code example and is *browsable*: The newcomer can click through all suggestions using the buttons below the right pane. Each time, the rationale on the left pane and the suggestion will update. Seeing senior developer's test code in action should give the newcomer an understanding of how such tests can be used. Both the rationale on the left as well as the test code on the right are highlighted. This facilitates a quicker grasp of important parts of the code.

3.3 Technical Approach

We want to match current changes in the production code to tests in the existing test suite. Our goal is to present a subset of *suitable* tests to the novice in the manner of "Look, here are some tests, these might be useful to you." The suggested tests should be an aid to the user, giving her an idea of how to write a test and how to use the test framework and take advantage of its features. Here, our general rationale for finding a suitable tests is "suitable test code will use the same types as the current changes to the production code" (types can be classes or primitive data types). For example, a test that covers a certain class will at least use this class to instantiate an object of this class. Going through the set of tests one by one, we compare types used in the changes to the ones used in each test and rank these tests accordingly. At best, the newcomer changes a class that is already covered by a written test case — this test case would certainly refer to this class in test code and it would be suited for adoption. At worst, the newcomer introduces a new class that is not mentioned at all in the test suite. Our tool will attempt to look for *similar* classes in the test suite that reside in the same package: Maybe tests for other classes in the same package as the current code changes could be useful as test examples? For example, classes from the package 'view' and tests for classes in the package 'view' have similar instantiation patterns. If the newcomer adds a new view object with no tests, being presented with a test for another, similar view object can help.

The following steps illustrate how our tool works: **First, we need to *identify the changes*** made to the project without any interaction from the user. We leverage information of the versioning system (Subversion): We compare the changed code base to the last commit in order to find which lines of code have been touched.

Second, we *search the set of changed lines for classes used* and distinguish our findings into three categories: *Project classes* are user written, more unusual and point to a stronger connection to the test code (if found there as well). *Non-project classes* are part of the core libraries of the programming language and are more common, such as *String*. These reside somewhere in the packages of the programming languages. Lastly, *primitive data types*, such as *int* are fundamental building blocks of the programming language and have no specific package location. The set of project, non-project classes, primitive types and their package locations make up the so-called *change set*.

Third, we *analyze the test suite* in a similar manner to the change set, i.e. extract project classes, non-project classes, and used primitive data types of *each test*. This results in a so-called *test set* for each test.

Fourth, in the *matching step*, we compare the *change set* to each *test set*. Our matching takes into account five categories of findings:

1. Matches in project classes: the strongest indicator for suitability — the candidate test uses the same user-written classes as the current changes.

2. Similarity of project classes used in the change set and project classes used in the test set: do project classes used in the test originate in the same package as the current changes to the production code?

3. Matches in non-project classes. Similar to step 1 but the classes originate outside the project, be it the core library of the programming language or any 3rd party library.

4. Similarity in non-project types: see step 2, but with non-project classes.

5. Exact match in primitive types: the least significant match indicator — a test uses the same primitive types as the current changes. Our similarity concept will not work for primitive data types as they are fundamental building blocks of the programming language and have no specific package location.

4. EVALUATION

Our approach is primarily aimed at graduates who are joining the software engineering workforce. To better understand the impacts of our approach, we performed a first tentative evaluation. We wanted to understand wether they agreed with the provided matching and prioritization of a suitable test code and what advantages they recognized in using our approach. Our population consisted of 10 students in computer science (nine) and computer engineering (one) who were just about to get their degrees. These students included four master and six bachelor students. All of these students had been programming during lecture courses of their recent semesters. Their programming experience ranged from one year up to eight years. Their testing experience ranged from none to up to two years.

In our evaluation process we used paper printouts of source code as our Test Recommender tool was not finished yet. We used the open source project 'Picasso' [1] as a source for real-world production code and test code. We selected a relatively small commit that only edited one class. We removed the developer-provided test code that belonged to these changes from the project's test suite and applied our approach to the remaining tests. Our tool ranked 23 of those tests according to their hypothesized suitability as a 'copy&paste'-template for a newcomer, from high to low. We picked six test code suggestions: two test codes with highest ratings for suitability, two with middle ratings and two with lowest ratings. We were interested in whether these ratings were at least reflected in how students would assess the suitability of these test code suggestions for writing their own test code. Using two tests for each quality level helped to smooth out minor deviations.

Each participant was presented with the source code of the commit (as a printed listing, presented with formatting and syntax highlighting similar to Eclipse's code editor), asked to examine the code and to explain the character of the changes made. We rated their level of understanding: Seven participants were able to explain both of the two main changes to the code while the three explained only one.

After that, each participant was shown the six suggestions for test code. We asked each participant to examine the tests one by one and explain them for better understanding. Each participant was asked to select one of the tests that —according to them — was best suited to be used as a 'template-for-reuse' in this situation, i.e. for copy & paste & adaption. Eight of ten participants selected a test code suggestion from the two top matches. The remaining two participants opted for a middle match.

We asked participants to explain their reasoning behind their selection — we wanted to know what our participants were looking for when looking for test code to copy from. Participants preferred test code that used classes that were affected in the changes. Seeing *how things worked* was important to our participants. The main motivation was to see how these objects could be used and invoked: *"That test shows me how to use the constructor of the class used in changes."* One participant decided to go with the a longer test, calling bigger size an advantage, because *"bigger test shows more"*. However, short and concise style of test code can seem more accessible and less intimidating. One participant selected a middle-matched test code for its shortness which made it *"easy to understand"*.

Next, we asked the participants to describe briefly what advantages they saw in being presented with test examples. Participants welcomed the concept of test code suggestions as templates (*"It can be directly used as a template"*) over having to write tests from scratch. Test code suggestions helped them to get going with testing, *"It saves my time,*

[1] https://github.com/square/picasso

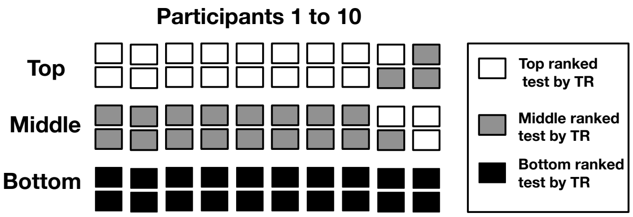


Figure 2: Participants' ranking of six test cases according to their usefulness as a copy-template.

I can start right away". Would our participants like to use TR when developing? On a scale of 1 ('absolutely not') to 10 ('definitely yes'), five participants chose rating 8 and four participants chose rating 6. One participant chose rating 7. As a final task, we asked the participants to order the remaining five tests according to their perception of usefulness as a template for copying. Unsurprisingly, the tests with no correspondence to the changes (only few classes in common) were quickly ranked last. Fig. 2 shows the ordering that participants chose, distinguished in three tiers: Test Recommender predicted white tests to be most useful to newcomers, gray tests had a middle rank and black tests had a very low rating. Overall, the ranking by our participants matched the ranking of the Test Recommender. This is most evident when considering low ranked tests — deciding that a test code suggestion is *not* relevant seems easier. As already indicated in the participants' multi-facetted reasoning for selecting the best suited test code suggestion, the ordering differs more regarding better ranked suggestions.

5. DISCUSSION AND FUTURE WORK

Our approach helps newcomers in adopting systematic software testing and facilitates learning of real-world test techniques. Our tool suggests suitable test code to newcomers based on the changes they made to the production code and saves them the hassle of searching a project's test suites on their own. Finding just the right test in a big test suite can be daunting for a newcomer.

We put additional effort in implementing a browsable and newcomer-friendly "single-click" solution. As we have seen in our first evaluation, the browsability feature of our approach is important when dealing with newcomers. Newcomers have little experience and can work best with a diverse offering of examples. This facilitates an understanding of what techniques are available and how senior developers use them. The reasoning for choosing one test case over another was diverse, which supports the need for a browsable offering of test examples.

Our evaluation is exploratory and of little statistical significance. It is more qualitative in nature. Our population of 10 students participants is small and we cannot claim generalizability. However, we deem students (who are just before their graduation) a suitable evaluation population for our actual target population: graduates who have just entered the job market. We do not think that the experience level in graduates between university and their first day at work differs much. Although the results of our evaluation are promising, we are missing a confirming evaluation that shows a measurable advantage of our approach in a real-world setting. This final evaluation is currently in preparation.

6. CONCLUSION

This work introduces the concept of strategically and automatically suggesting test code to newcomers. Newly hired and inexperienced developers — such as new graduates — sometimes have problems in writing test code in a real world setting. Analyzing existing and working test code can help in understanding how senior developers approach testing and facilitate learning. However, finding a suitable test code can be challenging for the new hire when the test suite is big — and asking senior developers can become bothersome. **Our Eclipse plugin *Test Recommender* analyzes the changes a newcomer has made to the production code and suggests a suitable example of test code from the existing test suite.** This can help the new hire in overcoming the technical learning curve faster by copying test strategies from senior developers. We evaluated our approach with promising results: most of our participants embraced our approach and would like to use test recommendation in real world software development.

7. REFERENCES

[1] O. Barzilay. Example embedding. In *10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 137–144. ACM, 2011.

[2] O. Barzilay, O. Hazzan, and A. Yehudai. Characterizing example embedding as a software activity. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 5–8. IEEE Computer Society, 2009.

[3] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

[4] J. Edwards. Example centric programming. *ACM SIGPLAN Notices*, 39(12):84–91, 2004.

[5] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[6] R. Pham, S. Kiesling, O. Liskin, L. Singer, and K. Schneider. Enablers, Inhibitors, and Perceptions of Testing in Novice Software Teams. In *22th Intern. Symposium on the Foundations of Software Engineering, FSE*, 2014.

[7] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *Int. Conf. on Software Engineering, ICSE*, pages 112–121. IEEE Press, 2013.

[8] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley. Scotch: Slicing and coupling based test to code trace hunter. In *WCRE*, pages 443–444, 2011.