UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering

Bolot Kasybekov

Lab Package Development & Evaluation for the Course "Software Engineering"

Master's Thesis (30 ECTS)

Supervisor(s):
Dietmar Pfahl

Tartu 2016

Lab Package Development & Evaluation for the Course "Software Engineering"

Abstract:
In this thesis, a lab package will be delivered for undergraduate students. The aim of the package is to teach students how to analyze and develop software following Test-Driven Development (TDD) process. TDD is one of the most used methodologies nowadays. It can be easily used in the educational context to develop programming skills. The lab package is aimed at bachelors' students who don't have a solid experience in programming. The lab package contains a set of necessary documents and has a certain structure. The documents are usually guidelines, which support the development of particular skills such as requirements' gathering, testing and refactoring. Those skills should be learned in a certain workflow so that students will follow TDD methodology rigorously. Hence, students need to understand all details of TDD. In my thesis, the lab package is divided into two parts. The first part develops analytical skills and the second part develops coding skills. In the first part, students are introduced to the theoretical background of TDD. Then, they see how TDD is used in practice by developing a special small app. During the first part, students learn how to generate requirements, develop domain model, develop examples based on the requirements. Examples are particular test cases for each requirement. There is a prepared " code skeleton" of the game and all examples that the students can build upon. In the second part, students do mainly coding. The main feature is that students follow TDD circle. I want students to understand all specifics of TDD. In the beginning, students will learn how to generate and develop test cases. All test cases are based on examples. Then, they start coding and move on from one test case to another. While coding, they also learn refactoring techniques. The lab package was evaluated by university professors. The results are provided in the form of answers to questionnaire. The main audience are university professors who have an extensive experience in teaching OOP. The results are quite interesting. On the one hand, the structure of the lab package was understandable and clear, the grading scheme was transparent and simple. The professors also agreed that the lab package develops a wide range of skills. Those skills are necessary for TDD. There is some research to be conducted to elaborate how TDD can be applied for educational purposes.

Keywords:
Test-Driven Development, eXtreme programming (XP), Testing
CERCS: P170

Praktikumipaketi arendamine ja hindamine aine „Tarkvaratehnika“ jaoks

Kokkuvõte:
Antud töös koostatakse üliõpilastele mõeldud praktikumide pakett. Paketi eesmärgiks on õpetada üliõpilased analüüsima ja arendama tarkvara järgides test-juhitud arendusprotsessi (TDD). TDD on tänapäeval üks enim kasutatud metoodikaid ja seda saab lihtsalt kasutada hariduslikus kontekstis. See on mõeldud bakalaureusetaseme tudengitele, kellel puudub programmeerimises tugev baas. Praktikumipakett sisaldab praktikumideks vajalikku dokumentide komplekti ja omab kindlat struktuuri. Dokumendid on juhendid, mis võimaldavad arendada konkreetseid oskusi nagu eelduste kogumine, testimine ja refaktoreerimine. Vastavad oskused omandatakse järgides rangelt TDD metoodikat. Seega üliõpilased peavad aru saama kõigist TDD detailidest. Antud töös on praktikumipakett jagatud kaheks osaks. Esimene osa arendab analüütilisi oskusi ja teine osa koodi kirjutamist. Esimeses osas tutvustatakse üliõpilastele TDD teoreetilist tausta ja nad õpivad TDD kasutamist spetsiaalse väikese rakenduse arendamise kaudu. Üliõpilased õpivad eeldusi looma, domeeni mudelit arendama ja eelduste põhjal loodud näidiseid arendama. Mängust ja kõikidest näidistest valmistatakse „koodi skelett“, mille peale saavad õpilased ehitada rakenduse. Teises osas tegelevad üliõpilased peamiselt koodi kirjutamisega ja järgivad TDD ahelat, et mõista kõiki TDD üksikasju. Kõigepealt õpivad üliõpilased looma ja arendama testjuhtumeid, mis kõik põhinevad näidetel. Seejärel alustatakse koodi kirjutamisega ja liigutakse ühelt testjuhtumilt teisele. Samal ajal õpivad nad ka refaktoreerimise tehnikaid. Praktikumipaketti hinnati ülikooli õppejõudude poolt. Vastused küsimustikule on esitatud töös. Peamiselt olid vastajateks õppejõud, kellel on laialdane kogemus OOP õpetamises. Tulemused on küllaltki huvitavad. Praktikumipaketi struktuur tundus mõistetav ja selge. Hindamiskava oli piisavalt lihtne ja õppejõud nõustusid, et praktikumipakett arendab laia valikut oskusi, mis on vajalikud TDD rakendamiseks. Oluline on veel edasi uurida, kuidas saab viimistleda TDD-d hariduslikel eesmärkidel kasutamiseks.
Märksõnad: test-juhitud arendus, eXtreme programming (XP), testimine
CERCS: P170

Table of Contents

1 Introduction

Despite many efforts, computer science students still have misguided views about programming activities

- When compiler processes to the code without complaining, all errors should disappear automatically.
- If the compiler gives the output student anticipates on test value. Student assumes the output will be always correct
- The code that I work on always seems "correct" to me. If the code generates wrong output, there must be something, which is not clear in the code.
- If the code produces the correct output for the sample data, the student might assume that he did everything correctly.

There is widespread belief that typical programming assignments are good practices for forcing the student to behave the above-mentioned way. Students receive feedback only after the code they produce and tend to believe that the code produces the correct result. Instructors don't see how students develop the code. Thus, nobody can be certain if the students have simply cheated or have done something wrong. The cognitive process doesn't play a fundamental role in grading, and students receive feedback only when solved programming task via comments on what and how they learn. Students are often able to succeed at simpler assignments using the above-mentioned methods. Those that are enlisted in bullet points. However, they adapt ineffective strategy which will hinder their performance in more complicated courses.

The above-mentioned approach is called trial and error. It is quite a common strategy for beginners in any discipline. Why do students stick to the same strategy long after it becomes an obstacle? Buck and Stucki describe one possible reason [4, 5]: most undergraduate curricula focus on developing program application and writing code, which is primarily obtained through practical experience. In addition, students must develop basic comprehension and analysis skills. Without them, they are incapable of embracing any strategy beyond trial and error.

Bloom's taxonomy depicts six increasing levels of cognitive development which are used for organizing learning objectives. They are labeled and sorted in increasing the order of complexity: knowledge, comprehension, application, analysis, synthesis, and evaluation. Buck and Stucki[1,2] give their own depiction of Bloom's taxonomy in an IT education. They state students must master basic comprehension and analysis skills as a prerequisite for effective program writing. Students should develop their skills in reading and comprehending source code, predict how a sequence of statements will behave and how a change to the code will result in a change of program behavior. Nevertheless, ordinary undergraduate curricula focus primarily on writing programs: application and synthesis skills.

2 Background information

To change the approach, students need more than just the ability to predict how changes in the code will result in changes of program behavior. Students also need strengthened skills in making hypotheses about the behavior of their code and then experimentally verifying the hypotheses. Students also need frequent, useful and fast feedback about the performance, both in forming hypotheses and in experimentally testing them.

These actions constitute the basis of software testing. To write an effective test, students must also foresee what kind of behavior they expect instead of just coming up with another sequence of code actions. The methodology, which uses testing extensively, is called Test Driven Development (TDD). The goal of this thesis project is to wrap up this methodology into a lab package. Within that package, students will be introduced to the problem, generate tests and code according to the tests they generated.

Why is it important to teach TDD? This methodology has a very complex nature. It doesn't only tests the code but also helps to improve the design aspect of the code. In the list below, some of the key concepts and ideas behind TDD.

- **Test**. The methodology involves designing tests for each unit of the program. A unit, in this context, means the smallest component of the software, which can be tested, such as method or instance variable. TDD needs the automated testing framework because it executes the test for the iterative development cycle. Without, automated testing framework, TDD would be big a burden to practice[3].
- **Analysis.** It refers analysis, design and programming decisions, achieved through refactoring. The analysis is based on two principles. Firstly, software design is incomplete and open to changes. Secondly, the process of writing the test is one the first steps in deciding what the application will do. It is also considered as the form of analysis.

  Based on these principles, tests are written before code is implemented and the test is the form of analysis. It is possible to assert that the process of writing tests drives the design of the system. In other words, TDD is the art of producing automated tests for production code and using that process to drive design and programming. For every small piece of functionality in the production code, test specifies and validates what the production code will do. Then you write enough code to make test pass[3].

- **Development** implies that TDD should be used in the context of other process models as a micro-process, it is not some sort of a software development methodology or process model[3].

  TDD supposes that automated tests aren't rejected once a design decision is made. On the contrary, those tests generated throughout the development cycle become an essential part of the development cycle by giving quick feedback to any subsequent changes made to the system. It helps developers to make changes with confidence as regression testing can be executed immediately after and should any change results in a failure, the tests are still fresh in the developers mind. However, the problem here is that the developer should maintain both code and the set of automated tests generated so far. [3].

Various researchers advocate that TDD offers many benefits to software engineers.

- **Predictability:** Beck[7] suggests that TDD allows engineers to know when they are finished because they have written tests to cover all of the aspects of a feature, and all of those tests pass
- **Learning:** Beck[7] also claims that TDD gives engineers a chance to learn more about their code. He argues "if you only slap together the first thing you think of, then you never have time to think of a second, better thing".
- **Reliability:** Martin [8] argues that one of the greatest advantages TDD is having a suite of regression tests covering all aspects of the system. Engineers can modify the

program and be notified immediately if they have accidentally modified functionality.

- **Speed:** A work by Shore and Warden [9] points out that TDD helps develop code quickly since developers spend very little time debugging and they find mistakes sooner.
- **Confidence:** Astels[10] maintains that one of the TDD's greatest strengths is that no code goes into production without tests associated with it, so an organization whose engineers are using TTD can be confident that all of the code they release behaves as expected
- **Cost:** It is argued by Crispin and House[11] that, because developers are responsible for writing all of the automated tests in the system as a byproduct of TDD, the organization's testers are freed up to do things like perform exploratory testing and help define acceptance criteria, helping save the company developing the software precious resources.
- **Scope Limiting:** TDD helps teams avoid scope creep according to Beck and Andres [12]. Scope creep is the tendency for developers to write extra code or functionality "just in case", even if it isn't required by customers. Because adding the functionality requires writing a test, it forces developers to reconsider whether the functionality is really needed.
- **Documentation:** It is noted by Langr[13] that Test-Driven Development creates programmer documentation automatically. Each unit test case acts as a part of documentation about appropriate usage of a class. Tests can be referred by programmers to understand how a system is supposed to behave, and what responsibilities are

While these advantages are substantial, Beck[21] summarizes the greatest benefit of TDD as "clean code that works". TTD is primarily meant to yield good, clean code. It is not about the quality of the software, it is about the quality of the code.

TDD methodology is convenient to use as a didactical package for various reasons.

- It reinforces incremental development, the application is always in "runtime" and it helps to detect errors as early as code is changed
- The student becomes more confident in the part of the code which he finished and be able to make changes and additions thanks to continuous regression testing.
- The student understands the assignment requirements better because student has to explore the gray areas to be able to completely test his own solution
- The student can always see the growing size of the tests and how much of the required behavior has been done. Thus, student can always check the progress of the development

3 **Research Goal**

Numerous research questions arise. While students analyze test cases, they should keep in mind many questions. What is the right number of tests needed to cover the functionality? What is the granularity of test that should be generated? What are the guidelines needed to write appropriate tests? Those type of research questions can be solved by applying TDD approach. If those questions are solved then the students have mastered the technique, which will improve their coding and testing skills. This, in turn, will increase the quality of the

code they develop. The best way to analyze the effect is to get personal feedback from students.

There are other lab packages related to TDD. However, they don't force students to analyze the functionality of an application. They already provide a ready set of test cases. Students only develop coding skills but the goal of TDD is much wider. In real life, we always have to develop test cases on our own. Nobody will provide them instead of us. The design of code should also be based on the use of a big number of highly related components, which have a weak relationship among each other. This, in turn, facilitates testing and code enhancement. The area of my research includes gathering requirements, converting requirements into test cases and refactoring techniques. Students will master a full cycle of the TDD.

4 Related Work

Because the thesis's objective is quite specific, it was a bit problematic to find similar solutions. There exists didactical software that helps students to generate tests. One example of such software is UnitTestGen.

However, in my opinion, this software has different objectives. As it was mentioned in the Problem Statement, a large number of students thought that using TDD in practice is difficult. This could be attributed to the foreign concept of Test-first, as Melnik[15] noted based on a case study that students believe the Test-first approach is almost like working backward. It is logically confusing. The case study observed that some students felt that writing the test code is more a part of design than testing which supports the hypothesis that writing tests before functionality is difficult as TDD forces design issues forward[15, 16].

According to the author's hypothesis, there should be a special software, which generates all test cases, in the form of a JUnit test classes. At first, a tutorial is distributed to guide students on testing and test case writing. In addition, this tutorial will also provide information on how to use a unit test case generation tool. Then, UnitGen is used in developing a suite of unit test cases, in the form of a JUnit test class, for use with the JUnit testing framework. It works by accepting test parameters from users for methods they wish to test, and generating black-box test cases based on user inputs. The features to be included into UnitTestGen try to eliminate deficiencies observed in the JUnit wizard support provided for Eclipse[17].

The aim of the tutorial is to cover following areas of testing

1. Testing Mindset and Principles
2. Preprocessing steps to generating Comprehensive Unit Test Cases. These are the steps users have to go through before beginning to generate the unit test cases.
3. Steps to generating Comprehensive Unit Test Cases. These are the steps the user has to go through in order to generate a set of comprehensive unit test cases. This is further divided into three sections, guidelines on Equivalence Partitioning, Boundary Value Analysis and considerations that needs to be taken into account when testing object-oriented systems
4. Postprocessing steps for generating Comprehensive Unit Test Cases. These are guidelines describing what the user should consider after the first set of unit test cases has been developed. It highlights the refactoring concept in TDD and what it means to testing, as well as areas in coding that are error prone and the user should pay further attention to.

UnitGen's purpose is to automatically generate JUnit testclass with test methods. Although it is difficult to generate test inputs due to the process being non-algorithmic[18], it is possible to automatically generate test cases that invokes methods for testing, using user-supplied input parameters. The success or failure of test cases is determined by two things. Firstly, if the method gives an output, a comparison is done between the user-supplied expected output against the actual output of the method. Secondly, a comparison is done between the user-supplied expected state of the object against the actual state of the object after method execution. As UnitGen creates the entire JUnit testclass automatically, users don't need to be familiar with how JUnit works in order to their classes using JUnit framework.

To facilitate the process of accepting test values, e.g. method inputs and expected outputs, from the user, a GUI is built. This GUI wizard directs the tester to provide the necessary information required from the user, from which it will generate a JUnit testclass based on the information provided. In order to reduce the amount of information that the user needs to provide in order to generate the JUnit testclass, UnitGen, employs Java reflection to obtain information on the class under test[19].

UnitGen, uses ideas from JNuke, by providing logging facilities, which provide a documentation output. It is a test data file (logfile) that contains information on generated test cases. However, the format of the test data is designed to be convenient for UnitGen to read and is not very readable to humans[19].

Users can use the logfile as a test documentation much like JNuke, after some formatting. Users can also load it back into UnitGen to reuse previously created test cases. It is also possible for experienced users to input test cases directly into the logfile and generate the test methods using UnitGen, rather than going through UnitGen Wizard to create test cases. In an education setting, teachers can predefine object states for students to use, thereby further reducing the time needed to create test cases by students. This might encourage students to be more receptive to the idea of testing and using TDD as the effort required is reduced[19].

However, UnitGen, differs from JNuke in the way it handles the examination of internal object states, defined by the values held in the class fields. Instead of providing strict requirements on string representations of Java classes, UnitGen uses reflection, a feature of Java, to examine internal object states to ensure that object state remains consistent,i.e. class fields only reflect expected changes, after method execution. This saves the tester effort in overriding the toString method of Java classes, in order to conform to the strict requirements of JNuke. Tester also need not resort to "dirty coding", i.e. changing private class fields temporarily to public or protected for the sake of testing. In addition, through the use of reflection, UnitGen allows testers to define certain states of the object to focus on for testing, e.g. testing method execution when a certain array is full or empty[19].

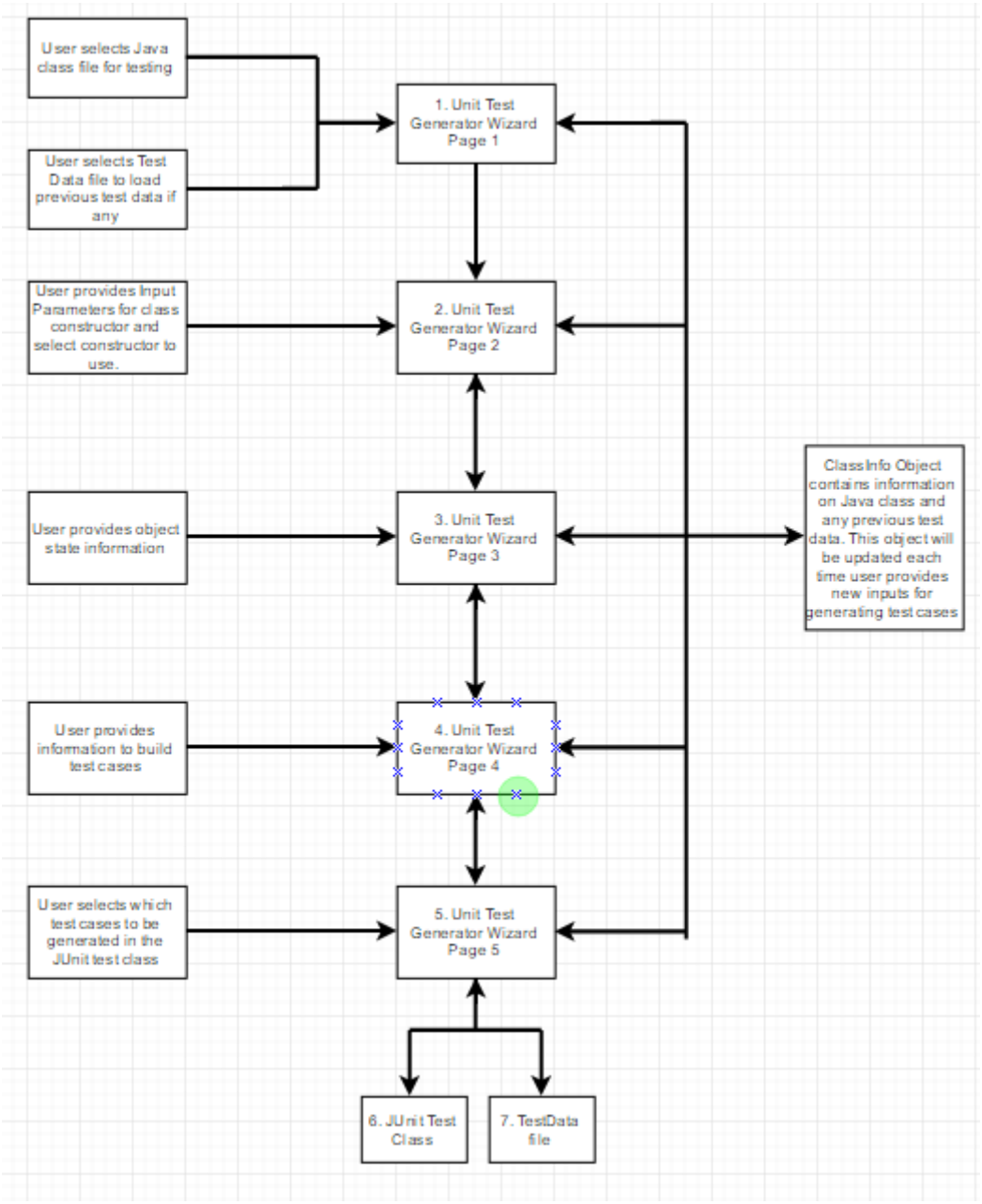


Figure 1. Workflow of UnitGen

This package seems to be very developed. This application is good for automating the test case generation. In other words, test case generation becomes much faster and avoids redundant work of writing test cases. I don't think that this package radically changes the notion of TDD. It just simplifies the testing part of TDD. It contains tutorials and software to generate test cases. In my opinion, it has some drawbacks.

- It doesn't enforce the analysis of the functionality. Before generating test cases, students should thoroughly understand how the system functions. In the beginning, they should understand what the Domain Model of the system is. Then, they should implement the basic functionality of the system etc.
- There is a special tutorial where Equivalence-Class Partitioning and Boundary Value Analysis are used. Those techniques might be complex for students to implement them because the system is quite large.

- The process of generating test cases seems to be complex. Students have to use logfile, which has an unreadable format. There are many steps to be done for generating test methods.
- The process of generating test methods is random. There might be the situation when some part of the functionality isn't covered by test methods.

This lab package omits many important steps which necessary to use within TDD scope. Before writing any tests, it is important to analyze the system, understand what the requirements are etc. The lab package lacks analysis, which plays an important role in TDD. Students might misunderstand the functionality and implement the wrong test case. Hence, I decided to fill those gaps in my didactical lab package.

Another issue is the verification process. The success of test cases is determined by two things. Actual method output is compared against user-supplied method output. User-supplied expected state of the object is compared against the actual state of the object. Those comparisons might be time-consuming while following TDD methodology. Students will have to do redundant work. Unlike UnitGen, my lab package will define all test cases in the beginning and students will only need to implement them. There are also many other unnecessary actions which students have to do. For example, students have to provide object state information, select test data etc. Those actions aren't related to rigorous TDD methodology. Moreover, they distract students from doing necessary work and students don't follow TDD methodology, unlike my lab package.

5 Contribution

UnitGen doesn't provide an answer to following questions "What is the proper number of tests which need to be generated?", "How can we analyze the functionality of the application and reflect the functionality in unit tests?", "In TDD, it is allowed to write a minimum amount of code to pass the test. If student does it to pass a test. How can he refactor the code?" etc. In my opinion, UnitGen doesn't answer them. On the contrary, there are steps, such as logging, which may complicate the process. After analyzing all drawbacks of previous lab package, I decided to create a lab package, which doesn't have all that problems. All gaps will be filled by new lab package. Hence, the ultimate goal is to develop a structure of the lab package, which will be taught to students according to TDD methodology, will not require them to do unnecessary work and develop certain analytical skills. I also decided to emphasize on the practical application such as bowling game. Unlike previous lab package, students will see the value of TDD in practice. Lab package solves the problem in many ways.

- Drives students to think of design issues, e.g. what input parameters are needed and what output is to be expected given certain inputs and specified behavior from requirements specification.
- Allows instant feedback as to whether a method has been implemented as intended by the specifications, this also acts as a form of quality assurance, as the developer can be assured the method implemented is working before moving on.
- Pushes testing to the forefront, making it an integral and unavoidable part of the software development and thus improve testing skills as well

Lab package, in this case, will be a description of the program. It can be described as small games where important features will be described as bullet points. In other words, the main

functionality of the program will be singled out. Why is it important? It will help students to understand the design of the application. They will view a ready "skeleton". After the design of the program is analyzed, students will be able to generate test cases from bullet points. It will be a gradual process. Once test cases are generated, they will be evaluated according to certain criteria.

Within the lab package, students will develop the 'Bowling game'. There are various reasons why this game was chosen.

- The rules of the game are easy to understand and analyze. If some other application was chosen, for example, an application which does a scientific calculation, then it would be a bit entangled for a student because student would need to get familiarized with formula to perform a scientific calculation.
- The game itself is a practical application. Students will see how TDD can be useful in practice by developing the game
- The complexity of the game matches students' knowledge and experience. Students need to know OOP and make a small Domain model.
- Students begin to see the benefits of using TDD after completing few unit tests

6 Solution

The workflow of Lab package follows TDD cycle. Lab package simulates the complete cycle TDD.

In the beginning, students will be delivered a theoretical information about TDD. It includes the workflow of TDD, what are the main steps and advantages why TDD is better than other methodologies. In order to support the latter statement, students will be shown a real-life example of Guitar application. Students will be shown how TDD will help to create a robust and fully functional application. There will be a test case which is not implemented. Then the code will pass the test case.

Next, students will be familiarized with Bowling Game rules. The reason why I decided to choose Bowling is that it is the common domain, which is known to many people. The rules of the game are also not sophisticated and easy to grasp. It would be needless for students to spend an effort by learning the unknown domain. The main point is that student should understand the functionality of the game via rules of the bowling. Once students read the rules, they should develop a domain model of the game. The domain model is the skeleton of the application. The methods will enrich the functionality of the game.

The next stage is requirements generation. The functionality of the system will depend on how test cases are generated. Test cases, in turn, depend on the requirements generated by students. Students will be given guidelines on how to write requirements. There is a special methodology generated by me. Students need to read it, understand it and apply it.

After requirements are completed, the next task is to develop examples. For each requirement, there should be several examples in order to cover the complete functionality of the game. Examples are a concrete representation of the requirement. They include real-life examples.

At the end of the lab, students will have tasks to continue it at home. After they submit the first part of the homework, they will be given a feedback from TA's. In the feedback, TA's will reveal weaknesses in analytical part of the homework. There will be comments saying what is wrong with the concrete requirement or example. However, it is important to mention

that students will be given an ideal set of requirements in order to avoid the further mistakes, which will be reflected in the code. The analytical part of the homework will constitute 30% of the final grade.

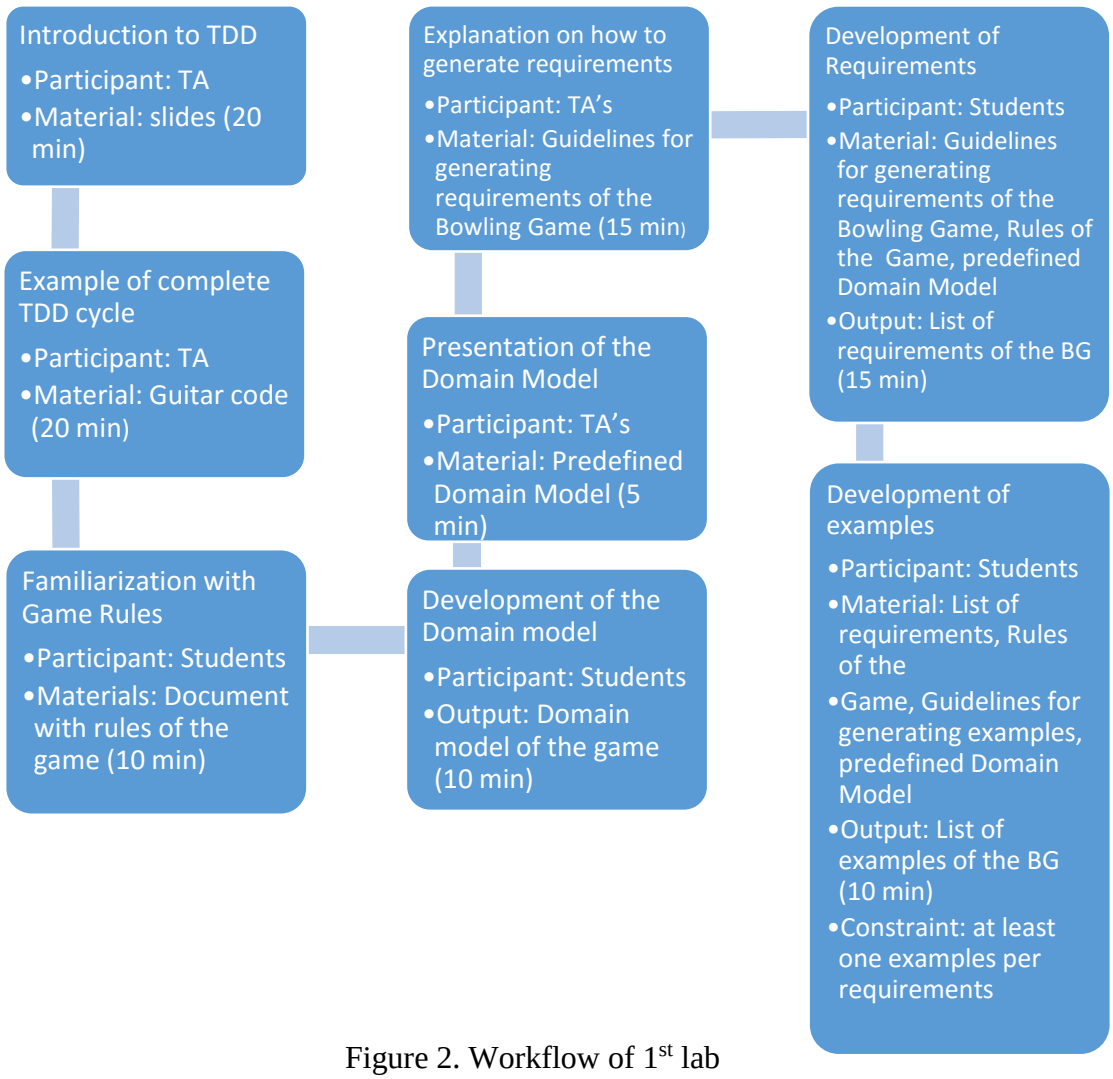


Figure 2. Workflow of 1st lab

The second part of the lab package is related to coding. It gives 70% of the total grade. Students use various coding techniques in order to improve the code reliability, maintainability etc. It will help students to detect errors quickly and enhance the system. It is obvious that it plays a vital role in TDD and, therefore, it constitutes so much percentage of the grade. Unlike the similar packages, it is complex part because it consists of several techniques such as testing, refactoring. Other packages only require to code and pass the test. It also follows the workflow of the TDD so students will see how TDD works in real life.

In the beginning, students will be explained how to generate tests. They will be given a predefined list of requirements, a predefined domain model, and guidelines for test case generation. The purpose at this stage is to transform the requirements into tests. It is an intermediate stage.

Once students are instructed about test case generation, they are given an opportunity to do it on their own. What is important at this stage is that students should generate one-to-many test cases per requirement. Each test case represents a certain aspect of functionality. Hence, the functionality should be fully covered. The students work with the same set of materials.

The next stage is code refactoring. Students are exposed to various refactoring techniques to be able to write a flexible code. They will be given refactoring guidelines. Because there are many test cases present, there can be a situation when students are stuck at one test case. They can write a minimum amount of code in order to barely pass a test. Refactoring may help to solve that issue. It is worth introducing beforehand to avoid further collisions.

Students can code at this stage. They already have a "skeleton" which consists of a ready set of test cases and predefined Domain Model. They need to understand how TDD functions at this stage

After the lab finishes, students will be given a home assignment. They need to complete the whole system. They submit the code and TA's will have time to grade and review it. For TA's, there will be grading criteria.

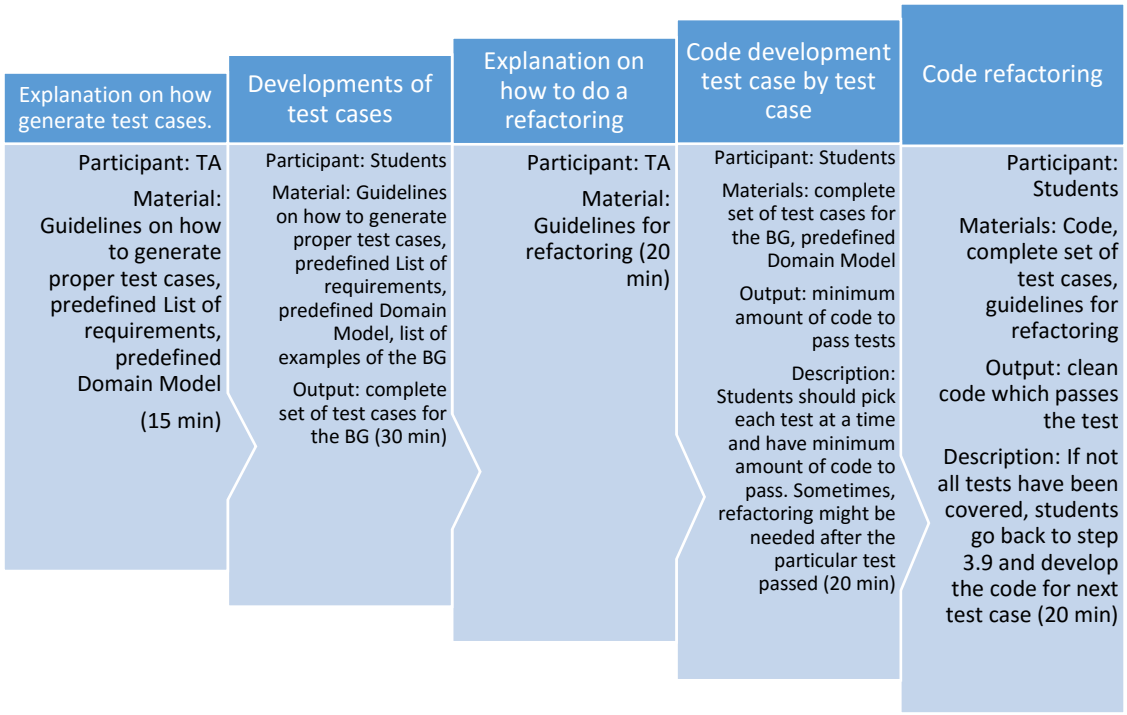


Figure 3. Workflow of 2nd lab

The structure of the lab package is fairly simple. It can be divided into 2 parts. Each part includes set of specific materials. The first part is aimed at students. It includes mainly guidelines about analyzing, refactoring and developing test cases. The second part includes materials for Teaching Assistants. The materials are mainly related to the course organization. They include slides about TDD, Grading guidelines etc.

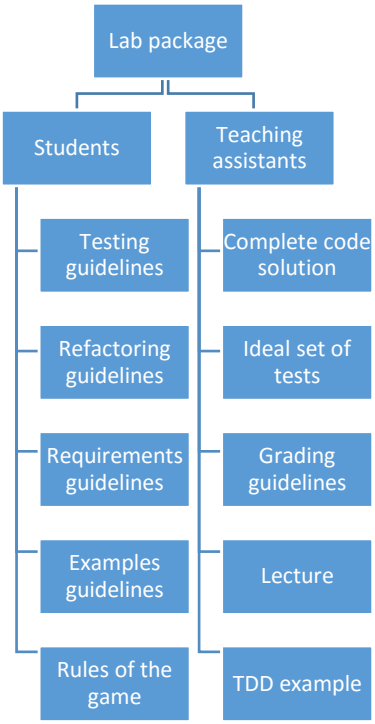


Figure 4. Set of all documents

Each guideline is responsible for a certain type of work which student will do. The idea of those guidelines is to help students to work according to TDD standards.

Testing guideline. Developing an appropriate test is not just a standard, but also it should be understandable to other developers. For example, the method where the test is implemented should follow certain naming convention. It will help the developer to understand which piece of functionality is covered. In the case of the bowling game, it is possible to test an ordinary score of the game. The method name should have the depiction of the functionality, which is tested, and unit test.

```
@Test
public void computeGameScore() {
    
    assertEquals(81, bowlingGame.computeScore());
}
//The goal of this test is to check how the game can compute ordinary score
```

Such approach simplifies the navigation within the code. If some piece of functionality is broken then it is possible to detect by test method names

Another example is that tests should not rely on another test because it will make code hard to maintain. There might be the situation when the player hits 10 pins in one shot. It is called a strike. It can be implemented as the test case.

```
@Test
public void computeStrikeOfFrame() {
    Frame frame1 = new Frame(10, 0);
    assertTrue(frame1.isStrike());
```

However, a strike can be used during a whole game. If there is a need to always verify if a score contains a strike, then the next test case is dependent on strike test case. Thus, the issue with the strike can cause a chain reaction.

By following such guideline, students will develop certain skills necessary for TDD. The guideline will be used at the 2nd stage of the lab package. To see testing guidelines, check Appendix 1

Refactoring guideline. As it was mentioned earlier, the goal of refactoring is the process of changing the code structure without changing the external behavior. The code will become more readable and less complex. It can become more extendable. There are certain specific techniques which refactor the code. Students should use one or more of the techniques once student writes the code, which passes the test, but it is obvious the code won't pass on new one. The goal of this guideline is to develop refactoring skills. The guideline will be distributed at the end of the 1st stage of the lab package.

After tests are generated, The code should be done as well. Each time the code passes the test if it is necessary students should refactor the code. After each test, students should commit the code to the repository. Once code base becomes larger, they should choose a refactoring technique and justify it. It can be done in the form of comments. The main focus is how students learn refactoring techniques. All refactoring techniques are available in Appendix 2

Requirements guideline. The requirement is an intermediate step between test case and a certain piece of functionality of the game. It plays a vital role in analytical part of the lab package. Hence, it is mandatory for students to transform requirements into test code. The guideline reveals a certain strategy about how to generate requirements. It teaches students how to analyze the functionality and generate requirements in a certain way. For example, the bowling game consists of 10 frames. Each frame has two throws. This piece of functionality can be reflected in a requirement. It should be thoroughly described and should have an understandable format. Such format will help other developers to implement the system.

Number of the Requirement. Name of the Requirement

Description of the Requirement

The goal. (In other words, a student should directly state what functionality must be implemented)

The example. (Concrete example should be written by the student. The number of examples is unlimited)

3. Game
A single game consists of 10 frames
Requirement: Define a game, which consists of 10 frames
Example:

Students will follow certain "direction". Requirements are divided into four segments:

- **Basic Functionality.** Once students are done with Domain Model. All initial classes which belong to the game are analyzed. Those classes should be initialized and implemented. The student should also generate requirements, which belong to those structures. If everything is implemented correctly, then the skeleton of the game is finished.
- **Common rules.** Each system has common functionality which ordinary user faces every day. In the case of the game, it is a common set of bowling game rules. It all should be written by a student. In our case, each rule should be written in the form of a test. For example, a spare can be written as a test. The behavior of the class should be adapted to the test. The adaptation may include refactoring. This where students can use refactoring extensively.
- **Combination or Edge cases.** While the system is used on everyday basis, various extreme cases might appear. Those cases should be analyzed and covered by tests. In the case of the bowling game, there might be a situation when there are spare and strike appear at the same time. There might be a situation when the player gets a perfect score.
- **Real-life situations.** Testing the system in real environment ensures that the system will be robust and behave according to the requirements. In the case of the game, the student just needs to simulate the sequence of the frames containing all pins.

It is worth noting that there are variations including Strike and Spare. The strike might appear at the end of the game but it changes the code functionality. Hence, this situation with the Strike at the end should be reflected in the separate requirement. The requirement generation should move iteratively. Once the basic functionality is implemented like Frame, Frame Score; students should move on to more complex requirements. That is where Strike at the end should be written in requirement.

Examples guideline. Examples are specific cases for requirements. Once the requirement is generated, students should develop one-to-many examples to cover a certain aspect of functionality. The guideline gives a hint how to cover a certain aspect of functionality. It also gives good practices about generating examples. Developing the previous point, the example used is below:

Example: The sequence of frames [1, 5] [3, 6] [5, 5] [10, 0] [0, 6] [4, 3] [8, 2] [3, 4] [1, 1] [2, 7] is a game. This game will be reused for various scenarios, where few frames will be modified each time.

It is better to develop various examples to cover functionality. At least, one example should be present. Another good practice is to think as a black-box tester. It is better to read how black-box testing is implemented and follow its rules. All information about examples is provided in Appendix 3.

Rules of the game. While working on generating requirements, examples etc. Students should always refer to the primary source. In our case, the rules of the game are the one we need. The idea why need such type of material is because students need to develop analytical skills. By understanding the functionality of the system, students will be able to generate correct test case etc. All rules of the bowling game is explained in Appendix 4.

Another set of materials will be used by TA's. TA's have two responsibilities: grading and teaching. They will deliver the materials to students and check the progress of students. Hence, all materials are related to those responsibilities.

Lecture. The theoretical background will be provided in the lecture. Students will be exposed to fundamental concepts of the TDD. Students will know how TDD is used. Each step of TDD will be explained thoroughly. There will be also an analysis of TDD and test last approach. The lecture will explain why TDD is better than test last. To see the content of the lecture, check Appendix 5.

TDD example. An example is important because students will see how TDD is applied in practice. A guitar application will be used because the application will convert tabs into notes. It will start from simple tabs to ones that are more complex. Students will witness how test cases will help to implement functionality that is more complicated. The test case will be a certain "progress bar".

Grading guidelines. It is a set of rules needed for grading students' code and requirements. There will be bullet points about grading requirements, example and code. It will indicate which one of them is wrong and which one is right. The schema for grading looks following way

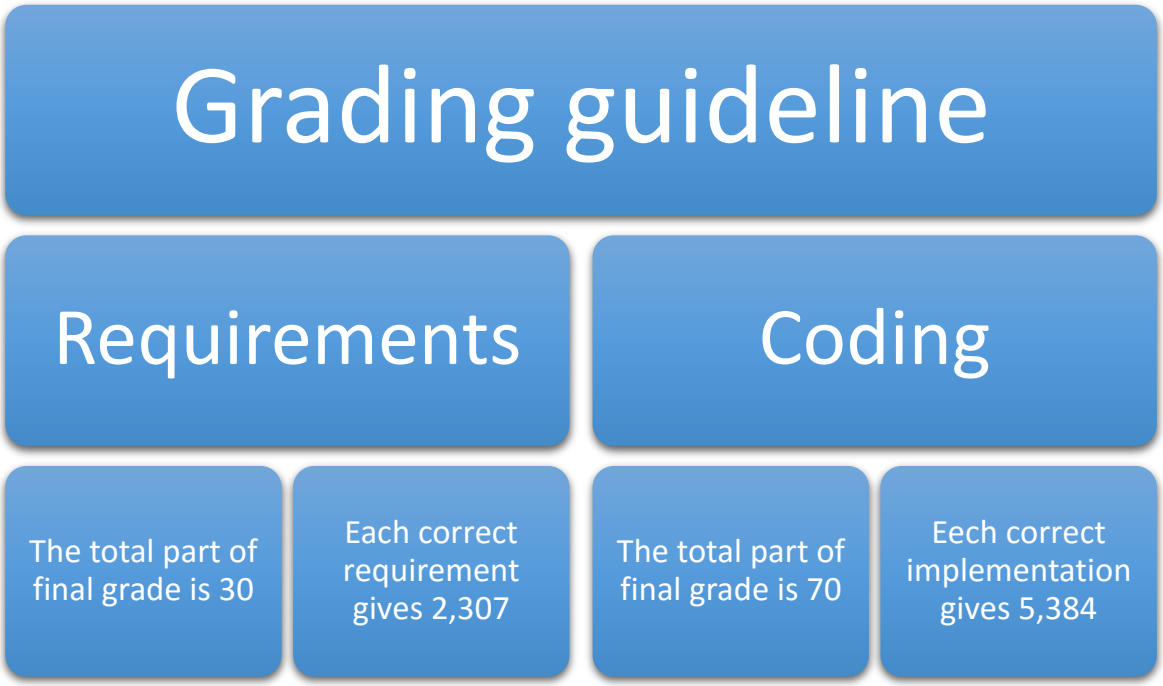


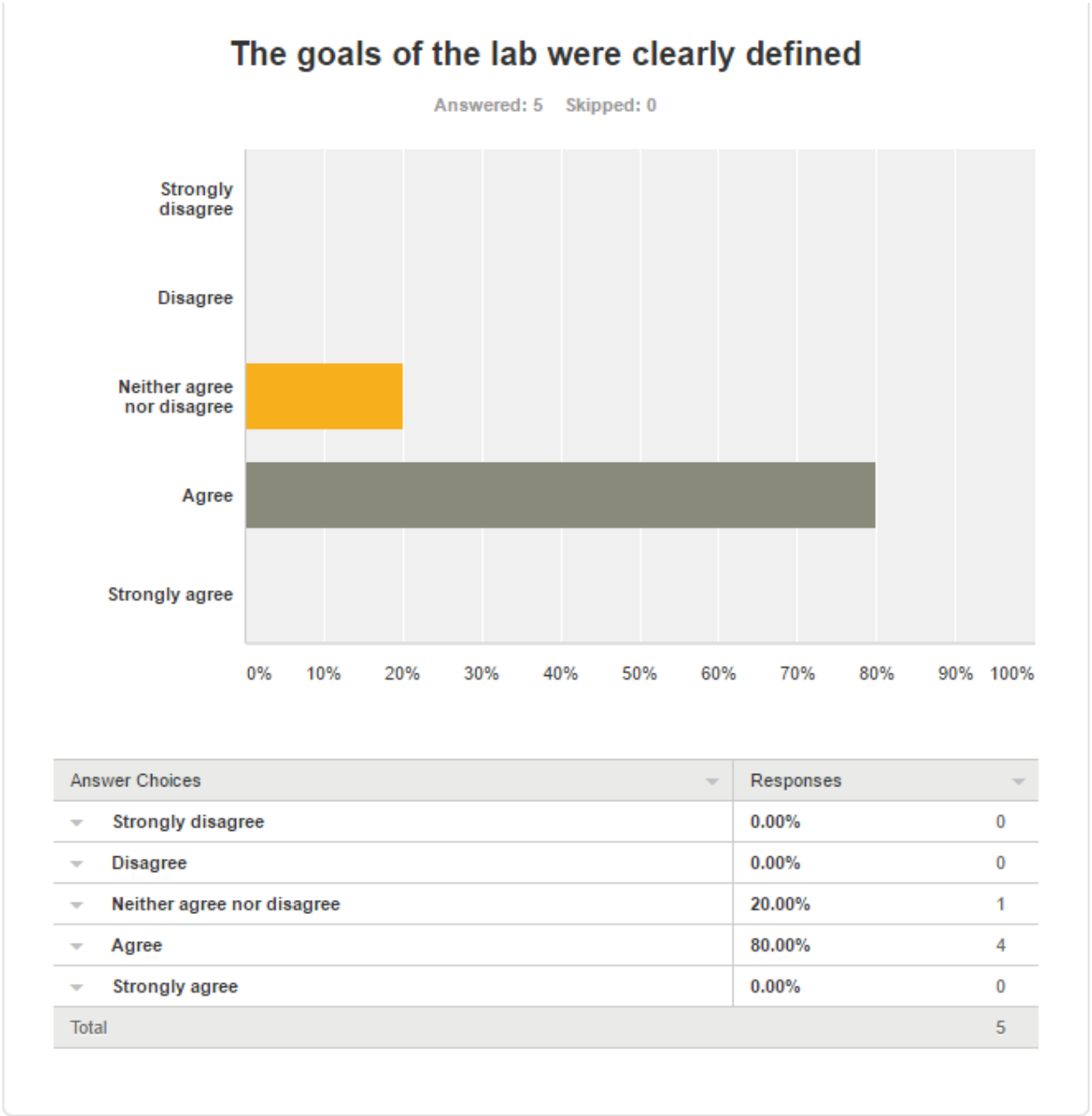
Figure 5. Grading guideline

It is important to separate grading into 2 parts. The first part is called an analytical part. It will constitute 30% of the grade. If the analytical part meets one of the requirements mentioned above, then each point should be subtracted. Because there should be 14 requirements and 14 or more examples. For each requirement and example, the student should get 2,307 points. If a student generates all requirements correctly, then he will 30 points in total. The second part is a coding part. It constitutes the remaining 70% of the grade. Each completed test case will give 5,384 points as well. To see

Ideal set of tests. There is already an implemented solution of the bowling game. TA's will compare that solution with the one students submit. Based on comparison results, TA's will either give a point or skip it. It all depends on how requirements match and test cases have a similar meaning. In case, there are more test cases per requirement than necessary students but they have similar meaning students will be given a full point.

7 Evaluation

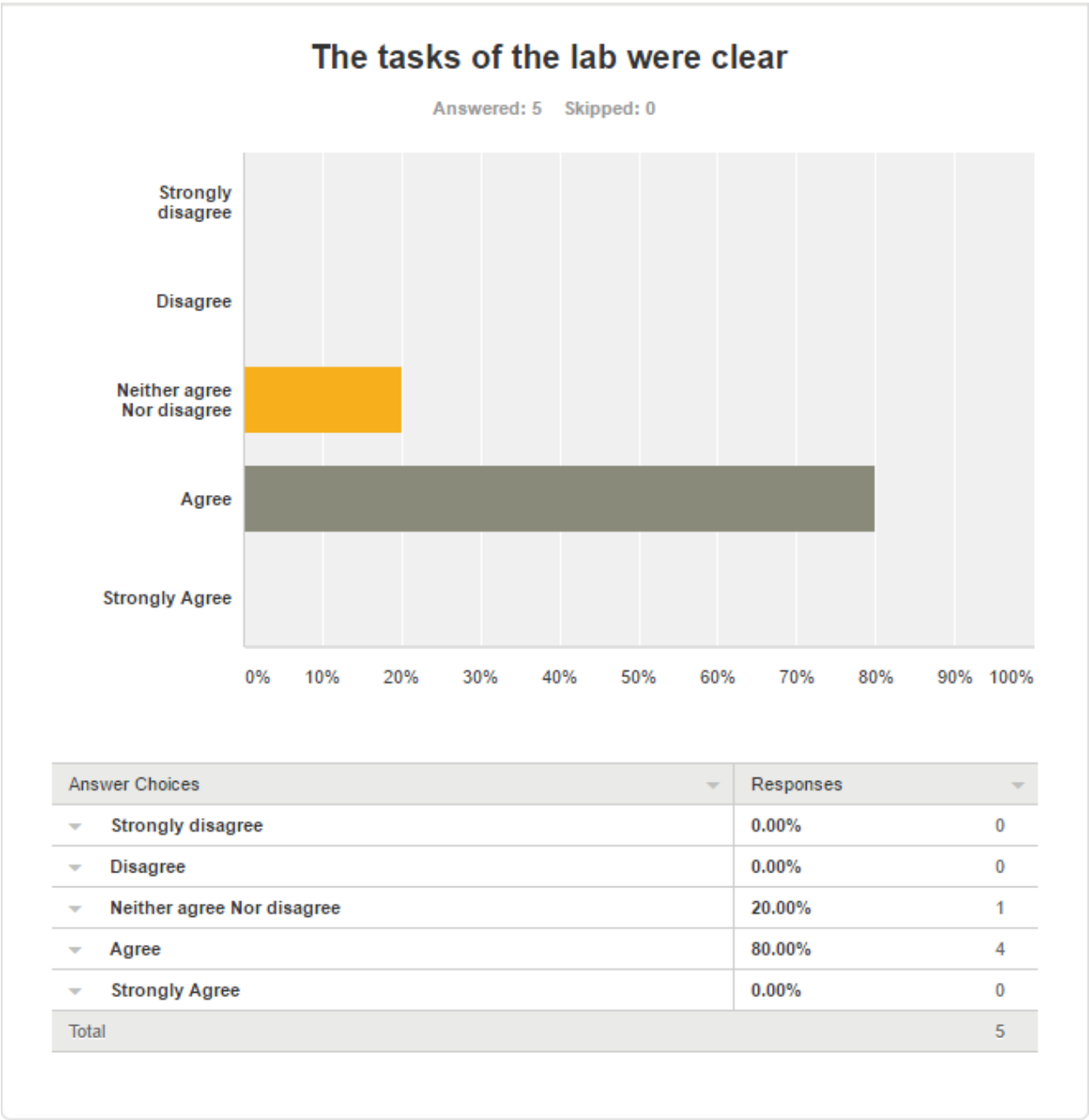
The lab package was evaluated with the help of a questionnaire. There are many ways to measure the quality of software. Jones[23] describes a number of metrics that can be used to measure the quality of software and the productivity of developers. It details how to calculate how much money each line of code costs a business, how to measure "requirements creep" during the requirements phase of software development, and how to measure the effectiveness of integrating outsourced code, among many other things. Pandian[22] describes in detail various ways to measure the quality of software by looking at defects, lines of code, and time. While these metrics are valuable for helping improve the quality of products that engineers create, they don't provide tools for evaluating whole lab package. Lab package evaluation should include a lot factors. For example, it is necessary to evaluate the grading scheme or materials used. The metrics mentioned above do only particular job. It isn't enough to cover whole lab package. The best solution was the questionnaire. In the context of this lab package, the feedback from university professors was used. The professors have the necessary experience in didactics and have the necessary knowledge of TDD. All of them have substantial experience in coding and delivering complex assignments to students. Thus, they can evaluate the game and check if the complexity of the code is suitable for students. I assume they are competent enough to evaluate all other didactic materials such as slides, practical example etc. and the structure of the lab package as well [24]. The feedback will be given in the form of answers to the questionnaire. The answers which will be given along with feedback can reveal how applicable my lab package is and how it can develop necessary skills for students. Their feedback plays an essential role in determining if the lab package can work in real environment. Even though the number of people who met such requirements was limited, I was able to gather valuable information from them. The results were sometimes controversial but interesting as well.



Answer Choices	Responses	
Strongly disagree	0.00%	0
Disagree	0.00%	0
Neither agree nor disagree	20.00%	1
Agree	80.00%	4
Strongly agree	0.00%	0
Total		5

Figure 6. “The goals of the lab were clearly defined”

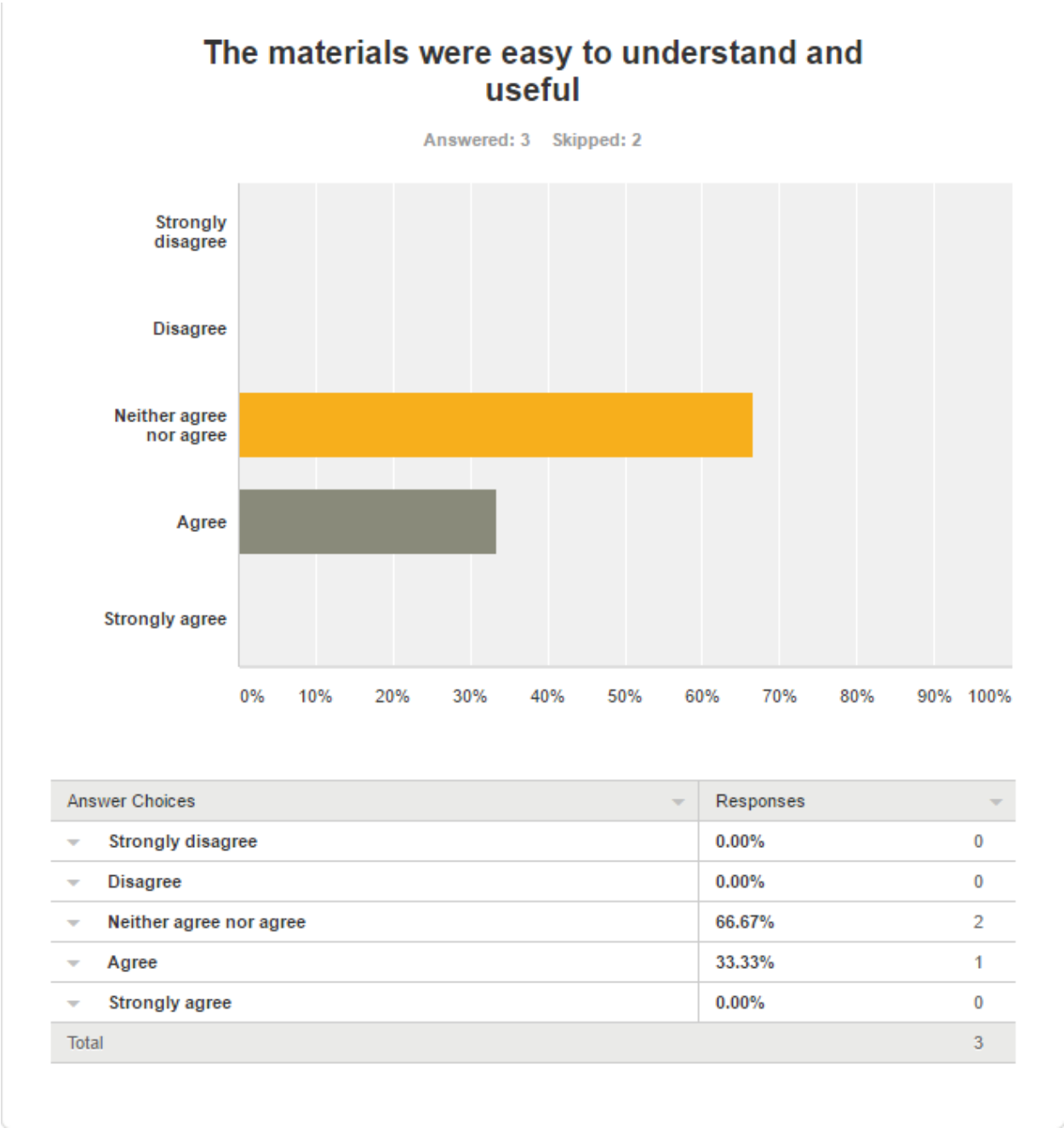
The first question was “*The goals of the lab were clearly defined*”. The professors were given the information about what is expected from students in the lab. In other words, the lab package expects students to learn methodology properly, to do a certain amount of work to develop necessary skills etc. The main goal consisted of many other sub-goals. The main goal itself was to teach students to understand and apply TDD in practice. The professors were informed about all details. Most of the professors agreed with the question.



Answer Choices	Responses	
Strongly disagree	0.00%	0
Disagree	0.00%	0
Neither agree Nor disagree	20.00%	1
Agree	80.00%	4
Strongly Agree	0.00%	0
Total		5

Figure 7. “The tasks of the lab were clear”

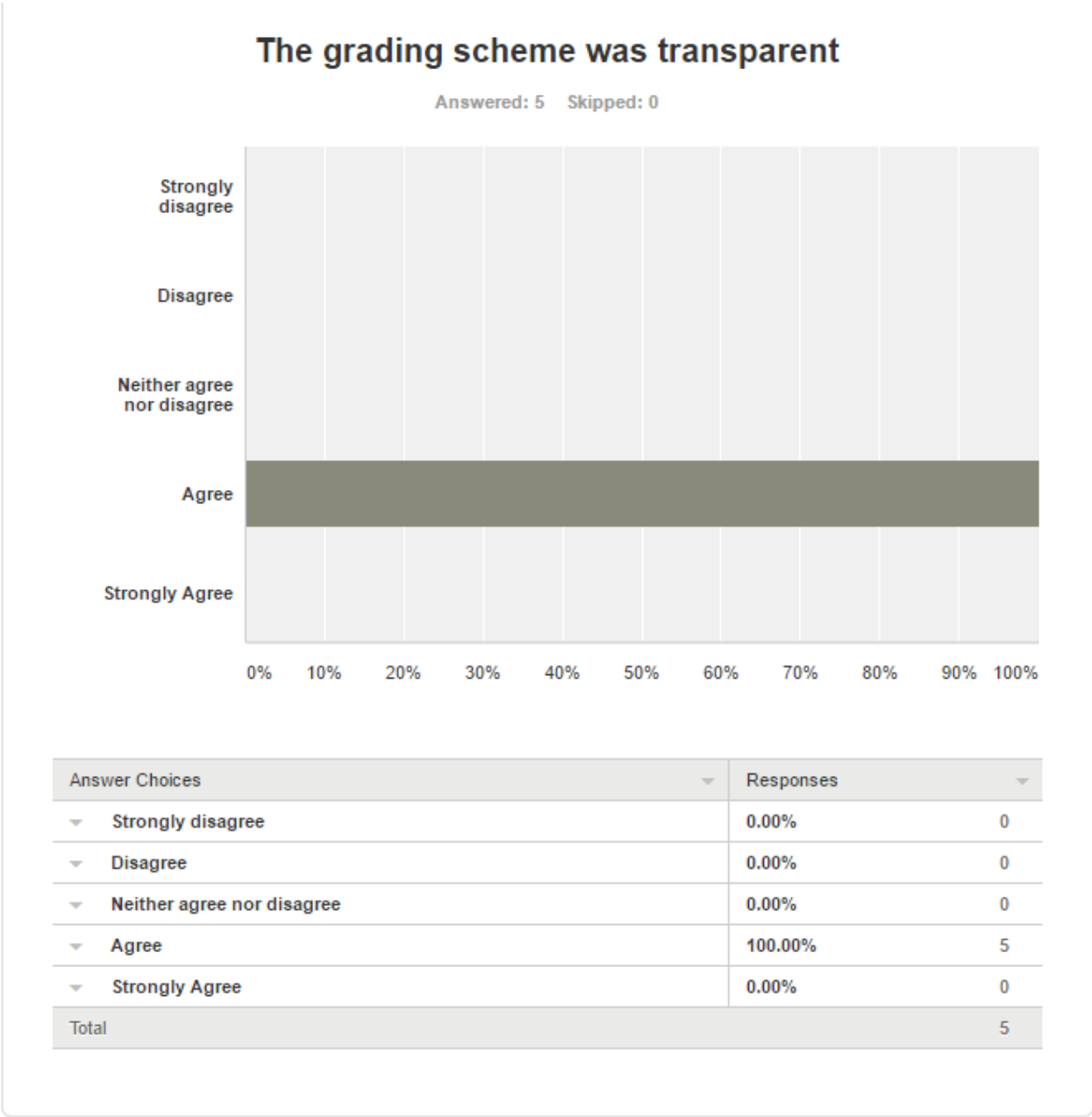
The ratio of professors who agreed with this is also the same. The lab package consisted of many tasks where students have to do a particular job. There was a special diagram which visualized all tasks. Those mini-tasks help to understand TDD step-by-step. Those tasks include actions from figures 2 and 3. Following those steps, students will understand all specifics of TDD methodology. All those tasks were presented and explained to professors. Most of the professors agreed about the content of the tasks and their structure.



Answer Choices	Responses	
Strongly disagree	0.00%	0
Disagree	0.00%	0
Neither agree nor agree	66.67%	2
Agree	33.33%	1
Strongly agree	0.00%	0
Total		3

Figure 8. “The materials were easy to understand and useful”

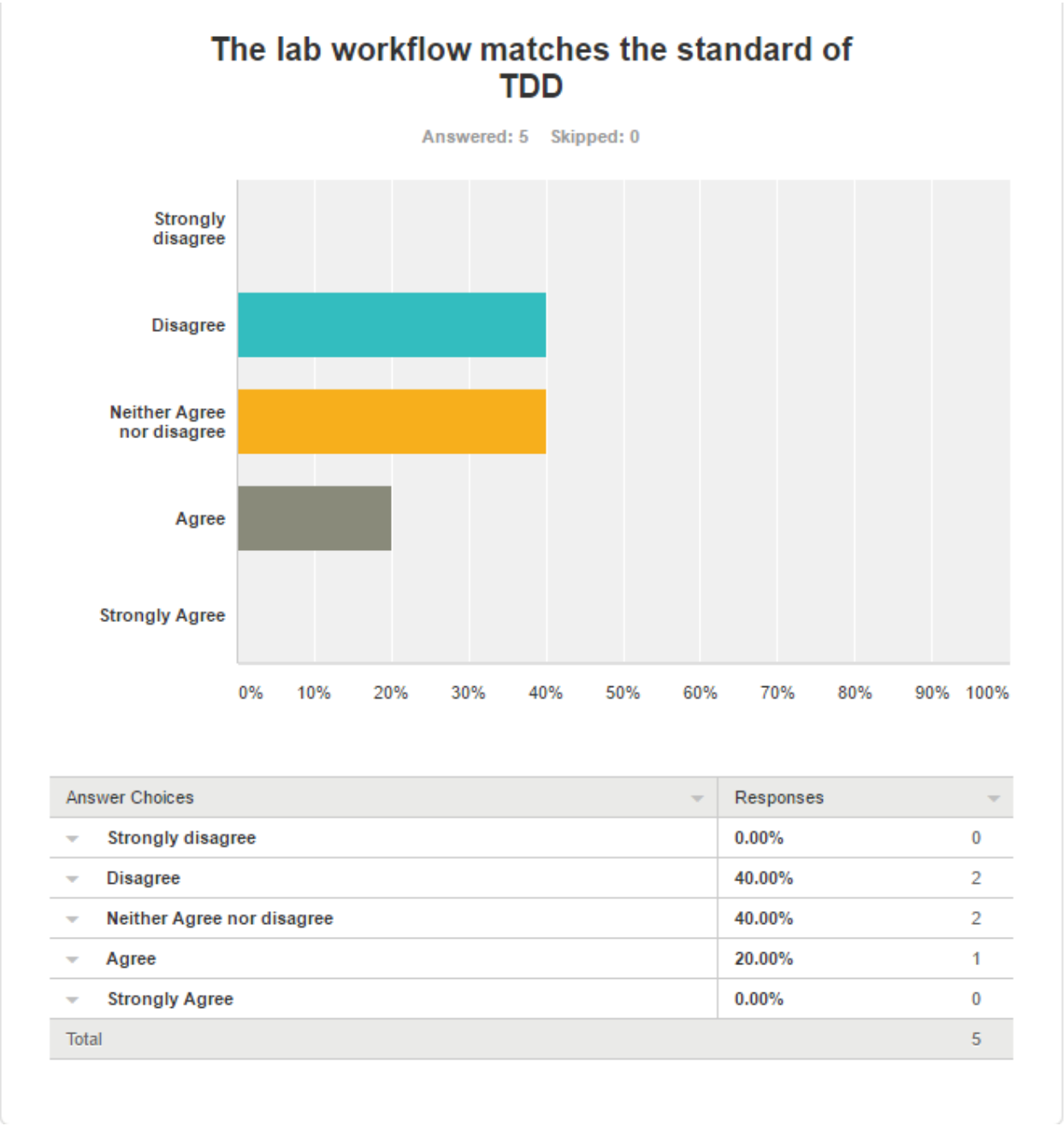
From the picture, you can see that not all professors responded to this question. Most of them had a neutral position. The argument is that the lab package wasn’t tested in the real environment. Students didn’t provide their feedback about the package itself. The problem here is that I couldn’t gather students to test the lab package. The email inviting students was sent throughout the whole department. I guess the problem is that students want any form of rewards such as credit or money. The problem will be solved next semester during the course “Software Engineering”. Students will give certain feedback about particular lab package.



Answer Choices	Responses	
Strongly disagree	0.00%	0
Disagree	0.00%	0
Neither agree nor disagree	0.00%	0
Agree	100.00%	5
Strongly Agree	0.00%	0
Total		5

Figure 9. “The grading scheme was transparent”

The question “The grading scheme was transparent” was fully supported by professors. The grading scheme was visualized in a special graph and divided into two parts. It was quite easy to understand and apply in practice. It was simple and effective at the same time because It counted students’ performance during 2 days. In my opinion, that’s way I gathered positive answer.



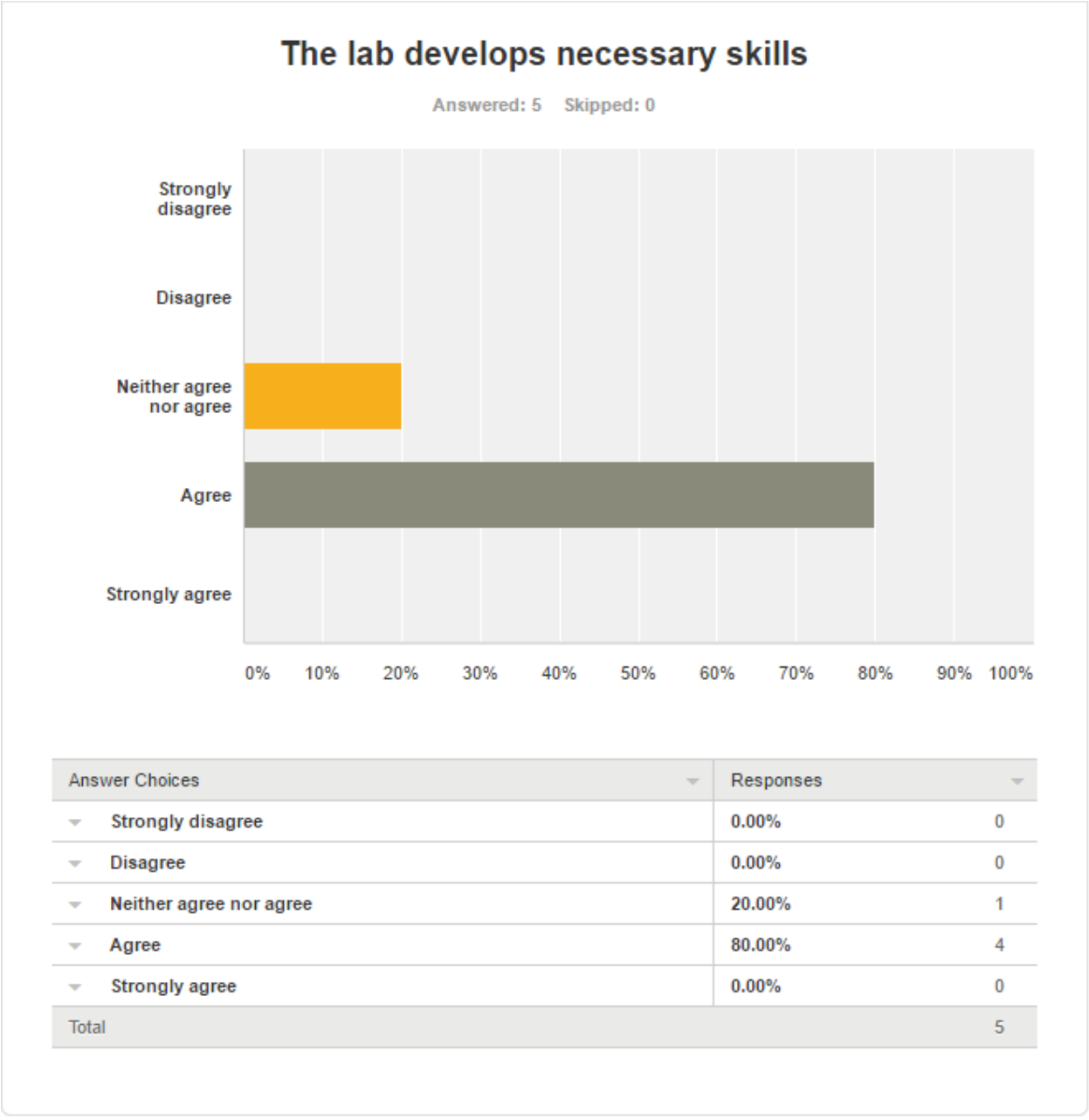
Answer Choices	Responses	
Strongly disagree	0.00%	0
Disagree	40.00%	2
Neither Agree nor disagree	40.00%	2
Agree	20.00%	1
Strongly Agree	0.00%	0
Total		5

Figure 10. “The lab workflow matches the standard of TDD”

The question “The lab workflow matches the standard of TDD” is controversial. Some professors agreed, other disagreed. The opposite point of view states each new feature begins with writing a test. The test defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature’s specification and requirements. Once the test is written, the developer should run all tests and check if any test fails. The developer should write a minimum amount of code to pass the test. He can even hardcode make the test pass. The developer should run the tests again. Those tests which seem to work in an inelegant way should be refactored. The code base should be clean up regularly during TDD. The new code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. The emphasis is on delivering the

code first rather than documenting the analysis[19]. According to the respondent's point of view, this is classical definition of TDD.

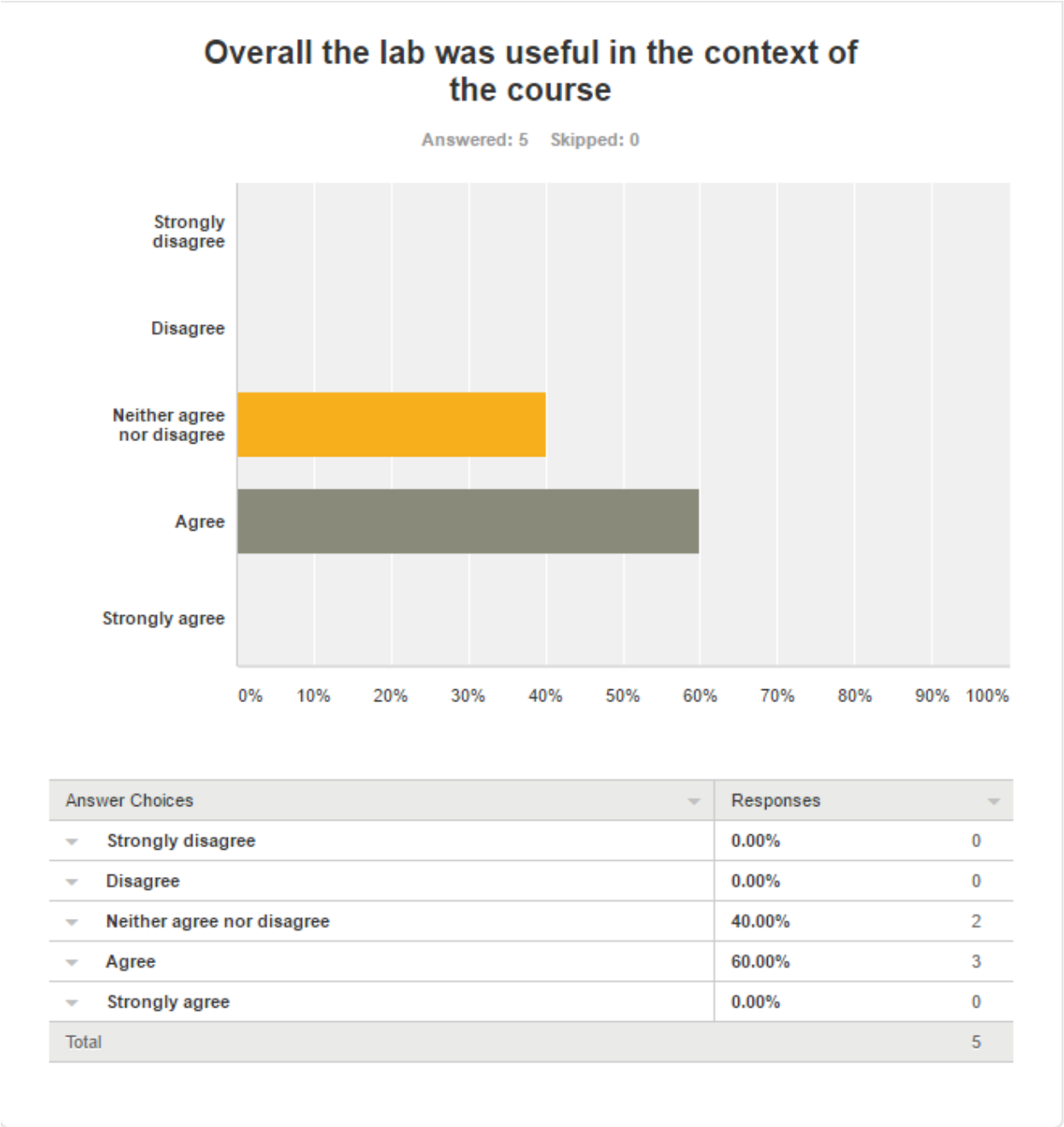
I agree with this position. However, the lab package is also intended to enrich the skills of students by forcing to analyze the requirements. Thus, students will be able to solve a wider range of tasks. They will also develop a wider range of skills. For example, in the context of the lab package they will work as analysts and developers. I believe that they will be better prepared for the industry challenges. Their value as IT specialists will be much higher. This is important because there are plenty of tutorials available on the internet but their purpose is limited coding through tests. Tutorials blindly force students to follow TDD. Those tutorial don't even force students to learn various refactoring techniques, they don't develop analytical skills etc. The lab package solves those issues and also meets the purposes of the course. The main argument is push boundaries of TDD methodology.



Answer Choices	Responses	
Strongly disagree	0.00%	0
Disagree	0.00%	0
Neither agree nor agree	20.00%	1
Agree	80.00%	4
Strongly agree	0.00%	0
Total		5

Figure 11. "The lab develops necessary skills"

The question “the lab develops necessary skills” was mainly supported by professors. There are many skills used in the lab package such as testing, analyzing, and refactoring. In order to learn and use them properly, I used special guidelines. During a certain phase of the lab package, students will use those guidelines. The guidelines are simple and clear. It is fairly easy for students to read and understand them. Hence, professors highly evaluated them.



Answer Choices	Responses	
Strongly disagree	0.00%	0
Disagree	0.00%	0
Neither agree nor disagree	40.00%	2
Agree	60.00%	3
Strongly agree	0.00%	0
Total		5

Figure 12. “Overall the lab was useful in the context of the course”

The answer to a question “Overall, the lab was useful in the context of the course” was a bit uncertain. The main argument of the opposing side is the lab package wasn’t tested in the real environment. There was no feedback from teaching assistants. Such question can be only answered in real practice. The only way solve is to use the lab package next semester, develop a special questionnaire and get feedback.

Other’s side argument is that my lab package more or less attempts to follow the standards of TDD. Hence, students will know how to apply TDD in the industry and will see the benefit

of using it. They also note that my lab package develops some adjacent skills such as analyzing the requirements.

8 Conclusion

In conclusion, I can state that the feedback was positive but there were two important remarks. The first remark is that the lab package wasn't tested in practice. As I mentioned before, it was difficult to gather a certain number of people because they weren't promised any incentives. However, the lab package could be used next semester and then it would be possible to get practical results. Another remark was related to requirements gathering. The main remark is that standard of TDD are perceived differently by professors. My argument here is that I want to adapt lab package in the educational setting. In point of view, I don't necessarily contradict rather I attempt to enrich standard for the educational purposes. The analysis is important because students will have a certain "blueprint" for their code so they won't spend extra efforts by redesigning the tests. They will also develop additional skills. I believe that this lab package will help students to be better prepared for the industry and solve a wide range of tasks such as analyzing, testing, and refactoring.

9 References

[1] Buck, D., and Stucki, D.J. “Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development”, pp. 75-79, 2000.

[2] Buck, D., and Stucki, D.J., J Karel, “Robot: a case study in supporting levels of cognitive development in the computer science curriculum”, pp. 16-20, 2001

[3] Janzen, D. S. and, Saiedian, H., “Test-Driven Development: Concepts, Taxonomy, and Future Direction”, pp. 43-40, 2005

[4] Edwards, S. H., “Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action”, vol. 36, no. 1, pp. 26-30, 2004

[5] George, B. and, Williams, L., ’An Initial Investigation of Test Driven Development in Industry’, pp. 1135-1139, 2003

[6] Melnik, G. and, Maurer, F., “Introducing Agile Methods: Three Years of Experience”, pp. 334-341, 2004

[7] Beck, K., “Test-driven development: By Example”, pp. 7-8, 2002

[8] Martin, R. C., “Agile software development, principles, patterns and practices”, pp. 28-30, 2002

[9] Shore, J. and, Warden J., “The art of agile development”, pp. 33, 2007

[10] Astels, D., “Test-driven development: A practical guide”, pp. 7, 2003

[11] Crispin, L. and, House, T., “Testing extreme programming”, pp. 7, 2002

[12] Beck, K. and, Andres, C., “Extreme programming explained”, pp. 5-8, 2004

[13] Langr, J., “Agile Java: Crafting code with test-driven development”, pp. 8, 2005

[14] Beck, K., “Test-driven development: By example”, pp. 7-8, 2002

[15] Melnik, G., Maurer, F., “Introducing Agile Methods: Three Years of Experience”, pp. 334-341, 2004

[16] Janzen, D. S. and, Saiedian, H., “Test-Driven Development: Concepts, Taxonomy, and Future Direction”, pp. 40-43, 2005

[17] Jimm, Boh S., “Tool Support for Test Generation in Test-Driven Development”, pp. 6, 2007

[18] Olan, M., “Unit Testing: Test Early, Test Often”, pp. 319-328, 2003

[19] Jimm, Boh S., “Tool Support for Test Generation in Test-Driven Development”, pp. 36, 2007

[20] Beck, K., “Test-Driven Development by Example”, pp. 195-210, 2003

[21] Beck, K., “Test-Driven Development by Example”, pp. 7-8, 2003

[22] Pandian, C.R., “Software Metrics: A guide to planning, analysis and application”, pp. 19, 2003

[23] Jones, C., “Applied software measurement”, pp. 19, 2008

[24] Kasybekov, B., “Materials for TDD Lab package”, https://www.dropbox.com/home/thesis%20materials, 2016

Appendix 1

Guidelines for tests

- **Measure the tests.** Use tools which check the coverage analysis so that it is possible to see all how much of the code was covered and investigate which parts of the code is executed and not.
- **Prioritize testing.** Unit testing can be considered a bottom-up process, and if there are not enough resources to test all parts of a system priority should be put on the lower levels first.
- **Keep tests independently.** It is important to make test not to rely on other test and not to depend on the order in which tests are executed. It will make test suite robust and simplify maintenance.
- **Write tests to reproduce bugs.** When a bug is reported, write a test to reproduce the bug (i.e. a failing test) and use this test as success criteria when fixing the code.
- **You should fully automate unit tests and make them non-interactive.** The test suite is executed regularly and must be fully automated to be useful. If you manually examine the tests then they aren't right unit tests
- **Make unit tests simple to run.** It would be great to configure development environment in the way that all tests can be run by a single command or by one button click.
- **Fix failing tests immediately.** Each developer is responsible for a portion of code he is working on. He should be confident that all tests which he has written can run successfully upon code check in and every new test will run successfully. If a test fails, the entire team should focus on that problem, drop their work and fix the problem.
- **Name tests properly.** It is useful to cover one distinct feature of the class with test method and give the proper name to it. The example for naming convention are testSaveAs(), testAddListener(), testDeleteProperty() etc.

Tests should be generated according to requirements. Hence, the number of tests should match the number of requirements. The bigger number of tests, the more thorough the functionality is covered.
However, the redundant number of tests may slow down the development process.
Guidelines for good tests are:

- Long initialization code. For one assert() statement, there shouldn't be long lines of code. If it is so, then the objects are too big and need to be separated
- Tests should execute quickly. If tests work slowly, then some components have serious issues. Those issues indicate that there is a serious deficiency in the design. In other words, if we improve the design then we will improve the speed of the tests.
- Fragile tests. If your tests break in unpredictable situations, it means that the part of your system influences another one. In this case, it is important to improve the design in such way that this effect would be eliminated

Appendix 2

Refactoring guidelines:

- **Isolate changes.** How is it possible to modify one part of the method or object, which consists of a several parts? At first, you should change variable part. You might notice that after you isolated change and made a change to the code the result became so trivial so you can cancel an isolation. For example, if you noticed that there is one action within findRate() method – the return of the field value. We can directly access the field instead of accessing the method. As the result, findRate() method can be removed. However, such changes can not be implemented automatically. Try to find a balance between related to the cost of usage of additional method and benefit which is brought by a new concept.
- **Extract method.** How is it possible to make a long and complicated code easy to read? Extract a tiny part of long method into separate one and access that part of the long method
    - Outline the fragment of the code, which can be put into a separate method. Good candidates are the bodies of loops, loops and the branches of conditional operators.
    - Make sure that inside the fragment there is no assignment of values to the temporary values, which are declared outside the scope of visibility that match to that fragment
    - Copy the code from old method to the new one. Compile it.
    - For each temporary variable or parameter of initial method used in new method add the parameter to the new method
    - Make sure that at necessary place the old method accessed the new one

This method is used to understand a complicated segment of code because you help your partner and understand what really happens in that complicated segment of code. It is also used to get rid of code duplication when two methods have similar pieces of code. In this case, such segment should be moved into a separate method.

- **Inline method.** How can you simplify a code in the case when it becomes hard to observe the sequence of transfer control from method to method? Replace the access to the method with the code of that method
    - Copy the code the method to the clipboard
    - Insert the code of the method instead of access to the method
    - Replace all formal parameters with real parameters. If, for example, you transfer reader.getNext() which is the expression that has a side effect, be careful and assign the received value to the temporary variable.

There is one example where object Bank will convert object Expression to an object Money

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10). reduced);
}
```

However, it looks very complicated. Why can't Money do a conversion? Let's insert sum.reduce() and look at it.

```
94
95  public void testSimpleAddition() {
96      Money five= Money.dollar(5);
97      Expression sum= five.plus(five);
98      Bank bank= new Bank();
99      Money reduced= sum.reduce(bank, "USD");
100     assertEquals(Money.dollar(10), reduced);
101 }
102
```

It is important to understand that inline method helps to experiment with the sequence of the action execution. When student implements refactoring, the student should form a picture of the system with logic pieces and execution flow, which moves from one object to another one. This how the student can avoid a mess in the logic.

- **Move method.** How can you relocate the method to a new place where it should belong. Add it to the class where it should belong and then access it
    - Copy the method into clipboard
    - Insert method into the target class. Assign it a necessary name. Compile it
    - If within the method there is an access to the initial object. Add the parameter which will pass the object inside the method. If within the method there is an access to member variables of an initial object, pass them as the parameters. If inside the method member variables are assigned values, you should refuse from the idea of transferring to new object
    - Replace the body of the initial method with the access to new method

It can be considered one of the most effective refactoring techniques. It effectively shows wrong assumptions about code design. Let's take, for example, an object Shape which calculates Area

```
int width = bounds.right() - bounds.left();
int height = bounds.bottom() - bounds.top();
int area = width * height;
```

Every time inside a method, which belongs to one object; there is an access to several methods of another object, the student should be suspicious. In this case, a method, which belongs to object Shape accesses to four methods of bounds object. It should be moved to a Rectangle class:

```
public int area() {
    int width = this.right() - this.left();
    int height = this.bottom() - this.top();
    return width * height;
}
```

This technique has three important advantages

- If the student can't understand deeply the meaning of the code, it can still be easily applied. If the student notices two or more messages addressed to another object then he can easily apply it.
- Execution mechanics is quite and safe.
- As the result, student can understand the code better

- **Method Object.** How can you implement a complex method which uses several parameters and local variables? Convert method into a separate object
  - Create class with the same number of parameters as the original method
  - Convert local variable into instance variables of new class
  - Define new method run() inside new class. The body of that method will be same as the body of the original method.
  - In original method create new object and access to the method run() of that object

Method object is useful as a preparatory stage before adding new type of logic

Appendix 3

The complete list of examples is shown below

- **Keep examples at a unit level.** There should be one-to-many examples related to the requirement. Each example should be attached to the specific behavior of the class. Avoid the temptation to test an entire work-flow. For example, there is a requirement related to a game score. The student needs to fill in frame with a various score. In this case, a student can just have one-to-many scores.

  Example: The score of the game [1, 5] [3, 6] [7, 2] [3, 6] [4, 4] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] is 81.

  Example: The score of the game [2, 7] [3, 6] [7, 2] [3, 6] [4, 5] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] is 85.

However, it should not contradict further requirements

- **Test the trivial cases too.** Usually, it is advised to skip all trivial methods like getters and setters and test non-trivial test cases. However, there are several reasons why you should test trivial cases
  - Trivial is difficult to determine. Different people have a different understanding.
  - From a black-box perspective, there is no part of code, which can be considered trivial.
  - The trivial cases also contain errors, frequently as the result of copy-paste operations. The advice to test everything. The trivial cases are quite simple to test.
- **Test each feature once.** There is no need to come up with repetitive examples because it delays the work time. For example, there is no need to develop absolutely identical test case because it is a redundant work.

Example: The score of the game [2, 7] [3, 6] [7, 2] [3, 6] [4, 5] [5, 3] [3, 3] [4, 5] [8, 1] [2, 6] is 85.

- **Be aware of the limitations.** Unit tests never prove the correctness of code. A failing test only reveals that the code contains errors in the structure, but even if the test succeeds it doesn't prove anything at all. Unit tests are dependent on proper up-front design. They verify and document the requirements at a low level and verify that code invariants are stable during code evolution and refactoring. They can be considered a valuable supplement to the established development methodologies.
- **Think in terms of black-box testing.** You should view the code as separate third party class consumer, and test if the class meets the requirements. It would be quite beneficial to use two famous black-box techniques such as Equivalence class portioning or Boundary value Analysis. If there is a limited set of input variables, both techniques are applicable. Let's take, for example, a module which calculates the square root. The specification describes for the tester conditions relevant to the input/output variables x and y. The input conditions are that the variable x must be a real number and be equal to or greater than 0.0. The conditions for the output variable y are that it must be a real number equal to or greater than 0.0, whose square is approximately equal to x. If x is not equal to or greater than 0.0, then an exception is raised. From this information, the tester can easily generate both invalid and valid equivalence classes and boundaries. For example, input equivalence classes for this module are the following:

EC1. The input variable x is real, valid.

EC2. The input variable x is not real, invalid.
EC3. The value of x is greater than 0.0, valid.
EC4. The value of x is less than 0.0, invalid.

After the equivalence classes have been identified in this way, the next step in test case design is the development of the actual test cases. A good approach includes the following steps.

1. Each equivalence class should be assigned a unique identifier. A simple integer is sufficient.
2. Develop test cases for all valid equivalence classes until all have been covered by (included in) a test case. A given test case may cover more than one equivalence class.

The test cases based on equivalence class partitioning can be improved by use of another technique called boundary value analysis. With experience, testers soon realize that many defects occur directly on, and above and below, the edges of equivalence classes.

- **Provide a random generator.** When the boundary cases are covered, one of the ordinary ways to get better test coverage is to generate random parameters so that the tests get different input every time they are executed. Create simple utility class that generates random values of the basic variables like integers, doubles, strings, dates etc. If the tests are fast, it would be good to run them inside loops to cover all possible input combinations. The example verifies that converting between one end and another end gives back the original value. Because the test is fast, it is executed on one million different values each time.

```
void testByteSwapper()
  {
   for (int i = 0; i < 1000000; i++) {
    double v0 = Random.getDouble();
    double v1 = ByteSwapper.swap(v0);
    double v2 = ByteSwapper.swap(v1);
    assertEquals(v0, v2);
   }
  }
```

- **Know the cost of testing.** Not writing unit tests is dangerous, but writing them is also difficult. There is a trade-off between them. In terms of execution coverage, the typical standard is at about 80%. The areas where it is hard to get test coverage is on error and exception handling which deals with external resources. Simulating a database breakdown in the middle of a transaction is allowed but it might take a lot of time comparing to extensive code reviews which are the alternative approach.

Appendix 4

A game of bowling consists of ten frames. In each frame, the bowler will have chances to knock down as many pins as possible with their bowling ball. If a bowler is able to knock down all ten pins with his first ball, he is awarded a strike. If the bowler is able to knock down all 10 pins with the two balls of a frame, it is known as a spare. Bonus points are awarded for both of these, depending on what is scored in the next 2 balls (for a strike) or 1 ball (for a spare). If the bowler knocks down all 10 pins in the tenth frame, the bowler is allowed to throw 3 balls for that frame. This allows for a potential of 12 strikes in a single game, and a maximum score of 300 points, a perfect game.

In general, one point is scored for each pin that is knocked over. Therefore, if a player bowls over three pins with the first shot, then six with the second, the player would receive a total of nine points for that frame. If a player knocks down 9 pins with the first shot but misses with the second, the player would also score nine. When a player fails to knock down all ten pins after their second ball it is known as an open frame. In the event that all ten pins are knocked over by a player in a single frame, bonuses are awarded.

When all ten pins are knocked down with the first ball, a player is awarded ten points, plus a bonus of whatever is scored with the next two balls. In this way, the points scored for the two balls after the strike are counted twice. The most points that can be scored in a single frame are 30 points (10 for the original strike, plus strikes in the two subsequent frames). A player who bowls a strike in the tenth (final) frame is awarded two extra balls so as to allow the awarding of bonus points. If both these balls also result in strikes, a total of 30 points (10 + 10 + 10) is awarded for the frame. These bonus points do not count on their own; they only count as the bonus for the strike.

A ten-pin bowling score sheet showing how a spare is scored:

A “spare” is awarded when no pins are left standing after the second ball of a frame; i.e., a player uses both balls of a frame to clear all ten pins. A player achieving a spare is awarded ten points, plus a bonus of whatever is scored with the next ball (only the first ball is counted). It is typically rendered as a slash on score sheets in place of the second pin count for a frame.

A player who bowls a spare in the tenth (final) frame is awarded one extra ball to allow for the bonus points. The maximum score in a game of ten-pin is 300.

After the strike, there can be a spare. The strike and spare scores can be combined. At first, the strike’s score is combined with spare’s score. After that, spare’s score is combined with ordinary frame score. The final score is a combination of strike, spare and ordinary score frame.

Two strikes in a row are possible. In this case, the score of the first strike is the sum of first two strikes and a first throw of the third frame. The score of the second strike is the sum of the second strike and third Frame.

Two spares in a row are possible. Let’s assume that situation is when there are two spares in a row. The score of the first frame is the sum of its two elements and the first element of next frame. The same situation is for next frame.

If the last frame is a spare. The player is allowed to have a bonus throw. Bonus throw is added to the spare. It is important to note the bonus throw doesn’t belong to any frame.

If the last frame is a strike. The player is allowed to have two bonus throws. They also don't belong to regular frames as well.

Further bonus throws are not granted when a game's last frame is a spare and the bonus throw is a strike.

Perfect consists of all strikes (a total of 12 of them including bonus throws), and has a score of 300

Appendix 5

What is TDD?

Test-Driven Development is based on three laws. Famous software engineer (Bob Martin) describes them

1. If you can’t pass a failing unit test, you can’t further write any production code.
2. Write one unit test at a time. Never write two or more unit tests
3. Write enough of the production code to pass one failing test.

The student begins by writing a unit test for the functionality they intend to write. But you can’t more than one unit test at a time. As soon as the unit test code fails, the student must stop and write production code to cover it. According to rule 3, student should write necessary amount of production code to pass one unit test

If you think about this, you will realize that you simply cannot write very much code at all without compiling and executing something. Indeed, this is really the point. In everything we do, whether writing tests, writing production code, or refactoring, we keep the system executing at all times. The time between running tests is on the order of seconds, or minutes. Even 10 minutes is too long.

Most programmers, when they hear about that technique, think: “This is stupid!”, “It's going to slow me down”, “it's a waste of time and effort”. However, think about what would happen if you walked in a room full of people working this way. Pick any random person at any random time. A minute ago, all their code worked.

If all your code works every minute, how often will you use a debugger? The answer, not very often. It's easier to simply hit ^Z a bunch of times to get the code back to a working state, and then try to write the last minutes worth again. And if you aren't debugging very much, how much time will you be saving? How much time do you spend debugging now? How much time do you spend fixing bugs once you've debugged them? What if you could decrease that time by a significant fraction?

But the benefit goes far beyond that. If you work this way, then every hour you are producing several tests. Every day dozens of tests. Every month hundreds of tests. Over the course of a year, you will write thousands of tests. You can keep all these tests and run them anytime you like! When would you run them? All the time! Any time you made any kind of change at all!

Why don't we clean up code that we know is messy? We're afraid we'll break it. But if we have the tests, we can be reasonably sure that the code is not broken, or that we'll detect the breakage immediately. If we have the tests we become fearless about making changes. If we see a messy code or an unclean structure, we can clean it without fear. Because of the tests, the code becomes malleable again. Because of the tests, software becomes soft again.

But the benefits go beyond that. If you want to know how to call a certain API, there is a test that does it. If you want to know how to create a certain object, there is a test that does it. Anything you want to know about the existing system, there is a test that demonstrates it. The tests can be compared to small design documents, which depict how the system functions and how users can work with it.

When you follow the three rules of TDD, all your code will be testable by definition! And another word for "testable" is "decoupled". In order to test a module in isolation, you must

decouple it. So TDD forces you to decouple modules. Indeed, if you follow the three rules, you will find yourself doing much more decoupling than you may be used to. This forces you to create better, less coupled, designs.

TDD moves (starting)

The TDD cycle can be described in 3 steps: Red, Green, Refactor. At a first stage, you need to write unit test in order to make it fail. In TDD, each new requirement starts with writing a new test. To write a test, the developer should have a clear understanding of the requirement's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. The next step is to write a code which can barely pass the test. The new code doesn't have to be perfect and may, for example, be hardcoded to pass a test. That is allowed because more elegant solution will be provided later. The only purpose of the code is to pass the test; no further (and therefore untested) functionality should be predicted nor 'allowed for' at any stage. The code base should be cleaned up constantly during software development. New additions to the code can be moved from where it was used for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As code grows, the volume of method bodies and other objects can become greater. It is good to split them and name their parts carefully to improve readability and maintainability. It will be useful later in the software development. Inheritance can be rearranged to be more concise and logical, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and for creating clean code.

The analysis is very important because it will help to understand the functionality of the system. In the beginning, you need to understand how a system functions. Each specific functionality should be reflected in the code. In order to understand functionality better, you can also write user stories or use cases. It will help you to cover necessary functionality. Once it is done, you can write a unit test. As the result, there should be a list of tests which cover all functionality of the system. If you return from subsequent stages, you should review an existing code, find out a missing functionality, repeat same steps and proceed again.

TDD moves (get to red)

On the next stage, you can create the testing class where unit tests will be written. A set of particular examples should be wrapped up into one method. This is how you separate functionality. It is important to follow naming convention because the number of examples/tests can be quite large. A set of examples/test is responsible for particular functionality. At this stage, you can write code that might fail the test.

TDD moves (get to green)

On the green stage, you must make the test pass. You don't need to implement some nice logic or pattern or whatever to do it. It is even possible to hardcode or fake it to pass the test. If it is too difficult to pass the test, it is better to simplify test. In other words, make an easier test as an alternative and start over.

TDD moves (get to refactoring)

The last stage is refactoring. The growing code base must be cleaned up regularly during test-driven development. The new code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. It is also important to remove code smells. If the student notices that there is a violation of fundamental design principle and it impacts design quality. If it is detected at the early stage, a lot of issues will be avoided. There are specific and general guidelines for refactoring and for creating clean code.

Appendix 6

List of requirements to grade

- **The requirement is related to wrong output.** Because the requirement doesn't reflect the certain aspect of the functionality of the game it should be mentioned in feedback.
- **The requirement is too vague or complex.** The requirement might cover more pieces of functionality than necessary. It can be too "big". In that case, some aspects of functionality will not be implemented.
- **The requirement is too specific.** The granularity of requirement must be at certain scale. If it is too small then it will lead to longer time of analysis
- **Missing requirement.** Missing requirement leads to incomplete functionality. Thus, the system will not be complete.

Grading guidelines for examples

- **Missing examples.** An example is a concrete output of requirement. If it is missing, it means that the system doesn't have a specific behavior
- **An example related to the wrong requirement.** If a requirement has wrong output, the implementation will go in the wrong direction. Thus, it will lead to unpredictable consequences. It should be mentioned in feedback
- **The Example contains mistakes.** The example shouldn't contain any error. Otherwise, it will lead to wrong functionality
- **Complex example.** An overly complex example will lead to difficulty in the implementation.

Grading guidelines for code

- How students maintain separation of tests and actual code. If the test class contains the implementation, it indicates that the logic is out of the boundaries. If the test class is removed, the logic will not be consistent
- How they analyze refactoring techniques and use them. It is better for students, at least, a tiny amount of refactoring techniques listed in order to understand the full cycle of TDD.
- How certain functionality is covered by tests. Check whether the functionality is implemented by the test case. If it is not, then it is not implemented.

License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Bolot Kasybekov** (date of birth: 26.11.1989),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until the expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until the expiry of the term of validity of the copyright,

of my thesis

Title, "Lab Package Evaluation and Development for the Course "Software Engineering"",

supervised by Dietmar Pfahl,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **08.08.2016**