Integrated Test Development

An integrated and incremental approach to write software of high quality

A. Bijlsma
H.J.M. Passier
H.J. Pootjes
S. Stuurman
Lex.Bijlsma@ou.nl
Harrie.Passier@ou.nl
Harold.Pootjes@ou.nl
Sylvia.Stuurman@ou.nl
Open Universiteit
Heerlen, The Netherlands

ABSTRACT

Creating test cases is a difficult task for students. The number of existing recommendations on how to create test cases is overwhelming. There is a lack of guidelines on how to apply those recommendations one step after another. This problem even becomes more complicated when students are taught to refactor their code as a habit. We propose an approach to teach students how to develop and test their code systematically, with refactoring integrated. In our approach, we pay attention to both functionality and robustness.

CCS CONCEPTS

• **Software and its engineering → Software design engineering**; **Software development techniques**;

KEYWORDS

Testing, Refactoring

ACM Reference Format:
A. Bijlsma, H.J.M. Passier, H.J. Pootjes, and S. Stuurman. 2018. Integrated Test Development: An integrated and incremental approach to write software of high quality. In *Proceedings of The 7th Computer Science Education Research Conference (CSERC'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Students in Software Engineering encounter difficulties with testing their software products [5]. They often use a trial and error approach to find and fix errors. Many students have the following ideas about testing software [7, 20]:

- a program is correct if the compiler accepts it,
- if a program runs and produces some output that looks reasonable, it will work well in all other cases,
- when a program looks correct but the output is not right, trying to switch some things will get the output right, and
- once a program produces output according to the instructor's data, the development process is finished.

In addition to the use of these wrong ideas about correctness, many Software Engineering students encounter difficulties in composing efficient and effective test suites [28]. For example, many students are not able to create a complete test suite without the use of a code coverage tool, the suites they compose often contain a large number of redundant test cases, and they have a tendency to perform minimal and ineffective amounts of testing [4, 15, 28]

These findings correspond with our own experiences. For example, in a course about web application development, students often fail to develop adequate test cases for even simple JavaScript functions: they define an unnecessarily large number of test cases, many of those redundant, and yet they often miss crucial test cases. Overall, we observe the problem that students develop test cases without using an explicit procedure: test cases seem to be composed in an ad hoc manner.

We think that a solution might be to provide students with an explicit procedure to develop test cases.

The problem is more complex, because often, code is not developed in one go. To satisfy non-functional aspects such as readability and maintainability, code is refactored a number of times. Refactoring code can easily affect existing test cases [6]. For example, a new function can be added (using function extraction), or a complex control flow can be simplified. In the first case, extra tests should be added. In the second case, test cases should be adjusted to satisfy certain coverage criteria.

Neither Test First, nor Test Last approaches really help in the problem of creating test cases in a development process that contains refactorings. Test First approaches, with Test Driven Development (TDD) as its most widely known instance, are software development practices in which test cases are written before the code is implemented [8]. In Test Last approaches, the development of test cases takes place when the function's specification and implementation are ready. As we will describe in Section 2 and Section 3, both approaches have their own advantages and disadvantages. For now, we state that neither approach provides any advice as to which test techniques should be applied and in which order, and

CSERC'18, October 2018, Saint Petersburg, Russia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

neither approach provides guidelines on how to handle the problem of testing after a refactoring.

To summarize, the problem we see in teaching students how to test, is a lack of explicit, step by step guidelines on how to develop test cases, and how to maintain test cases after refactorings.

The importance of high quality software needs no argument. It is our duty as lecturers to teach students the importance of producing software of high quality. Systematic testing and refactoring are important techniques to achieve this goal. To teach systematic testing effectively, the following three conditions should be satisfied:

(1) Testing is a recurring activity throughout the whole Software Engineering curriculum and is certainly not a distinct or even a single topic [5].
(2) Testing should not be an activity on its own, but should be integrated in an approach to write software of high quality. We will elaborate on this condition in Sections 3 and 4.
(3) Testing is a complex activity and as such should be supported by some procedural guidance [19, 30, 31].

In this report paper, we focus on conditions two and three, i.e. we will develop an integrated procedure to develop and test software systematically. The unit that we focus on, is the unit of a function. As far as we know, such an integrated procedure is lacking. With a procedure, we mean a step-by-step recipe as for example used by Felleisen et al. [11].

Terminology. We use the terms *Test suite* and *Test case* as follows: A *Test case* is a set of test inputs and expected results to test a particular execution path. Because we focus on function development, the case is a single function. A *Test suite* is a set of related test cases. A suite may contain initialization and cleanup actions specific to the test cases included.

Contributions. We have two contributions. *First*, we describe an incremental and integrated procedure to develop and test code of high quality. The procedure is independent of the programming language or paradigm, with the restriction that we rely on strongly typed languages to prevent having to check whether every input variable conforms to the type that we specify. This contribution is not about approaches, techniques or methods for the development of test cases only. Our contribution is a procedure defining which tests should be created, and in which order, in relation to an iterative and incremental development process.

Secondly, we argue why the procedure developed consists of the right steps in the right order.

Structure of the paper. We describe related work in Section 2. In Section 3, we describe the three well known characteristics of tests that we focus on in our procedure, namely: 1) the level of testing (for example an application or a function), 2) the main approaches (specification based or implementation based testing), and 3) the aspect a test case focuses on (in our approach, core functionality and robustness). In Section 4, we define an ordering on these test activities and aspects described in Section 3. Based on these findings, we describe the procedure in detail in Section 5. In Section 6, we illustrate the detailed procedure at work based on the well known Triangle problem (see for example [18]). In our examples, we use TypeScript. Finally, we draw our conclusions and describe future work in Section 7.

2 RELATED WORK

There are many books about testing. For example, in *Pragmatic Unit Testing in Java with JUnit* [17], attention is paid to *structuring* a unit test in pre-test, test and post-test activities, using the various JUnit *syntax* constructs, to test, for example, error conditions resulting in an exception. In addition, a number of *guidelines* are mentioned that might be important to compose a test case, for example, checking boundary conditions, checking inverse relationships, and forcing error conditions.

Another example of a book about testing is *Software Testing: A Craftsman's Approach* [18], describing a number of main test techniques for unit testing, for example, Boundary Value Testing, Equivalence Class Testing, and Decision Table-Based Testing. Each of these main techniques contain several special techniques, for example, Boundary Value Testing comprises Normal boundary value testing, Robust boundary value testing, Worst case boundary value testing, and Robust worst-case boundary value testing. Each of these main techniques is provided with some guidelines. These guidelines are merely points of attention, as opposed to describing, for example, which techniques should be used in which order.

As far as we know, literature about procedural guidance for test development does not exist, especially not for test development as an integrated part of incremental and iterative software development approaches. What is missing, is a higher level of procedural guidance, i.e. an ordering on all these guidelines, especially in relation to the incremental and iterative development approaches.

2.1 Procedural guidance for testing

Punnekkat et al. mention that work on improving the test design phase is new [27].

A roadmap of relevant challenges to be addressed [2] mentions that one of the challenges is the fact that so many test methods and criteria exist that the capability to make a justified choice, or rather to understand how they can be most efficiently combined, becomes a real challenge.

It is now generally agreed that it is more effective to use a combination of techniques, rather than applying only one, even if judged the most powerful, because each technique may target different types of faults, and will suffer from a saturation effect.

A suggestion for the test process is to use testing patterns [2]. Research should strive to produce effective solutions that are easily integrated into development and do not require deep technical expertise.

2.2 Testing in education

An experiment with students to evaluate the possible impact of knowledge about software testing on the production of reliable code, shows that such knowledge can improve code reliability in terms of correctness in as much as 20% [23]. However, it was also found that instructors that teach introductory programming courses lack proper testing knowledge.

Even the book *How to Design Programs* [11], describing procedural guidance to develop functions systematically, does not pay much attention to how to use all the different types of tests and how they fit into the steps as part of the procedure.

2.3 Test First, and Test Late approaches

Test First approaches, with TDD as its most widely known instance, are software development practices in which test cases are written before the code is developed [1, 8, 13]. In a first step, the interface of, for instance, a function, is specified. After that, a test comprising a number of test cases is developed, to verify the function's behavior. These test cases are considered as a specification of the function to be developed. Finally, the body of the function is completed throughout an iterative process, consisting of the activities coding, refactoring and testing, until all test cases succeed.

In *Test Last* approaches, testing is done after coding. In these approaches, the development of test cases takes place when the function's specification *and* implementation are ready.

Both approaches have their own advantages and disadvantages. An extensive comparison of the two approaches [13] suggests that the main advantage of the Test First methodology lies in the fact that it encourages developers to consistently take fine-grained refactoring steps. Another advantage is that this approach forces the programmer to specify the unit's desired behavior in the form of test cases in advance. On the other hand, Scanniello et al. [29] found that applying the Test First approach TDD often leads to low quality code, i.e. the process encourages developers to write quick-and-dirty code to make the tests pass. Subsequent improvement of the code's quality by refactoring is often ignored, resulting in bad quality code.

Test Last approaches on the other hand, enables one to take the code structure into account, thus ensuring that test cases cover all execution paths and focus on loci where trouble may occur. This is often called 'White Box Testing', as opposed to 'Black Box Testing' that uses only the specification (we will use the terms implementation based testing and specification based testing in this paper). Also, observe that in a modern iterative development process, where both implementation and specification refinement proceed iteratively, practically all testing takes place when some features have already been implemented and other features are yet to be realized. Obviously, this tends to blur the distinction between early and late testing.

The Test First as well as the Test Last approach do not provide any advice as to which tests should be applied and in which order. Logically, Test First approaches start with some specification based based tests, but *at some time*, certain implementation based tests should be added too!

2.4 Related issues

Several papers, e.g. the one by Falessi and Kruchten [9], discuss the concept of *technical debt*. This is defined [24] as 'a design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)'. In terms of testing, this means that any additions or modifications made during iterative program development, must be reflected in the test suite, either by adding new tests or by relocating existing ones. Parodi, Matalonga, Macchi and Solari [26] did an experiment with undergraduate students, expecting to find that applying TDD would have less Technical Debt than those developed with Test Last and ad hoc programming. However, the test results dit not produce any evidence to confirm this.

Punnekkat et al. [27] developed an analysis method based on identifying mistakes made during test case design, resulting in a categorization of mistakes and a meta-method to improve the effectiveness of test case construction. They describe some problems we think are addressed by our approach. First, they mention that a known source of problems is formed by boundaries. Boundaries should always be tested, but are often missed in practice.

Secondly, they observed that students find it easier, in general, to define valid input than to define invalid input, and as such use mostly obvious test cases, and often do not vary in inputs, resulting in low coverage and less robust systems.

3 ACTIVITIES AND ASPECTS OF TESTING

There are many different kinds of tests. These tests can be categorized in several ways. Two main categories can be distinguished: *testing by experimenting with the code behavior* and *analyzing* the implementation and/or the related design products [14]. In our approach, we mainly focus on the first category, testing by experimentation, also called *dynamic testing*. The second category, analyzing, encloses methods as, for example, code walk-throughs, code inspections and correctness proofs, which are out of scope in this report, although, we use code inspection in our procedure.

We distinguish three characteristics of dynamic tests [14, 18]: 1) the level of testing, 2) the approaches for testing, and 3) the aspect a test case focuses on. In the next section, we define an ordering on these approaches and aspects.

3.1 Test level

The first characteristic we consider is the level a test is applied on. Do we apply a test on a single software component or do we test a complete system consisting of several software components? In test literature, a component is often called a unit [3]. Testing a single software component is then called *unit testing*. A unit can be, for example, a function, a method, a class, a module, or even a subsystem [3]. The other level of testing, *system testing*, focuses on complete systems consisting of a number of components. In our approach, we focus on unit testing, where a unit is considered to be a function or a method.

3.2 Test approaches

There are two main approaches to test a unit: *specification based* ('black box') testing and *implementation-based* ('white-box') testing [14].

Specification based testing means testing a piece of software without any knowledge of its implementation. Test cases are developed and their results evaluated solely on the basis of the specifications of the unit under test, i.e. what the piece of program is intended to do. Specification based is also called *functional testing*. Techniques to use for specification based tests are boundary value testing, decision-table-based testing, equivalence classes testing, and the cause-effect-graph technique [14, 18].

Implementation based testing on the other hand, uses information about implementation details, for example, the condition part of an if-then-else statement. Implementation based testing is also

called *structural testing*; the program's structure is used explicitly to derive tests. Implementation based tests can be classified according to certain coverage criteria, as, for example, in increasing finesse, the statement coverage criterion, the edge coverage criterion, the condition coverage criterion, and the path coverage criterion [14].

Having several coverage criteria raises the question 'What coverage criterion should be used?'. In general, there are no rules for determining 'the right' coverage criterion. As stated above, the condition coverage criterion is stronger than the statement coverage criterion, but it is known that satisfying the condition coverage criterion does not guarantee that all faults are detected [16]. The general advice is to combine several coverage criteria together and to use them in a practical way to find the critical parts in the code.

Tools can help to measure how thoroughly software is tested. A number of tools to measure coverage criteria exist; some of them have unique features tailored to a certain application domain [32].

When compared to the 'Test First' and 'Test Late' approaches that we discussed in Section 2.3, we notice that in the 'Test First' approach, the tests are considered to function as the specification, while specification based tests assume the existence of an explicit specification. 'Test Late' approaches may use both specification based tests and implementation based tests.

3.3 The aspects a test case focuses on

An important quality of a test or test case is that it only tests one thing, or one *aspect*, at a time [17]. Our approach addresses the aspects *core functionality* and *robustness* of a unit. Other aspects are, for example, performance, security, profiling and logging. We suppose that our approach also works for those aspects, but we do not cover them in this paper.

3.3.1 Testing core functionality. The first aspect we focus on is the correctness of the core functionality of a function. We specify *core functionality* as a precondition-postcondition pair, where the precondition specifies input values such that the function's body reaches the post condition, without a need for testing on unexpected input values. Such a precondition is called *strong*. Thus, testing of core functionality means taking input values such that the strong precondition is satisfied.

The term *core functionality* bears comparison with the term *main success scenario* as is used in use case modeling, part of the UML language [21]. A main success scenario is described as a typical unconditional happy path scenario of success. A difference, however, is that our concept of core functionality concerns *one* unit of software, i.e. a function or a method yielding a result value, whereas a use case concerns usually a number of sequences of function calls that a system as a whole performs yielding a result of value to an actor.

3.3.2 Testing robustness. The second aspect we focus on is robustness. Robustness is about how well software reacts in abnormal conditions [25]. Leino [22] advocates that every program should have two specifications, one for the case where it ends normally and another one for the case where something goes wrong. Here, we distinguish between *problematic input values* as well as *internal errors*.

A *problematic input value* is a value that causes a processing error while it is executed by the implementation without taking special measures. Robustness regarding problematic input values is reached by weakening the precondition, in an extreme as making the precondition true. As a consequence, the function's arguments must be tested on suitability, and, in case of an unsuitable value, a special action must be taken, for example, throwing an exception. Generally, by weakening a precondition one has to consider what to do with problematic input values leading to a non success scenario. Possibilities are, for example, to throw an exception or to return a special value such as -1. Considerations depend on, for example, the required functionality of the function, the context in which the function is called, and the required efficiency of the function [22].

Weakening the precondition means adapting the specification and changing the implementation. As a consequence, the specification based tests and the implementation based tests must be adjusted.

Robustness regarding *internal errors* concerns implementation related errors, for example possible overflow occurrences or read actions of a file that does not exist. These type of errors can be diagnosed by inspection of the implementation, i.e. code inspections [14]. The results of these diagnoses result in code adaptions, as for example an extra exception, and should be incorporated in the implementation based tests too.

4 ORDERING THE ACTIVITIES AND ASPECTS

In this section, we discuss 'When to test what?', i.e. we define a logical order on the activities and aspects described in the previous section. This ordering forms the basis for the step by step procedure that we describe in the next section.

Core functionality. We start with the aspect core functionality. Testing the core functionality of a function implies specification based as well as implementation based testing. A specification based test (*sbt*) is based on the specification that specifies the core functionality. An implementation based test (*ibt*) is based on the implementation that realises the specified core functionality. The implementation (*impl*) is based on the specification too. As such, an implementation based test is supplementary to a specification based test. Figure 1 shows these *based-on* relations, i.e. the meaning of each arrow is 'provides the information needed to compose'.

To be able to specify the function's signature, precondition and result (*spec*), we need to analyze what the function should do in terms of inputs, processing and output (*ana*). Analyzing and specifying are important activities, because undetected faults have mostly to do with incomplete specifications and missing logic [16] *and* leaving out specifications often leads to low quality code [29].

Notice that Figure 1 shows only the information needed to develop specification based and implementation based tests. The figure says nothing about other activities, for instance, running the tests, or what to do in cases of test failures, i.e. removing errors, or even improving the analysis, specification, implementation and test cases. These topics will be included in Section 5. The dotted arrow on the right symbolizes these other activities.

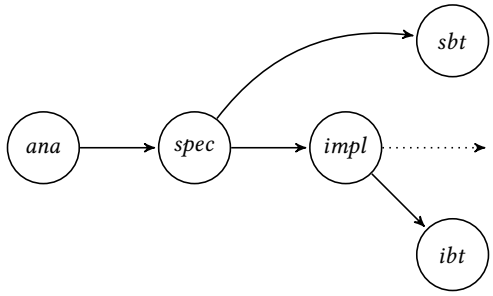

Figure 1: The information needed to test the aspect Core functionality

Robustness. The aspect robustness assumes a specified core functionality. As we have seen in the previous section, making a function robust means: 1) weakening the core function's precondition, and 2) extending the function's body with code to test the arguments on suitability and, if necessary, to perform special actions in cases unsuitable values are provided. This means that the function's specification and implementation both change. As a consequence, the specification based tests and the implementation based tests must be adjusted.

Again, to be able to specify the new precondition and signature (*spec*), we need to analyze to which extent the precondition should be weakened and what to do in cases of unsuitable parameter values, for example, throwing an exception (*ana*). The specification based test (*sbt*) as well as the implementation (*impl*), both based on the changed specification, should be be adapted. An implementation based test (*ibt*), based on the implementation, should be adapted too.

General ordening. We can see that the activity diagram of Figure 1 is applicable to the aspect core functionality as well as to the aspect robustness. Figure 2 shows that the diagram is, as we assume, applicable to the other aspects mentioned in the introduction of this section, for example, performance and security: for each aspect, the specification so far is extended, based on an analysis resulting in an extension of the specification based test as well as extensions in the implementation and accompanying implementation based test.

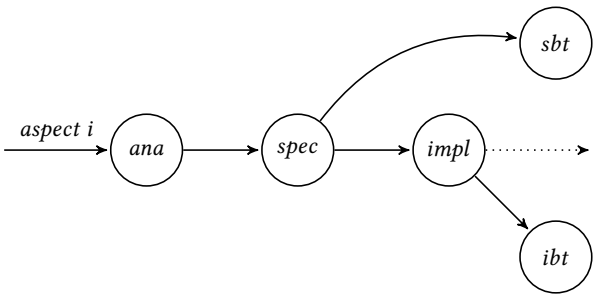

Figure 2: The general information model to test an aspect

In Figure 3, one sees the general procedure for integrated test development. The implementation is adjusted until both the specification based tests and the implementation based tests no longer give any errors. It then may be refactored to enhance the quality of the code. Refactoring may have implications for the implementation based tests, and even for the specification, and specification based tests. Then, the cycle can be instantiated for another aspect. In the next Section, we will explain this procedure in detail.

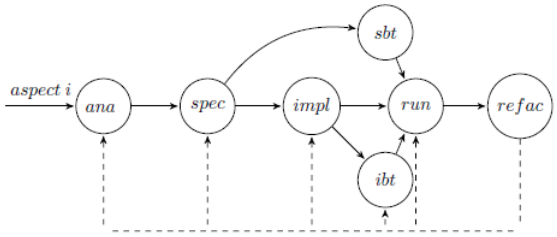

Figure 3: General procedure for integrated test development

5 THE PROCEDURE

The purpose of this paper may be expressed as the ambition to extend Felleisen's method [11] with a step-by-step recipe for setting up unit tests. Therefore, we take the design recipe of Felleisen as a starting point:

Step 1. Write as documentation the main purpose of the function, the signature of the function (the function's name, the parameter's names and their types, and the function's return type).
Step 2. Create test cases for the function.
Step 3. Write the body of the function.
Step 4. Test the function using the test cases.
Step 5. If needed, refactor the function's body to improve the implementation on for example readability and run the test cases again.

This recipe has no explicit steps for testing other than to create test cases and to test the function using the test cases. In this paper, we exend this method with a step-by-step recipe for setting up unit tests, enclosing the effective techniques of specification based testing, implementation based testing, code inspection and refactoring [10]. In addition to attention to core functionality, we strive for robustness.

As is substantiated in the previous section, drawing up a unit test according to our approach assumes a *specification* (to compose the specification based test) as well as an *implementation* (to compose the implementation based test). Furthermore, the specification, the specification based tests, the implementation and the implementation based tests are developed in an iterative and incremental way: aspect after aspect is analyzed, specified, implemented and tested.

Refactoring. After an aspect has been implemented and tested, an obvious additional step is to refactor the code that has been produced so far, that is, to optimize its structure (for example, to improve readability and/or extensibility) without changing the code's

external behavior [12]. Refactoring is an integral part of most iterative and incremental software development approaches [21].

After refactoring, the optimized code can be tested using the same specification based test. If, through refactoring, the structure of the code has been changed, the implementation based test should be examined and adapted. Furthermore, there could be several reasons to optimize the specification, or even to perform some extra analysis, as the dashed arrows in Figure 3 indicate.

In the next two subsections, we instantiate the general procedure for integrated test development for the aspects core functionality and robustness.

Limitations. In our approach, we rrequire functions to be pure functions, i.e. functions with the property that the output is solely dependent on the input values. These functions are deterministic, that is, with the same input they always return the same output.

5.1 The step-by-step guidelines

5.1.1 Aspect: Core functionality. During the first iteration, we focus on the aspect core functionality.

Analyze Analyze the problem and determine the function's signature, including the function's name, the types of the input parameters and the output type. As a guideline, keep in mind that a function should have one single goal. If necessary, design special composite types for the input parameters and the function's result value. Think about a precondition and the result of the function. During this step, the precondition can be as strong as possible, so you can focus on the core functionality of the algorithm without concerning, for example, robustness issues. At this stage, the specification can be written in a natural language or in a more formal language.
Products: 1) a description of the function's goal, 2) the function's signature including the function's name, parameters' names and types, and the output type, and 3) the function's precondition.

Specify Write the results of the analysis step into a specification in a more formal language (in our case JSDoc comment notation), consisting of the function's name, description, parameter names and types, precondition and the result type. The JSDoc tags to use are:
`@function` to give the function's name,
`@description` to describe the goal of the function,
`@param` to describe the function's arguments in terms of names and types,
`@precondition` a predicate on the argument values that must be statisfied[1],
`@returns` to describe the function's return value.
Notice how the analysis products are mapped to the jsdoc tags as follows: 1) the function's description is used to define the description tag, 2) the function's signature is used to define the function tag, the param tag as well as the return tag, and 3) Vervangen door: When working on the core functionality, one chooses a strong precondition.
Students may, thus, analyze the problem using natural language, and are forced to formalize their analysis in the specification step (in particular with regard to the precondition and the specification of the
`@returns`). Finally, implement the function as a stub.
Products: 1) a specification in JSDoc comment notation, and 2) the function implemented as a stub.

Define specification based tests Write examples of function calls with the expected output, based on the problem analysis and the function's signature, preferably in the form of a table. Consider which techniques to use, for example, boundary value testing. Write the specification based tests taking the preconditions into account. Run the tests, which should fail because of the stub, in order to check whether the test suite functions well.
Product: Specification based tests.

Implement Design the function's body, possibly first in pseudocode, and implement the function's body. The results of the previous step, the examples of function calls with expected output, will probably help in writing the algorithm. As soon as the function's body is finished, run the specification based tests. If a test fails, improve the code so that the tests succeed.
Product: A running implementation of the function according to the functional specification.

Define implementation based tests Now we have the implementation at our disposal, we are able to design and implement implementation based tests. Think about the coverage criteria that are valuable. Also, think about how to test whether the arguments that are used in function calls to other functions, satisfy the precondition of those functions. Remember that implementation based testing can lead to extra partitions of the input domain, based on the code's structure. Finally, run the implementation based tests. Again, if a test fails, improve the code so that the specification based as well as the implementation based tests succeed.
Products: 1) implementation based tests, and 2) a running implementation of the function according to the functional specification.

Refactor Often, the implementation can be improved by refactoring, for example, by removing redundant code through an additional function, or by simplifying a complex control structure. In case of refactoring, the specification based test remains applicable. The implementation based tests, however, should often be adapted. When a new function is added, the procedure is followed again for that function.
Products: 1) a refactored implementation, 2) if needed, an adapted implementation based test, and 3) a running implementation of the function according to the functional specification.

The function so far has core functionality but is, probably, not yet robust.

5.1.2 Aspect: robustness. In the second iteration, we focus on the aspect robustness.

Analyze Consider to weaken the precondition. Retain the original postcondition in case the original precondition holds

[1] Actually, the precondition tag is not an existing JSDoc tag, but can be defined as a custom tag.

(so that the contract is not broken), but add clauses to specify other cases. This depends on, for example, the context in which the function will be used. Think about the result value of the function when a problematic input value occurs, for example, a special value such as minus one. When the function should throw an exception, the function's signature should be extended. If a special value should be returned in some cases (sich as −1), the postcondition should be adapted as well. Furthermore, the precondition and the result should be adapted according to the analysis results.
Products: Possible adjusted 1) function's signature, i.e. the parameters' types changed, the exceptions that can be thrown, and the changed result type, and 2) function's precondition and postcondition (in such a way that the original contract is not broken).

Specify If the precondition is weakened, the specification in JSDoc should be adapted. In case of an exception, the tag `@throws` should be used.
Products: 1) adjusted JSDoc specifications, and 2) adjusted stub.

Define specification based tests When the precondition is weakened, the specification based tests should be extended. That is, input values from additional partitions containing problematic values should be considered, resulting in additional specification based tests. These tests should take into account the special return values, for example, minus one or a thrown exception. Run the tests, which should generally fail because of the missing implementation parts, in order to check whether the test suite functions well.
Product: extended specification based tests.

Implement First, design and implement additional code that tests on possible problematic input. This code often comprises some control flow, testing the input values on suitability. When the input values are suitable, the core functionality can be performed. When the input values are problematic, a special value can be returned or an exception thrown, according to the specification. The additional code, testing on possible problematic input, wraps the existing non-robust version using the following pattern:

```
function robustX(parameters) {
  if (!isOK(parameters) {
    throw new RangeError("Arguments not ok");
  }
  return x(parameters);
}
```

As soon as the function's body is extended, run the specification based tests. If a test fails, improve the code so that the specification based tests succeed. Furthermore, search the entire implementation for implementation related errors, for example, possible overflow occurrences. If there is an implementation related error, take the measures needed. If, for example, an additional exception should be thrown, the specification, as well as the specification based tests should be tailored.
Product: a running robust implementation of the function according to the functional specification.

Define implementation based tests The tests should be extended with respect to the added control flow. Run the implementation based tests. If the test fails, improve the code so that the specification based as well as the implementation based tests succeed.
Products: 1) implementation based tests, and 2) a running implementation of the function according to the functional specification.

Refactor Finally, if possible, improve the implementation by refactoring. The specification based tests remain applicable. The implementation based tests, however, should often be adapted.
Products: 1) a refactored implementation, 2) if needed, an adapted implementation based test, and 3) a running implementation of the function according to the functional specification.
Note that in this step we do much more than testing; indeed, the approach shows how testing is interleaved with other development activities.

6 THE PROCEDURE APPLIED

6.1 The Triangle problem

We use the Triangle problem (see, for example, [18]) to show the use of the procedure.

Input. The triangle program has three positive numbers a, b, and c as input, representing the length of three line segments.

Output. In a first version, the program has to indicate whether or not these three line segments form a triangle. In a second version, the program should also indicate the type of the triangle, for example, Equilateral, Isosceles, or Scalene. When the three numbers do not form an triangle, the output should be 'No triangle'. Additionally, we require a, b, and c positive integers, that are less than 200.

The first version is an application of the 'divide et impera' strategy, i.e. first solve a simpler version of the problem and rely on this solution to find the solution for the more complex problem.

In the next subsections, we will follow our procedure and explain each step.

6.2 Aspect core functionality, output boolean

Analyze The problem description gives us the precondition that all three numbers are greater than 0 and smaller than 200. We have chosen `isTri` as name for the function, which will have three arguments of type number. For a proper triangle, the sum of every two sides must be greater than the third side.
Goal: Do three line segments form a triangle?
Signature: isTri(a: integer, b: integer, c: integer): boolean
Precondition: $0 < a < 200, 0 < b < 200, 0 < c < 200$
Returns true if $(a + b > c) \wedge (b + c > a) \wedge (c + a > b)$

Specify Based on the analysis, we can write the specification. Furthermore, we can implement the function as a stub. The specification in JSDoc notation is as follows

```
/**
```

```
 * @function     isTri
 * @description  Do a, b and c form a triangle?
 * @param        a  integer: the length of side a
 * @param        b  integer: the length of side b
 * @param        c  integer: the length of side c
 * @precondition 0 < a < 200,
                 0 < b < 200,
                 0 < c < 200
 * @returns      true if (a + b > c) and
                      (b + c > a) and
                      (c + a > b),
                 otherwise false
 */
export function isTri(a: number,
                      b: number,
                      c: number): boolean {
    return false;
}
```

Notice that in this stage, the precondition is as strong as possible, so we can focus on the core functionality of the algorithm only.

Define specification based tests Based on the way we formulated what should be returned, we create seven function calls that either satisfy or dissatisfy these conditions. Three of them test on boundaries. Table 1 shows the corresponding test cases.

	a	b	c	*Expected output*
1	3	4	5	true
2	2	3	10	false, because $\neg(a + b > c)$
3	2	10	3	false, because $\neg(a + c > b)$
4	10	2	3	false, because $\neg(b + c > a)$
5	2	3	5	false, because $\neg(a + b > c)$, boundary
6	2	5	3	false, because $\neg(a + c > b)$, boundary
7	5	2	3	false, because $\neg(b + c > a)$, boundary

Table 1: Test cases for function `isTri`

Implement Based on the analysis, it is easy to write the body of function `isTri`.

```
export function isTri(a: number, \\
                      b: number, \\
                      c: number): boolean {
  return a + b > c && a + c > b && b + c > a;
}
```

If we run our tests, all test cases succeed.

Define implementation based tests For the function so far, there are no additional implementation based tests. The specification based tests and implementation based tests have the same classes of the input domain and corresponding boundaries.

Refactor Refactoring is not necessary.

The function so far has core functionality, but is not robust yet. Before making the the function robust, we first apply the procedure to a second version of the Triangle problem, i.e. instead of output bool, the output will have the type of triangle.

6.3 Aspect core functionality, output triangle

Analyze We have to determine the kind of triangle. To do this, we define a result type. We could use a string, but it is better to make use of an enumeration type.

```
enum TriangleType {
     Equilateral,
     Isosceles,
     Scalene,
     NotATriangle
}
```

Table 2 shows the rules to determine the kind of triangle, when it is certain that we have a triangle). Notice that the order of these rules is of importance.

Nr	*Name*	*Rule*
1	Equilateral	a=b=c
2	Isosceles	$a = b \vee b = c \vee a = c$
3	Scalene	$a \neq b \neq c$

Table 2: Rules for the kind of triangle

Specify We extend the specification and stubs with the result type `TriangleType`. We rename the function

```
/**
 * @function     triType
 * @description  Which kind of triangle do sides
                 a, b and c form?
 * @param        a  integer: the length of side a
 * @param        b  integer: the length of side b
 * @param        c  integer: the length of side c
 * @precondition 0 < a < 200,
                 0 < b < 200,
                 0 < c < 200
 *  @returns     NotATriangle if
                      (a + b <= c) or
                      (a + c <= b) or
                      (b + c <= a)
                 otherwise:
                 Equilateral if a=b=c
                 Isosceles if a=b or b=c or a=c
                 otherwise Scalene
}
 */
export function triType(a: number, b: number, c:
     number): TriangleType {
  return NotATriangle;
}
```

Define specification based tests Based on the test cases from the previous phase and the rules to determine the kind of triangle, we can derive the test cases, see Table 3. Notice, that the first six test cases are the same as the test cases two until seven in Table 1.

Implementation To implement this extended version of the algorithm, we use the previous version. In this case, we use function `isTri` to determine whether a, b and c form a triangle. If so, we can further investigate the kind of triangle. If the input parameters do not form a triangle, the value NotATriangle is returned.

```
export function triType(a: number,
       b: number,
```

Case	*a*	*b*	*c*	*Expected output*
1	2	3	10	NotATriangle: $\neg(a+b>c)$
2	2	10	3	NotATriangle: $\neg(a+c>b)$
3	10	2	3	NotATriangle: $\neg(b+c>a)$
4	2	3	5	NotATriangle: $\neg(a+b>c)$, boundary
5	2	5	3	NotATriangle: $\neg(a+c>b)$, boundary
6	5	2	3	NotATriangle: $\neg(b+c>a)$, boundary
7	5	5	5	Equilateral: $a=b=c$
8	5	5	6	Isosceles: $a=b$
9	5	5	15	NotATriangle: $\neg(a+b>c)$
10	6	5	5	Isosceles: $b=c$
11	15	5	5	NotATriangle: $\neg(b+c>a)$
12	5	6	5	Isosceles: $a=c$
13	5	15	5	NotATriangle: $\neg(a+c>b)$
14	5	3	4	Scalene:

Table 3: Specification based test cases

```
    c: number): TriangleType {
  let res = TriangleType.NotATriangle;
  if (isTri(a,b,c)) {
    if (a===b && b===c) {
      res = TriangleType.Equilateral;
    }
    else if(a===b || b===c || a===c) {
      res = TriangleType.Isosceles;
    }
    else {
      res = TriangleType.Scalene;
    }
  }
  return res;
}
```

Define implementation based tests The specification based tests already covers the boolean expressions in the if-parts completely for Equilateral and Scalene. For Isosceles, however, there is only one test case, so we add two cases, `b===c` and `a===c`. Table 4 shows the resulting implementation based tests.

Case	*a*	*b*	*c*	*Expected output*
1	2	3	10	NotATriangle: $\neg(a+b>c)$
2	2	10	3	NotATriangle: $\neg(a+c>b)$
3	10	2	3	NotATriangle: $\neg(b+c>a)$
4	2	3	5	NotATriangle: $\neg(a+b>c)$, boundary
5	2	5	3	NotATriangle: $\neg(a+c>b)$, boundary
6	5	2	3	NotATriangle: $\neg(b+c>a)$, boundary
7	5	5	5	Equilateral: $a=b=c$
8	5	5	6	Isosceles: $a=b$
9	6	5	5	Isosceles: $b=c$
10	5	6	5	Isosceles: $a=c$
11	5	3	4	Scalene: $a\neq b\neq c$

Table 4: Implementation based test cases for `triType`

Refactor We apply the 'Decompose conditional' refactoring [12] and use these functions in the if-parts in function `triType`. The resulting code is as follows:

```
/**
 * @function     hasThreeEqual
 * @description  Are all arguments equal?
 * @param a
 * @param b
 * @param c
 * @precondition true
 * @returns      true if a=b=c otherwise false
 */
export function hasThreeEqual(a: number, \\
                              b: number, \\
                              c: number): boolean
                                   {
  return a === b && b === c;
}

/**
 * @function     hasTwoEqual
 * @description  Are the two arguments equal?
 * @param a
 * @param b
 * @param c
 * @precondition true
 * returns       true if a=b or a=c or b=c
 */
export function hasTwoEqual(a: number, \\
                            b: number, \\
                            c: number): boolean {
  return a === b || b === c || a === c;
}

export function triType(a: number, \\
                        b: number, \\
                        c: number): TriangleType {
  let res = TriangleType.NotATriangle;
  if (isTri(a, b, c)) {
    if (hasThreeEqual(a,b,c)) {
      res = TriangleType.Equilateral;
    }
    else if (hasTwoEqual(a,b,c)) {
      res = TriangleType.Isosceles;
    }
    else {
      res = TriangleType.Scalene;
    }
  }
  return res;
}
```

Notice that it is not longer necessary to test function `isTri` explicitly, because we have used all the test cases to test function `triType`.

6.4 Aspect robustness and additional constraints

Analyze According to the problem description as formulated by Jorgenson [18], the input parameters a, b, and c must satisfy the following condition: $0 < a, b, c < 200$. At the same time, we weaken the precondition, i.e. we decide to have no precondition at all. We decide that if an input parameter does not confirm the extra condition, an error value will be generated. The corresponding signature will be:
Goal: Do three line segments form a triangle?
Signature: isTri(a:number, b:number, c:number): boolean throws RangeError

Precondition: *true*
Exception: RangeError, if $\neg(0 < a, b, c < 200)$

Specify The specification and signature of function `triType` becomes:

```
/**
 * @function      triType
 * @description   Which kind of triangle?
 * @param         a integer: the length of side a
 * @param         b integer: the length of side b
 * @param         c integer: the length of side c
 * @precondition  true
 * @returns       the type of the triangle (one out
      of TriangleType)
 * @throws        RangeError if not(0 < a,b,c <
     200)
 */
export function triType(a: number, \\
                        b: number, \\
                        c: number): TriangleType
```

Define specification based test For the specification based tests, we now must add test cases so that an exception of type `RangeError` will be generated. For each argument we define four test cases: one on the boundaries 0 and 200 and two for values outside the specified range $0 < a, b, c < 200$. Table 5 shows the resulting specification based tests.

	a	b	c	*Expected output*
1	2	3	10	NotATriangle: $\neg(a + b > c)$
2	2	10	3	NotATriangle: $\neg(a + c > b)$
3	10	2	3	NotATriangle: $\neg(b + c > a)$
4	2	3	5	NotATriangle: $\neg(a + b > c)$, boundary
5	2	5	3	NotATriangle: $\neg(a + c > b)$, boundary
6	5	2	3	NotATriangle: $\neg(b + c > a)$, boundary
7	5	5	5	Equilateral: $a = b = c \wedge \neg NotATriangle$
8	5	5	6	Isosceles: $a = b \wedge \neg NotATriangle$
9	6	5	5	Isosceles: $b = c \wedge \neg NotATriangle$
10	5	6	5	Isosceles: $a = c \wedge \neg NotATriangle$
11	5	6	7	Scalene: $a \neq b \neq c \wedge \neg NotATriangle$
12	0	5	5	RangeError: $a < lower$, boundary
13	-100	5	5	RangeError: $a < lower$, boundary
14	200	5	6	NotATriangle: $a = upper$, boundary
15	201	5	5	RangeError: $a > upper$, boundary
16	5	0	5	RangeError: $b < lower$, boundary
17	5	-100	5	RangeError: $b < lower$, boundary
18	5	200	6	NotATriangle: $b = upper$, boundary
19	5	201	5	RangeError: $b > upper$, boundary
20	5	5	0	RangeError: $c < lower$, boundary
21	5	5	-100	RangeError: $c < lower$, boundary
22	5	6	200	NotATriangle: $c = upper$, boundary
23	5	5	201	RangeError: $c > upper$, boundary

Table 5: Specification based test cases for `triType`

Implement Compared to the previous version, we now must also check whether all arguments are in the range 0 to 200. For this purpose, we write a function `argsInRange` with signature and stub:

```
/**
 * @function      argsInRange
 * @description   For all elements el of ar:\\
                   low < el < high ?
 * @param         ar array of numbers
 * @param         low lower boundary not included
 * @param         high upper boundary included
 * @precondition  low < high
 * @returns       true if for all elements of ar:
 *                  low < el < high else false
 */
function argsInRange(ar: Array<number>, \\
                     low: number,
                     high: number): boolean {
  return true
}
```

Notice that we have used an array to make the function more general. Also notice that we have a precondition low < high. Because we use `argsInRange` as an internal function of `TriangleType`, we can always take care that low < high holds. Table 6 lists the test cases for this function.

Case	a	b	c	*Expected output*
1	[1,2,3,4,5,6,199,200]	0	200	true
2	[0,1,2,3,199,200]	0	200	false
3	[1,2,3,199,200,200.01]	0	200	false
4	[0,2,3,199,200.1]	0	200	false

Table 6: Specification based test cases for `argsInRange`

To implement the function `argsInRange`, we can use the standard array function `every`:

```
function argsInRange(ar: Array<number>, \\
                     low: number, \\
                     high: number): boolean {
  return ar.every((el)=>{ return el > low && el <=
      high});
}
```

Instead of modifying the function `triType`, we write a new function `triAngle` that first checks the arguments. If the arguments are not correct it throws an exception; otherwise it calls function `triType`.

```
/**
* @function      triAngle
* @description   Which kind  of triangle?
* @param         a integer: the length of side a
* @param         b integer: the length of side b
* @param         c integer: the length of side c
* @precondition  true
* @returns       TriangleType
* @throws        RangeError if not(0 < a, b, c < MAX
     )
*/
export function triAngle(a: number, \\
                         b: number, \\
                         c: number): TriangleType {
  let ar=[a,b,c];

  function argsInRange(ar: Array<number>,\\
                       low: number\\
                       high: number):boolean {
    return ar.every((el) => { return el > low && el
        < high });
  }
```

```
  if (!argsInRange(ar, 0, MAX)) {
    throw new RangeError("Argument(s) not in range"
        );
  }
  return triType(a,b,c);
}
```

Define implementation based test Now, we examine the code. Because the arguments are between 0 and 200, overflow will not occur. Because we use a language that does not know integers, we have a problem: the comparison of floats. Simply using operator `===` may cause a problem, due to internal representation and rounding errors. Instead, we can check whether the absolute value of `0.1+0.2-0.3` is less than some value `EPS`. For `EPS`, we can use the constant `NUMBER.EPSILON`. So, instead of using operator `===`, we use a function `isEqual (a: number, b: number): boolean` that evaluates the expression `Math.abs(a-b)< NUMBER.EPSILON`. As a result, we rewrite some functions and use function `isEqual` in stead of using `===`.

```
/**
* @function     isEqual
* @description  Compares two floating point numbers
     \\
                 taking rounding errors into account
* @param        a number
* @param        b number
* @precondition true
* @returns      true if a < b, otherwise false
*/
export function isEqual(a: number, \\
                        b: number): boolean {
  return Math.abs(a-b) < Number.EPSILON;
}
```

Refactor There is no need for refactoring anymore.

7 CONCLUSIONS AND FUTURE WORK

We developed a stepwise procedure enclosing and ordering a selected number of test methods, integrated in a process of analysis, specification, implementation, testing, and refactoring, applicable in a context of a first year course about software construction and testing. Such a step by step instruction, for students, on how to integrate writing tests while developing functions, including advice on how to proceed after having refactored the code, was missing.

Our approach differs from both Test First and Test Last approaches. We use an explicit specification (instead of using tests themselves as a form of specification, as is usable in many Test First approaches), and in contrast with Test Last approaches, we create tests *before and after* implementation, and, we revise the tests after each refactoring.

With our approach, we have solved the problem that it becomes a real challenge for students to understand how test methods and test criteria may be combined most efficiently [2]. It is exactly this absence of guidance where our approach jumps into, and gives certain scaffolding. Our procedure can be considered as a process pattern [2].

Our approach gives guidance to solve this problem, by distinguishing focusing on separate aspects in each cycle. Our approach focuses on the aspects core functionality and robustness, and gives guidance in the form of a step-by-step approach in which the function's precondition is relaxed concerning the specification as well as the implementation of the function. We presume that this approach is applicable to other aspects as well.

We prevent students to write quick and diry code, as is often seen in Test First approaches, because they are forced to analyse the problem first, and write a precise specification. On top of that, we have made refactoring code an explicit step in the procedure.

Another future enhancement of our guidelines could be to give more detailed guidelines to create test cases for both specification based testing and implementation based testing.

We have started to teach our students using this procedure, and will report later about our findings.

REFERENCES

[1] Kent Beck. 2002. *Test Driven Development: By Example.* Addison-Wesley Logman Publishing Co., Inc., Boston, MA, USA.

[2] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *2007 Future of Software Engineering (FOSE '07).* IEEE Computer Society, Washington, DC, USA, 85–103.

[3] Robert V. Binder. 1999. *Testing Object-oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] J. Carver and N. Kraft. 2011. Evaluating the testing ability of senior-level computer science students. In *24th IEEE-CS Conference on Software Engineering Education and Training (CSEE T).* Springer, Berlin, Germany, 169–178.

[5] Henrik Bærbak Christensen. 2003. Systematic Testing Should Not Be a Topic in the Computer Science Curriculum! *SIGCSE Bull.* 35, 3 (June 2003), 7–10.

[6] Arie van Deursen and Leon Moonen. 2002. The video store revisited–thoughts on refactoring and testing. In *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, XP 2002.* Springer, Berlin, Germany, 71–76.

[7] Stephen H. Edwards. 2004. Using Software Testing to Move Students from Trial-and-error to Reflection-in-action. *SIGCSE Bull.* 36, 1 (March 2004), 26–30.

[8] Hakan Erdogmus, Forrest Shull, Burak Turhan, Lucas Layman, Grigori Melnik, and Madeline Diep. 2010. What Do We Know about Test-Driven Development? *IEEE Software* 27 (2010), 16–19.

[9] Davide Falessi and Philippe Kruchten. 2015. Five Reasons for Including Technical Debt in the Software Engineering Curriculum. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15).* ACM, New York, NY, USA, 28:1–28:4.

[10] Sheikh Umar Farooq and Smk Quadri. 2013. An Externally Replicated Experiment to Evaluate Software Testing Methods. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE '13).* ACM, New York, NY, USA, 72–77.

[11] Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Krishnamurthi Shriram. 2001. *How To Design Programs, An Introduction to Programming and Computing.* The MIT press, Cambridge, Massachusetts, Londen, England.

[12] Martin Fowler, Steven Fraser, Kent Beck, Bil Caputo, Tim Mackinnon, James Newkirk, and Charlie Poole. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. 2017. A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? *IEEE Transactions on Software Engineering* 43, 7 (2017), 597–614.

[14] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 1991. *Fundamentals of Software Engineering.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[15] Hisham Haddad. 2002. Post-graduate Assessment of CS Students: Experience and Position Paper. *J. Comput. Sci. Coll.* 18, 2 (Dec. 2002), 189–197.

[16] Hadi Hemmati. 2015. How Effective Are Code Coverage Criteria?. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on.* IEEE, IEEE Computer Society, Los Alamitos, California, United States.

[17] Andy Hunt and Dave Thomas. 2003. *Pragmatic Unit Testing in Java with JUnit.* The Pragmatic Bookshelf, Raleigh, USA.

[18] Paul C. Jorgensen. 2014. *Software Testing: A Craftsman's Approach* (4st ed.). CRC Press, Inc., Boca Raton, FL, USA.

[19] Paul A Kirschner, John Sweller, and Richard E Clark. 2006. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist* 41, 2 (2006), 75–86.

[20] Yifat Ben-David Kolikant. 2005. Students' Alternative Standards for Correctness. In *Proceedings of the First International Workshop on Computing Education Research (ICER '05).* ACM, New York, NY, USA, 37–43.

[21] C. Larman. 2009. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[22] K. Rustan M. Leino. 1995. Constructing a program with exceptions. *Inform. Process. Lett.* 53, 3 (1995), 159 – 163.

[23] Otávio Augusto Lazzarini Lemos, Fábio Fagundes Silveira, Fabiano Cutigi Ferrari, and Alessandro Garcia. 2017. The impact of Software Testing education on code reliability: An empirical assessment. *Journal of Systems and Software* 137, March (2017), 497–511.

[24] Steve McConnell. 2013. Managing Technical Debt. Retrieved December 22, 2017 from https://www.sei.cmu.edu/community/td2013/program/upload/technicaldebt-icse.pdf. (2013).

[25] B. Meyer. 1997. *Object-Oriented Software Construction.* Prentice-Hall, Upper Saddle River, NJ, USA.

[26] Eugenia Parodi, Santiago Matalonga, Darío Macchi, and Martín Solari. 2016. Comparing technical debt in student exercises using test driven development, test last and ad hoc programming. In *Computing Conference (CLEI), 2016 XLII Latin American.* IEEE, IEEE Computer Society, Los Alamitos, California, United States, 1–10.

[27] Sasikumar Punnekkat, Sigrid Eldh, and Hans Hansson. 2011. Analysis of Mistakes as a Method to Improve Test Case Design. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST 2011)* 00 (2011), 70–79.

[28] Alex Radermacher, Gursimran Walia, and Dean Knudson. 2014. Investigating the Skill Gap Between Graduating Students and Industry Expectations. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014).* ACM, New York, NY, USA, 291–300.

[29] Giuseppe Scanniello, Simone Romano, Davide Fucci, Burak Turhan, and Natalia Juristo. 2016. Students' and Professionals' Perceptions of Test-driven Development: A Focus Group Study. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16).* ACM, New York, NY, USA, 1422–1427.

[30] Jeroen J.G. van Merriënboer, Richard E. Clark, and Marcel B.M. De Croock. 2002. Blueprints for complex learning: The 4C/ID-model. *Educational Technology Research and Development* 50, 2 (2002), 39–61.

[31] Jeroen J.G. van Merriënboer and Paul A. Kirschner. 2013. *Ten Steps to Complex Learning, a systematic appraoch to four-component instructional design* (second ed.). Taylor & Francis, New York, NY, USA.

[32] Qian Yang, J. Jenny Li, and David M. Weiss. 2009. A Survey of Coverage-Based Testing Tools. *Comput. J.* 52, 5 (2009), 589–597.