ANALYZING STUDENT CODE AFTER INTRODUCING PORTFOLIO ASSESSMENT AND AUTOMATED TESTING

Author: Per Lauvås

Abstract

Teaching students how to write program code is difficult. Teaching students how to write good program code is even harder. When educating future programmers, it is of great importance that they know the concept of technical debt, and how good code will reduce technical debt. In this paper I show you how the quality of student code changes when introducing two new aspects in a programming course; portfolio assessment and test coverage. Code produced from two succeeding classes was analyzed using an analyzing tool called SonarQube. The results indicate that both portfolio assessment and automated tests improve the quality of the code, but not with the same effect.

Keywords: Automated testing, portfolio assessment, programming, teaching.

1 BACKGROUND

The course PG3100 is the third Java programming course for the programming students at Westerdals, Oslo ACT. The students are in their second year of a Bachelor in IT. They are familiar with the basic syntax of the programming language, and are now introduced to new areas of use for the language. These areas include automated testing, communicating programmatically with a database and web applications with servlets. Learning goals related to automated testing are found in the course description:

After completing the course, the students will
- know what automated testing is and its benefits.
- be familiar with the concepts: mock and stub.
- be able to test an application using unit tests and integration tests.

The students participate in two hours of lectures and two hours of lab exercises each week for 12 weeks.

In 2013, the assessment of the students in the course consisted of an evaluation of three mandatory exercises, each accounting for 10% of the final grade. A final written exam accounted for the remaining 70%. The students received a written feedback for each exercise with information on good and less good aspects of the written program. The feedback was provided by the lecturer, and the students did not have the opportunity to enhance the quality of the code based on the provided feedback.

2 TECHNICAL DEBT AND AUTOMATED TESTING

There exist multiple types and definitions of technical debt, originally described by Cunningham (1992). The type of technical debt we are referring to in this paper is the one originated from poorly written code caused by a desire for speeding up development pace (short-cuts). Or it can be caused unintentionally because the developer does not have the required skills and knowledge to write good
code. By adding or changing a piece of code in a code base, a developer can introduce a future cost of fixing that piece of code.

Using automated analyzing tools for identifying technical debt has been described earlier (Codabux & Williams & Niu 2014, Vetro 2012). Research using code analysis on student work has also been performed (Vetro, Torchiano, and Morisio 2010). But analyzing student code after introducing automated testing and portfolio assessment is an addition to existing research.

Evaluating code quality related to teaching automated testing is not new. Proulx (2009) found improvement in code quality and the understanding of object oriented program design when using TDD already from the CS1 level. But this improvement is not evaluated using a code analyzing tool like SonarQube. Proulx finds the improvement through reports from instructors of upper level courses and improved results in pretests for these courses. She also uses surveys to evaluate the improvement.

3 \hspace{1cm} \textbf{INTRODUCING PORTFOLIO ASSESSMENT AND AUTOMATED TESTS}

In 2014 we wanted to introduce some new elements into the course in an attempt to improve the quality of the code written by our students - future programmers. The curriculum for the course was mainly the same, but the number of lectures on automated testing was doubled from 1 to 2 (of the total of 12 lectures). A demand for test coverage for all three tasks was introduced. So, in addition to writing code to fulfill program descriptions, the students had to write automated tests for the programs as well. The students decided for themselves how to run the tests, although nearly all of them run them through their IDE (Eclipse or IntelliJ). The students had to provide documentation on the level of test coverage in their deliveries using a variety of tools (like EclEmma).

The three mandatory tasks now constituted 60% of the final grade (20% each). The final written exam was upheld, accounting for the remaining 40% of the final grade. The students still received feedback on each tasks, but now from tutors in addition to the lecturer. These tutors were students having taken the course the year before, and they had all achieved a good grade (A or B).

The introduction of tutors providing feedback was necessary to lessen the lecturer’s total amount of time spent on assessment. The tutors were provided guidance on how to provide feedback, and what to look for in student code. They were also presented a suggestion on how to solve each task.

The major change was the fact that the students now had the chance to enhance the program code (and test coverage) based on the provided feedback. The initial submission of a first version of the solution was not mandatory. But the first submission was required for receiving the feedback. All students were to hand in a final version of all of their tasks in what might loosely be called a portfolio.

Portfolio assessment grew in popularity in educational circles in the 1990s. Instead of evaluating student performance at a particular point in time (a traditional exam), the student can provide a selected portion of relevant work to show what they have learned. An important part of the portfolio is student self-reflection (Paulson, Paulson and Meyer 1991).

Our use of portfolio assessment did not include a selected portion of work, and did not emphasize self-reflection, although the initial submit was to include self-reflection so that those evaluating the code could more easily provide quality feedback. The self-reflection was not a part of the assessment regarding the grade. The students could not select what to include in the portfolio. The portfolio included the three completed tasks with some complementary documentation.

Our use of the term portfolio relates to the fact that it was a collection of work performed by the student over a period of time (one semester). But more importantly, it was the result of an iterative delivery. Cain and Woodward (2012) describe use of portfolio assessment when teaching programming in a constructive learning perspective. In their model presented in the paper, formative feedback is important to help students construct appropriate knowledge. Their suggestion is small, frequent feedback. In our course, we chose a combination of a more formal written feedback for each
task, supplemented with the possibility of discussing the tasks and feedbacks in class with a tutor or the lecturer.

Of the three mandatory tasks in the course of 2013, two were reused in 2014. In 2014, one task involving data structures was replaced by a task on unit testing. The task involving communicating with a database was given to both classes, and is the focus of attention when analyzing code for this paper.

There were no other major changes made to the course. The curriculum was nearly identical, the lecturer and number of lectures was the same. The main book of reference was the same, although a newer version was used in 2014.

4 ANALYZING STUDENT CODE

The solutions were handed in through a Learning Management System (LMS). The two classes involved were the classes of 2013 and 2014. Both classes had the same background of courses taken prior to this course. The 2014 class had two lectures on automated testing prior to the task. The class of 2013 had only one such lecture. There were 14 participants in the class of 2013. In 2014, 48 students handed in the assignment for review, but only 46 participants delivered the same assignment in their portfolio.

In order to evaluate a possible change in student code, we need to know what the programming task is about. In short, the students were to implement a method `copyFile(String filename, String tableName)` in a class `DBHandler`. The filename describes a path to a file with metadata for the table (number of columns, column names and column data types). After the initial table metadata, the file provided table data (the rows in the table). The `tableName` is the name of the table to be created in a database. The `DBHandler` is expected to have a reference to another class `DBConnector` providing a connection to a database. So, the `copyFile` method should use the information in the metadata file to create a table and populate the table with the provided data. `DBHandler` should also have a method `getTable(String tableName)` returning the content of a table with name `tableName`.

The task description only describes the two classes: `DBHandler` and `DBConnector` as well as a client class with a main method to run the code using a provided file with table (meta-) data. But the students were encouraged to add other classes and methods if that would improve the solution.

A tool called SonarQube was chosen for analyzing the code. SonarQube is an open source project that is easy to install and easy to use. It covers a multitude of languages, including Java. Installing SonarQube involves installing a SonarQube server, configuring a database to store the results from code analyses and installing an analyzer to perform the analyzing of the code. In this project, the most commonly used analyzer, “SonarQube Runner”, was installed as the analyzing tool. The default settings for the runner were used when analyzing the code.

SonarQube is presented as a way of controlling the level of technical debt in your code base, or as they say: “Put your technical debt under control”. We wanted to see if the perceived technical debt in the code for the two classes would differ. SonarQube Runner assigns a technical debt value to each problematic issue found in the code. Examples of such issues include complexity issues, not following code conventions, exception handling etc. The value indicates how much time it would take to fix the issue. An issue is found if one of the many described rules within the analyzer is violated. These rules may be changed by adding or removing rules, or by re-estimating the cost associated with the rule. In our analysis, we used the default rules with the default time estimates. In addition to calculating a total debt for a project, SonarQube Runner calculates a technical debt ratio. The ratio makes it possible to compare projects of different size. As the number of participants in our two classes varies, we use this ratio in our analysis.

In addition to the technical debt measurement provided by the SonarQube Runner, we wanted to pay extra attention to some of the difficulties we have experienced that our students are facing in their
The second year in their bachelor program. The first is following the Single Responsibility Principle (SRP). SRP, within the context of object oriented programming, states that every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. In our case, it would be impossible to follow this principle when only using the three classes described in the task at hand. The DBHandler, for example, would have to be responsible for reading and parsing the input file in addition to communicate with the database. That would clearly violate the principle. So, we wanted to count the number of classes the students used to solve the task. A high number of classes will indicate a better understanding of the SRP.

The second difficulty is keeping the method size small. There is no definite maximum size of a method, but keeping a method small makes it easier to understand and maintain. So, we wanted to investigate how many methods the students used within their total lines of code (LOC). A high ratio of LOC / number of methods would indicate a large average method size in a solution.

Although automated tests were a part of the evaluation in the class of 2014, the test classes were omitted when analyzing the code. We wanted to see the effect of the “production code” when automated tests were written, not the tests themselves. By using filters when using the SonarQube Runner, we could easily leave out all test classes.

In addition to evaluate the quantitative data produced by SonarQube, we wanted to look at some examples of student code. Maybe we could exemplify a change of coding behavior for some of the students using the portfolio assessment.

5 RESULTS

The results are displayed in the table below.

<table>
<thead>
<tr>
<th>Year</th>
<th>Number of participants</th>
<th>Total classes</th>
<th>Total LOCs</th>
<th>Total methods</th>
<th>Average classes</th>
<th>LOCs /Methods</th>
<th>Technical debt ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>14</td>
<td>40</td>
<td>1967</td>
<td>116</td>
<td>2.9</td>
<td>17</td>
<td>7.9%</td>
</tr>
<tr>
<td>2014, review</td>
<td>48</td>
<td>210</td>
<td>11429</td>
<td>895</td>
<td>4.4</td>
<td>12.8</td>
<td>5.6%</td>
</tr>
<tr>
<td>2014, portfolio</td>
<td>46</td>
<td>242</td>
<td>13184</td>
<td>1063</td>
<td>5.3</td>
<td>12.4</td>
<td>5.2%</td>
</tr>
</tbody>
</table>

Table 1. Code quality metrics for the class of 2013, the initial exercise delivery of 2014 (“2014, review”) and the portfolio delivery of 2014 (“2014, portfolio”).

The totals in the table are present to indicate the size of the analyzed code. The numbers of interest are the three rightmost columns: Average number of classes, lines of code per method and technical debt ratio. We see that for all three of those metrics there is an improvement from 2013 to “2014, review” and from “2014, review” to “2014, portfolio”.

The main difference in the delivery of 2013 compared to “2014, review” is the requirement of automated tests. The main difference between “2014, review” and “2014, portfolio” is that the student has received written feedback from a tutor or the lecturer. The results therefore indicate that automated tests have a higher impact on code quality than portfolio assessment.

But what is not revealed by the initial analyze of the data is how the level of code quality is distributed among those who are able to write tests, and those who are not. When examining the portfolio, we saw that we had three main categories of code:

1. Code with practically no coverage.
2. Code with some coverage, but less than 50%.
3. Code with a high degree of coverage.
What if we examine the same metrics for these three categories? The table below displays the same metrics for the three levels of test coverage in the code delivered in the portfolio in 2014.

<table>
<thead>
<tr>
<th></th>
<th>Number of participants</th>
<th>Total classes</th>
<th>Total LOCs</th>
<th>Total methods</th>
<th>Average classes</th>
<th>LOCs/Methods</th>
<th>Technical debt ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>No coverage</td>
<td>13</td>
<td>46</td>
<td>2185</td>
<td>155</td>
<td>3,5</td>
<td>14,1</td>
<td>7,7%</td>
</tr>
<tr>
<td>Some coverage</td>
<td>9</td>
<td>44</td>
<td>2568</td>
<td>172</td>
<td>4,9</td>
<td>14,9</td>
<td>4,6%</td>
</tr>
<tr>
<td>Good coverage</td>
<td>24</td>
<td>152</td>
<td>8431</td>
<td>736</td>
<td>6,3</td>
<td>11,5</td>
<td>4,4%</td>
</tr>
</tbody>
</table>

Table 2. Code quality metrics categorized by level of test coverage in the portfolio delivery of 2014.

These numbers support the indication that the requirement for automated tests will improve student code. The only deviation is found in LOCs/Methods that slightly increases from “No” to “Some” coverage (14,1 to 14,9). The students managing to write tests write better code than those who do not. Or we could argue that the students who manage to write good code (for example following the SRP and keeping their method sizes small) find it easier to write tests for their code. It is hard to write tests for badly written code.

6 ITERATIVE DEVELOPMENT

The quantitative data from the analysis indicate that the requirement of automated test had a bigger impact on code quality than the portfolio assessment. It would be interesting to highlight an example of student work that exemplifies the positive effect of the iterative element of our version of portfolio assessment.

There were several examples of improvement. One of them was chosen. We call the student “Student A”. Analyzing Student A’s code provides the following result:

<table>
<thead>
<tr>
<th></th>
<th>Total LOCs</th>
<th>Total methods</th>
<th>Classes</th>
<th>LOCs/Methods</th>
<th>Technical debt ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014, review</td>
<td>136</td>
<td>13</td>
<td>3</td>
<td>10,5</td>
<td>4,5%</td>
</tr>
<tr>
<td>2014, portfolio</td>
<td>328</td>
<td>48</td>
<td>7</td>
<td>6,8</td>
<td>3,9%</td>
</tr>
</tbody>
</table>

Table 3. Code quality metrics for one of the students (Student A) in the class of 2014.

Generating a class diagram for the two deliveries provides the following results:
Figure 1. Class diagram for the initial solution written by Student A (prior to receiving feedback from tutor or lecturer).

Figure 2. Class diagram for the final solution written by Student A (after receiving feedback from tutor or lecturer).

Student A initially delivered a fully working program providing the desired functionality. But the two class diagrams tell us that a significant change occurred in the program from the initial delivery to the delivery of the portfolio. The feedback provided at the initial delivery included a suggestion on reducing the responsibility of DBHandler, as parsing a file does not sound like a database related responsibility. Furthermore it was suggested to return a table object in `getTable` instead of a nasty looking list of string arrays in the initial solution. We see that the student also chose to use this table class when parsing the file in the parser class.

This is just one example of code improvement for students using the provided feedback as input for code refinement.
The results from the code analysis indicate that introducing automated testing and portfolio assessment improves the quality of student code. We see that when comparing quantitative data from the code analysis of the class of 2013 and the class of 2014, and we have highlighted an example of improvement based on feedback in the portfolio assessment. The requirement of automated tests seems to have greater impact on the improvement than the portfolio assessment.

One natural uncertainty in these findings is the size of the data material. The class of 2013 consisted of only 14 participants with a total of approximately 2000 LOCs. Could it be that the two classes had a significant difference in student qualifications entering the course? Maybe the class of 2014 had more programming experience? If so, it would impact the results concerning automated testing, as the requirement of automated tests was the main difference between the code of 2013 and the first version delivered in 2014.

The same uncertainty is not applicable to our findings on the impact of portfolio assessment. The effect of portfolio assessment was examined with students of the 2014 class only. As the initial delivery of the code in 2014 was not mandatory, and did only provide formative assessment, we could argue that some students may find it tempting to deliver “work in progress” rather than the best they can do. That would provide them with some useful information on the work ahead, but is not the intended use of portfolio. If this was the case, it would impact our findings on portfolio assessment. The initial delivery did require self-reflection and documentation. However, after reviewing these elements it was clear that delivery of “work in progress” was not a relevant factor in the course.

Portfolio assessment was chosen for a programming course in the following semester as well. As the student grew accustomed to portfolio assessment, delivery of “work in progress” became more frequent. This challenge was met with no feedback if the delivery was not considered complete.

As explained earlier in this paper, the exercises in 2014 counted for a bigger fraction of the grade compared to the final exam than what was the case in 2013. It could be that the students put in more work in their code because of this fact. That could affect the results described earlier. However, this was never observed in class as a subject of interest. And it was not found in the course evaluation performed at the end of the course.

What was found in the course evaluation was a positive attitude towards both automated testing and portfolio assessment. Especially portfolio assessment received a lot of praise (translated to English): “Portfolio assessment is absolutely king. I saw progress in just two months. I thought quite frankly that I had not improved my programming skills, but I noticed it when I refactored assignment 2. It provides awesome learning and insight revisiting “old” code.” And another student: “Portfolio assessment is an indescribably good form of assessment. I feel that I’m actually getting double dividends out of it. By revisiting my old submissions, I freshen up earlier subjects, and at the same time I have the opportunity to apply knowledge that I have acquired after the original submission deadline. It’s exciting to be able to learn by studying the differences between what I did then and what improvements I now can come up with. All subjects (where possible) should use this form of assessment.”

Another point of caution should be mentioned in this discussion of the results. Although one teacher taught both classes, the teacher has gained experience after the first year. It is reasonable to believe that the students will benefit from this, and this could affect the results.

It does not introduce a lot of extra time evaluating code when introducing automated tests. The tests are normally run within a few seconds. In a delivery of code, it can be a requirement that the student themselves provide documentation of the test coverage. There are many easy to use tools to produce test coverage metrics for a solution. The use of SonarQube for analyzing the student code was not an original plan entering the course. An improvement of code quality seemed likely after evaluating the assignments. An analyzing tool was chosen to see if the observed difference actually could be measured.
Introducing portfolio assessment, as we used it in this course, does not scale as easily as introducing automated tests. Delivering quality feedback takes time, and the code has to be evaluated again when the portfolio is delivered. An alternative to using tutors to help out with the delivery of feedback is to use peer-to-peer evaluation. I could also be interesting to introduce instant feedback using a tool like SonarQube as a part of the development process for the students.

8 CONCLUDING REMARKS

The motivation for introducing portfolio assessment was to provide quality feedback on student code. Code written by a student may fulfill the desired functionality, but may be poorly implemented. As a teaching facility educating programmers, it is our duty to not only let our newly educated programmers write code that works, but the code has to be easily maintainable. A part of that is following coding standards and well known coding principles like SRP. We find that some students find it hard to structure their code into classes having a single responsibility and methods with a limited number of lines. The exercise described in this paper gives a nice introduction to JDBC, but can also be used to emphasize code structure and coding principles.

The motivation for introducing automated tests consisted of two parts. First: Our programming students will most likely write tests for their production code when they start working as programmers. The more experience they have writing tests while they are students, the better. Secondly: It is much easier to write good and simple tests if you write nice and simple code. Students struggling with writing tests are often facing problems writing good and simple code. So, introducing automated tests as an assessment criterion will hopefully contribute to better code.

As both automated tests and portfolio assessment improved the quality of our student’s code, we encourage other facilities teaching programming to do the same. If you introduce any of these elements, you will be able to perform the same type of analyze using analyzing tools like SonarQube to measure the change in your student’s code.

References


