Integrated Test Development

Technical Report (TR-OU-ICA-2018-01)

A. Bijlsma[*1], H.J.M. Passier[†1], H.J. Pootjes[‡1] and S. Stuurman[§1]

[1] *Open Universiteit, Faculty of Management, Science and Technology, Department of Computer Science, Postbus 2960, 6401 DL Heerlen, The Netherlands*

June 25, 2018

[*] lex.bijlsma@ou.nl
[†] harrie.passier@ou.nl
[‡] harold.pootjes@ou.nl
[§] sylvia.stuurman@ou.nl

1 Introduction

The problem. Students in Software Engineering encounter difficulties with testing their software products [5]. They often use a trial and error approach finding and fixing errors instead of applying a systematic approach.

Many students have the following ideas about testing software [7, 20]:

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

Beside using these wrong ideas about correctness, many Software Engineering students encounter difficulties in composing efficient and effective test suites [29]. For example, students often 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, 29]

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.

The problem is even more complex, because code is usually 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, an extra tests should be added. In the second case, test cases should be adjusted to satisfy certain coverage criteria.

For testing software, there exist a overwhelming number of test techniques. Examples are Boundary Value Testing, Equivalence Class Testing, and Decision TableBased Testing. Each of these main techniques contains several special techniques, as for example Boundary Value Testing encloses Normal boundary value testing, Robust boundary value testing, Worst case boundary value testing, and Robust worst-case boundary value testing [18]. These techniques are often provided with some points of attention.

What is missing? What is missing, however, is a step by step procedure on which of these techniques should be applied and in which order alongside the development process of a unit of software. One could think of Test Driven Development (TDD) [8]

as such a procedue. TDD, however, only indicates that test cases should be defined; no advice is given about which of the test techniques should be applied, the order in which they should be applied, and the aspects with they concern.

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 for writing software of high quality.

3. Testing is a complex activity and as such should be supported by some procedural guidance [19, 31, 32].

In this report, we will focus on conditions two and three, i.e. we will develop an integrated procedure for developing and testing software systematically. As far as we know, such an integrated procedure is missing. With a procedure, we mean a step-by-step recipe as for example used by Felleisen et al. in the book *How to Design Programs* [11].

Terminology. In this report, we use the terms *Test suite* and *Test case*. These terms are used 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. The procedure defines in which order the known test techniques should be applied in relation to the iterative and incremental development process. Besides attention to core functionality, we strive for robustness. We illustrate our procedure by means of two cases. The procedure is independent of the programming language or paradigm. *Secondly*, we argue why the procedure developed consists of the right steps in the right order.

Remark. The symbol $\triangle$ indicates the end of a remark and the resumption of normal text. Likewise, the symbol $\square$ indicates the end of an example.

This report. In this report, we focus on the development of test cases as part of a unit test for functions and methods using the programming language TypeScript. In Section 2 we describe three well known characteristics of tests we focus on in our procedure, namely: 1) the level of testing (for example an application or a function),

2) the main approaches (black-box and white-box testing), and 3) the aspect a test case focuses on (core functionality and robustness). We discuss these characteristics based on an example, namely a program that computes the solutions of a quadratic equation of the form $ax^2 + bx + c = 0$. In Section 3, we define an ordering on these test activities and aspects described in Section 2. Based on these findings, we describe in Section 4 the procedure in detail. In Section 5, we show the detailed procedure at work based on the well known Triangle problem (see for example [18]). We describe related work in Section 6. Finally, we draw our conclusions and describe future work in Section 7.

2 Activities and aspects of testing

There are many different kinds of tests that can be performed on a software project. These tests can be categorized in various 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 this report, we focus mainly 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. However, we will 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. We discuss these characteristics in the following three subsections In the next section, we define an ordering on these approaches and aspects.

2.1 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 than 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 this report, we focus on unit testing, where a unit is considered to be a function or a method. In our examples, we use JavaScript and TypeScript

2.2 Test approaches

There are two main approaches to test a unit: *black-box* testing and *white-box* testing [14]. We give descriptions and examples of both approaches.

2.2.1 Black-box testing.

Black-box 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. Black-box testing is also called *specification based testing* or *functional testing*. Examples of black-box tests are boundary value testing, decision-table-based testing, equivalence classes testing, and the cause-effect-graph technique [14, 18].

Example. Suppose we have to test a program that computes the solutions of a quadratic equation of the form $ax^2 + bx + c = 0$. The specification is as follows:

Signature: `function roots(a:number, b:number, c:number): (number | null)`
Precondition: $a, b, c \in \mathbb{R} \wedge \neg(a = b = c = 0)$
Return: all x values such that $ax^2 + bx + c = 0$

This specification requires that a, b, and c are of type number. Notice that the cases a equals zero, b or c differs from zero (a straight line) and a and b equal zero, c is unequal to zero (a straight line with gradient zero) are included by the precondition. The implementation has to distinguish these both special cases. Furthermore, the precondition excludes a, b and c all equal zero, in which case an infinite number of solutions exist.

This specification allows for several implementations, as for example the quadratic formula, factorization, or a numerical method as for example Regula Falsi or Newton-Raphson.

Taken this specification into account, Table 1 shows our test cases so far. From the theory of quadratic functions, we know that a quadratic equation of the form $ax^2 + bx + c = 0$ has zero, one, or two solutions (test cases 1, 2 and 3). In case a equals zero, b and c are unequal to zero, there is one solution, namely $x = -c/b$ (test case 4). In case a and b equal zero and c is unequal to zero, there is no solution at all (test case 5). □

Test case	a	b	c	*Expected output*
1	1	4	5	No solutions
2	1	4	4	$x = -2$
3	1	-2	-3	$x_1 = -1$, $x_2 = 3$
4	0	2	-4	$x = 2$
5	0	0	1	No solutions

Table 1: Black-box test cases for testing function `roots`

2.2.2 White-box testing.

White-box testing on the other hand, uses information about implementation details, for example, the condition part of an if-then-else statement. White-box testing is also called *structural testing*; the program's structure is used explicitly to derive tests. White-box 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 [33].

Example. Consider the following program fragment, implementing the quadratic formula for calculating the solutions of a quadratic equation with input values a, b and c. Following our analysis so far, an implementation could be:

```
function roots(a:number, b:number, c:number): (number | null)[] {
  let res: (number | null)[] = [null, null];
  // Discriminant d
  let d = b*b - 4*a*c;
  // Both a straight line with gradient zero and ...
  // a quadratic equation with d < 0 have no solutions
  if (!((a===0 && b===0) || d < 0 )) {
    // A straight line has one solution
    if (a===0 && b!== 0) {
      res[0] = -c/b;
    }
    // A quadratic equation with d > 0 has two solutions
    else if (d > 0) {
      let rt = Math.sqrt(d);
      res = [(-b + rt) / (2 * a), (-b - rt) / (2 * a)];
    }
    // Finally, a quadratic equation with d = 0 has one solution
    else {
      res[0] = -b / (2 * a);
    }
  }
  return res;
}
```

Suppose we aim for the edge coverage criterion. This means that we have to choose values for the variables a, b, and c such that all boolean conditions in the if-then-else statements are true as well as false. Figure 1 shows the corresponding control-flow graph of the implementation of function `roots`. The test cases listed in Table 1 are plotted with similar numbers beside the edges. As we could see, the black-box test cases in Table 1 satisfy the edge criterion. So far, there are no differences between the black-box and white-box test cases. □

Remark. To be honest, a previous implementation of function `roots` contained an extra if-then statement in the last else-branch, namely to test explicitly whether d equals zero, as showed in the following code fragment:

```
// ... other code
else {
  if (d===0) {
     res[0] = -b / (2 * a);
  } // Implicit else-branch with null statement
}
return res;
```

As is shown in the code fragment, this implementation resulted in an implicit else-branch containing a null statement. By drawing the edge coverage graph, enriched with the test cases, it seemed that the test case corresponding to this implicit else-branch was missing (number 2 in Figure 1). A first intuitive response was to add

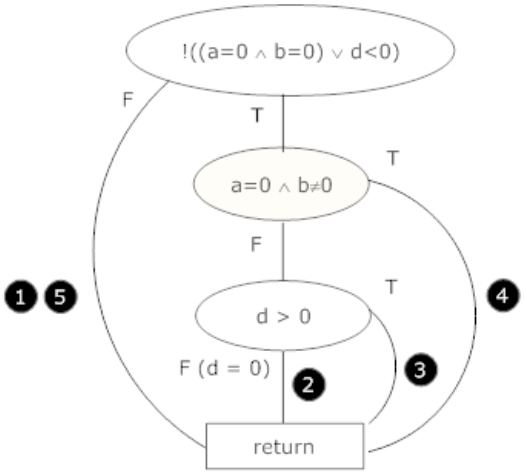

Figure 1: Control-flow graph concerning edge coverage

the missing test case. Closer inspection, however, revealed that the extra else-branch could never be reached. As a result, we removed the extra if-then statement, resulting in less complex code. △

The example in the remark shows the benefit of using a coverage graph. It helps to find missing test cases and it makes code complexity visible. Testing only could not have revealed this unnecessary if-then statement. It was the enriched edge coverage graph combined with code inspection that revealed the problem.

Instead of drawing a coverage graph yourself, it is possible to use a coverage tool. We use Istanbul[1] as a coverage tool and this tool also indicated a branch that was not reached by our test cases.

Remark. It should be noted that the cases $(a = 0, b \neq 0)$ (a straight line) might have been missed as part of the analysis. In that case, the denominator $2 * a$ should be pointed as problematic during implementation. Consequently, special measures, as we have already applied, must still be taken in the implementation as well as the extra test cases 4 and 5 in Table 1 should be added. Notice that in this situation, the test cases 4 and 5 would have been part of the white-box test in stead of the black-box test. It appears that a white-box test functions as a kind of safety net too if cases are missed as part of specifying black-box test. △

It is illustrative to show the difference between black-box and white-box testing in another way. Suppose that, instead of the quadratic formula, we have used the numerical method of Newton-Raphson for calculating the solutions in an iterative way until a

[1] `https://istanbul.js.org/`

specific precision *eps* has been reached:

```
function roots(a: number, b: number, c: number): (number | null)[] {
  // Declarations and instantiations
  let result: (number | null)[] = [null, null],
      eps    = Number.EPSILON,
      xn = 1,
      xn1 = xn-(a*xn*xn+b*xn+c)/(2*a*xn+b);
  while (Math.abs(xn1-xn) >= eps) {
    xn=xn1;
    xn1 = xn-(a*xn*xn+b*xn+c)/(2*a*xn+b);
  }
  // Code to calculate other solutions and assigning them to variable
      result
  return result;
}
```

This implementation does not use the quadratic formula and does not enclose a denominator $2a$. Instead, there is a denominator $2ax_n + b$ and as a consequence, a value of a equals $-b/2x_n$ causes arithmetic problems. Furthermore, the boolean expression $|x_{n+1} - x_n| \geq eps$ needs attention, i.e. some extra test cases are needed here as we have to be careful about precision problems due to the constant *eps*.

Notice that the specification and therefore also the black-box tests are independent of the implementation chosen.

2.2.3 Test techniques and white-box versus black-box testing

As we have seen, in applying white-box as well as black-box testing, test cases are derived on partitioning the input domain into suitable equivalent classes based on the specification or the analysis, assuming that the program's behavior is similar for all elements of each class.

Some errors, however, just happen to be at the boundary between classes. Common mistakes are for example the use of $\geq$ instead of $>$ or $==$ instead of $\leq$. A way of detecting this type of errors, is by using input values not only inside the classes but also values at their boundaries, i.e. boundary testing. Again, these boundaries can be found in the specification as well as in the implementation.

Generally, techniques as for example boundary testing and equivalence class testing are applicable to black-box as well as to white-box testing. Black-box and white-box determine the *source of information* test cases are composed on: the specifications versus the implementation.

In relation to black-box testing, white-box testing can lead to extra partitions of the input domain based on the code's structure. For example, in the first implementation of function `roots`, we had already distinguished the situations $(a = 0, b = 0)$ and $(a = 0, b \neq 0)$. Both situations gave rise to an extra partition.

2.3 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]. In this report, we focus on the aspects *core functionality* and *robustness*

of a unit. Other aspects are for example performance, security, profiling and logging, which are all out of scope in this report.

2.3.1 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 for sure 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 to analyse and design object oriented systems [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.

Example. We have already shown an example of testing core functionality in Section 2.1, paragraphs black-box testing and white-box testing. □

2.3.2 Testing robustness

The second aspect we focus on is robustness. Robustness is about how well software reacts on abnormal conditions [26]. Leino [23] 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 can 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 as for example throwing an exception. Weakening the precondition means adapting the specification and changing the implementation. As a consequence, the tests related to the specification (the black-box tests) and the tests related to the implementation (the white-box test) must be adjusted.

Robustness regarding *internal errors* concerns implementation related errors as for example possible overflow occurrences and 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 white-box tests too.

Example: a problematic input value. Following our example of calculating the solutions of a quadratic equation, we discerned the input value $a = 0, b = 0, c = 0$ as problematic. The precondition excludes this combination of input values, i.e. $a, b, c \in \mathbb{R} \neg(a = b = c = 0)$, and as such makes the function's caller responsible to prevent for this input combination.

Now, we will make the function robust by weakening the precondition and take some measures in the functions body. The precondition becomes: $a, b, c \in \mathbb{R}$, i.e. all input values should be of type number. No other condition needs to be met. As a consequence, we have to test on input values of $a = b = c = 0$ inside the function body. If this test shows that all three parameters equals zero, a special action should be taken. In our example, the special action could be to terminate the function's body raising 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 [23]. In Section 5 we show how to handle these exceptional cases in a proper way.

Example: an internal error. As an example of an internal error, consider the values of the input parameters in case the quadratic formula is implemented as the body of function `roots`. There is the risk that one of the sub expressions b^2,$4ac$, and $1/(2a)$ exceeds the value of `Number.MAX_VALUE`. As a consequence, we have to test these expressions on overflow and to take the necessary measures as for example throwing an exception.

Another internal error could appear if the value of the discriminant is compared with value zero. Because floating point numbers almost never perfectly compare to other numbers, as zero in our implementation, this is a potential source of problems. □

Again, we will show how to handle these exceptional cases in an eloquent way in Section 5. Finally, adapting the implementation means revising the white-box test.

3 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 in the next section.

Core functionality. We start with the aspect core functionality. Testing the core functionality of a function implies black-box as well as white-box testing. A black-box test (*bbt*) is based on the specification, specifying the core functionality. A white-box test (*wbt*) is based on the implementation, implementing the specified core functionality. The implementation (*impl*) is based on the specification too. As such, a white-box test is supplementary to a black-box test. Figure 2 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 (*anal*). 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 [30].

Notice that Figure 2 shows only the information needed to develop black-box and white-box tests. The figure says nothing about other activities as running the tests and 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 4. The dotted arrow on the right symbolizes these other activities.

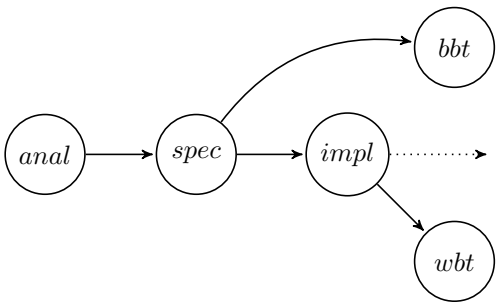

Figure 2: The information needed to test the aspect Core functionality

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 tests related to the specification (the black-box tests) and the tests related to the implementation (the white-box test) must be

adjusted.

Again, to be able to specify the new precondition and signature (*spec*), we need to analyze to which extend the precondition should be weakened and what to do in cases of unsuitable parameter values, as for example throwing an exception (*anal*). The black-box test (*bbt*) as well as the implementation (*impl*), both based on the specification, should be be adapted. A white-box test (*wbt*), based on the implementation, should be adapted too.

General ordening. It follows that the activity diagram of Figure 2 is applicable on the aspect core functionality as well as on the aspect robustness. It is assumable that the diagram is applicable on the other aspects mentioned in the introduction of this section, as for example performance and security: For each aspect, the specification so far is extended based on an analysis resulting in an extension of the black-box test as well as extensions in the implementation and belonging white-box test. Figure 3 shows this generalized diagram.

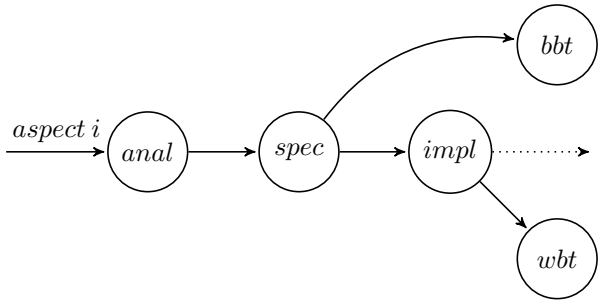

Figure 3: The general information model to test an aspect

4 The procedure

The purpose of this report may be expressed as the ambition to extend Felleisens methodology [11] with a step-by-step recipe for setting up unit tests. Therefore, we take the design recipe of Felleisen as a starting point, consisting of the following steps:

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 functions 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 report, we exend this methodology with a step-by-step recipe for setting up unit tests, enclosing the effective techniques of white-box testing, black-box testing, code inspection and refactoring [10]. Besides 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 for composing the black-box test as well as an implementation for composing the white-box test. Furthermore, the specification, the black-box tests, the implementation and the white-box tests are developed in an iterative and incremental way, i.e. aspect after aspect is analyzed, specified, implemented and tested.

Refactoring. After an aspect has been implemented and tested, an obvious extra step is to refactor the code 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]. That way, refactoring is a logical addition to our procedure.

After a refactoring has taken place, the optimized code can be tested using the same black-box test. If through refactoring the structure of the code has been changed, the white-box test should be examined and adapted. Furthermore, there could be several reasons to optimize the specification or even to perform some extra analysis. Figure 4 shows the high level procedure. In the next two subsections, the procedure is instantiated for the aspects core functionality and robustness.

Limitations. In this report, we restrict ourselves to write pure functions only, i.e. functions whose output is solely dependent on the input values. These functions are deterministic, that is with the same input they always return the same output. As a consequence, using a random function, for example, in the function's body is not allowed. Furthermore, we restrict ourselves to JavaScript and TypeScript as languages and belonging additional languages and tools as JSDoc for code's documentation.

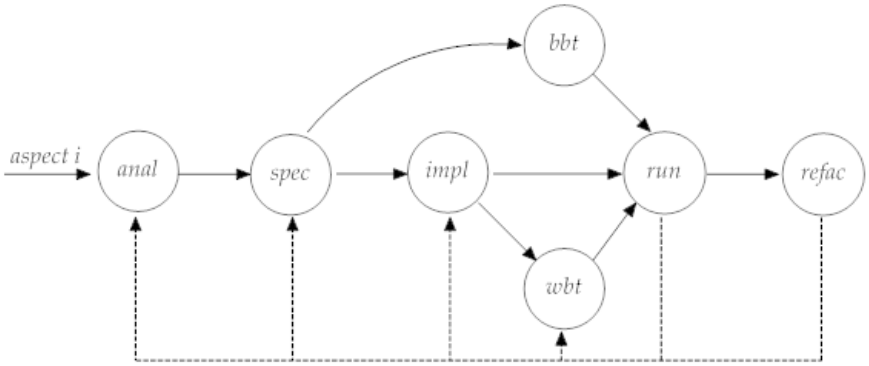

Figure 4: The procedure

4.1 The procedure

4.1.1 Aspect core functionality

In the first iteration we focus on the aspect core functionality:

Analyze. Analyze the problem the function has to solve and determine the function's signature, including the function's name and the types of the input parameters as well as the output type. Use as a guideline that functions should have one 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. In 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 so far can be notated in a natural language as well as in a more formal language as we have used in the example of Section 2. *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 down the results of the analysis step into a specification in JSDoc comment notation, consisting of the function's name, description, parameter names and types, precondition and the result type. The JSDoc tags that can be used 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[2],
`@returns` to describe the function's return value.

Notice that 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

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

tag, and 3) the function's preconditions is used to define the precondition tag. Finally, implement the function as a stub.

Products: 1) a specification in JSDoc comment notation, and 2) the function implemented as a stub.

Define black-box test. 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 like Table 1. Consider which of the black-box test to use as for example boundary value testing (see Paragraph 2.2.1). Write the black-box test 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: Black-box test.

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, can possibly help in writing the algorithm. As soon as the function's body is finished, run the black-box tests. If a test fails, improve the code so that the tests will succeed.

Product: A running implementation of the function according to the functional specification.

Define white-box test. Now we have the implementation to our disposal, we are able to design and implement white-box tests. Think about the coverage criteria that are valuable. Furthermore, remember that white-box testing can lead to extra partitions of the input domain based on the code's structure. Finally, run the white-box tests. Again, if a test fails, improve the code so that the black-box as well as the white-box tests succeed.

Products: 1) white-box test, and 2) a running implementation of the function according to the functional specification.

Refactor. Often, the implementation can be improved by refactoring, for example to remove redundant code by adding an extra function or by simplifying a complex control structure. In case of refactoring, the black-box test remains applicable. The white-box tests, however, should often be adapted. In case a new function is added, the procedure is followed again for that function.

Products: 1) a refactored implementation, 2) if needed, an adapted white-box 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.

4.1.2 Aspect robustness

In the second iteration, we focus on the aspect robustness.

Analyze. Consider to weaken the precondition. This depends on, for example, the context in which the function will be used. Think about the result value of the function in case a problematic input value occurs, for example a special value as minus one. In case the function should throw an exception, the function's signature should be extended. Furthermore, the precondition and 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.

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 black-box test. In cases the precondition is weakened, the black-box test should be extended. That is, input values from extra partitions containing problematic values should be considered resulting in additional black-box tests. These tests should take into account the special return values as for example minus one or an exception thrown. Run the tests, which should normally fail because of the missing implementation parts, in order to check whether the test suite functions well.
Product: extended black-box test.

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) {
  var result;
  if (isOk(parameters)) {
    result = x(parameters);
  }
  else {
    result = exception_value;
  }
  return result;
}
```

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

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

Refactor. Finally, if possible, improve the implementation by refactoring. The black-box test remains applicable. The white-box tests, however, should often be adapted.

Products: 1) a refactored implementation, 2) if needed, an adapted white-box test, and 3) a running implementation of the function according to the functional specification.

5 The procedure applied

5.1 The Triangle problem

The Triangle problem (see for example [18]) is developed 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 indicate, in case of a triangle, the type of the triangle, as for example Equilateral, Isosceles, or Scalene. In case of the three numbers do not form an triangle, the output should be No triangle. Additionally, we require a, b, and c to be less or equal 200.

Remark. 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.

5.2 Aspect core functionality with a boolean as output

Analyze. The problem description gives us the precondition $a > 0 \wedge b > 0 \wedge c > 0$. We have chosen `isTri` as name for the function, which will have three arguments of type number. Mathematics states that for a proper triangle three inequalities must hold (given that a proper triangle has side lengths a, b, c that are all positive and excluding the degenerate case of a zero area):

Goal: Indicates whether three line segments a, b and c form a triangle
Signature: `function isTri(a:number, b:number, c:number): boolean`
Precondition: a, b, c all greater than zero
Returns: $a + b > c \wedge b + c > a \wedge c + a > b$

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  Investigates whether a,b and c form a triangle
 * @param        a  number representing the length of side a
 * @param        b  number representing the length of side b
 * @param        c  number representing the length of side c
 * @precondition a, b, c all greater than zero
 * @returns      true if a,b,c form a triangle, 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 black-box test. Based on the inequalities in precondition, we formulate seven function calls that either satisfy or dissatisfy these inequalities. Three of them test on boundaries. The corresponding black-box test cases are shown in Table 2.

Test case	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 2: Test cases for function `isTri`

We have chosen NodeUnit as our test tool in TypeScript, because this tool is very easy to use. The code for the seven test cases is:

```
let p = require('./index') // reference to file index.ts
                           // with the code of function isTri
import nodeunit = require('nodeunit');
exports.BlackBoxTest1 = function(test:nodeunit.Test) {
  test.expect(7);
  test.equal(p.isTri(3,4,5),true);
  test.equal(p.isTri(2,3,10),false); // because a+b<c
  test.equal(p.isTri(2,10,3),false); // because a+c<b
  test.equal(p.isTri(10,2,3),false); // because b+c<a
  test.equal(p.isTri(2,3,5),false);  // because a+b=c
  test.equal(p.isTri(2,5,3),false);  // because a+c=b
  test.equal(p.isTri(5,2,3),false);  // because b+c=a
  test.done();
}
```

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 white-box test. For the function so far, there are no additional white-box tests. The black-box tests and white-box 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 we apply the procedure so far to a second version of the Triangle problem, i.e. instead of output bool, the output will be the type of triangle.

5.3 Aspect core functionality with the type of triangle as output

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

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

The rules to determine the type of triangle are shown in table 3. 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 3: Rules for the type of triangle

Specify. We extend the specification and stub so far with the result type `TriangleType`.

```
/**
 * @function      isTri
 * @description   Investigates whether a,b and c form a triangle
 * @param         a  number representing the length of side a
 * @param         b  number representing the length of side b
 * @param         c  number representing the length of side c
 * @precondition a, b, c all greater than zero &&
 *               a + b > c && b + c > a && c + a > b
 * @returns       the type of the triangle (one out of TriangleType)
 */
export function triType(a: number, b: number, c: number): TriangleType {
  return NotATriangle;
}
```

Define black-box test. Based on the test cases from the previous phase and the rules for determining the triangle type, we can derive the test cases as shown in Table 4. Notice, that the first six test cases are the same as the test cases two until seven in Table 2.

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

Table 4: Black-box test cases for testing function `isTri`

The corresponding implementation of these test cases is as follows.

```
let p = require('./index') // reference to file index.ts
                           // with the code of function isTri
import nodeunit = require('nodeunit');
exports.BlackBoxTest1 = function(test:nodeunit.Test) {
  test.expect(7);
  test.equal(p.isTri(2,3,10),NotATriangle); // because a+b<c
  test.equal(p.isTri(2,10,3),NotATriangle); // because a+c<b
  test.equal(p.isTri(10,2,3),NotATriangle); // because b+c<a
  test.equal(p.isTri(2,3,5),NotATriangle);  // because a+b=c
  test.equal(p.isTri(2,5,3),NotATriangle);  // because a+c=b
  test.equal(p.isTri(5,2,3),NotATriangle);  // because b+c=a
  test.equal(p.isTri(5,5,5),Equilateral);   // because b+c=a
  test.equal(p.isTri(5,5,6),Isosceles);     // because b+c=a
  test.equal(p.isTri(5,2,4),Scalene);       // because b+c=a
  test.done();
}
```

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. We name the new function `triType`.

```
export function triType(a:number, b:number,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 white-box test. Aside from class NotATriangle, we have three other classes of triangles, namely Equilateral, Isosceles and Isosceles. The black-box tests already covers the boolean expressions in the if-parts completely for the types Equilateral and Scalene. For Isosceles, however, we there is only one test case, so we add two cases, `b===c` and `a===c`. Table 5 shows the resulting white-box tests.

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:

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

Table 5: Test cases for testing function `isTri`

```
/**
 * @function     hasThreeEqual
 * @description  Determines whether all arguments are 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  Determines whether two arguments are 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;
}
```

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

5.4 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: $0 < a, b, c \leq 200$. 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: Indicates whether three line segments a, b and c form a triangle
Signature: `function 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   Determines the type of triangle
 * @param         a number  representing the length of side a
 * @param         b number  representing the length of side b
 * @param         c number  representing 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 black-box test. For the black-box test, 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 $0 < a, b, c \leq 200$. The resulting black-box test is shown in Table 6.

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:

Test case	a	b	c	*Expected output*
1	2	3	10	NotATriangle, because $\neg(a+b>c)$
2	2	10	3	NotATriangle, because $\neg(a+c>b)$
3	10	2	3	NotATriangle, because $\neg(b+c>a)$
4	2	3	5	NotATriangle, because $\neg(a+b>c)$, boundary
5	2	5	3	NotATriangle, because $\neg(a+c>b)$, boundary
6	5	2	3	NotATriangle, because $\neg(b+c>a)$, boundary
7	5	5	5	Equilateral, because $a=b=c \wedge \neg NotATriangle$
8	5	5	6	Isosceles, because $a=b \wedge \neg NotATriangle$
9	6	5	5	Isosceles, because $b=c \wedge \neg NotATriangle$
10	5	6	5	Isosceles, because $a=c \wedge \neg NotATriangle$
11	5	6	7	Scalene, because $a \neq b \neq c \wedge \neg NotATriangle$
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	200.1	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	200.1	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	200.1	RangeError: $c > upper$, boundary

Table 6: Test cases for testing function `triType`

```
/**
 * @function      argsInRange
 * @description   Indicates whether 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 7 lists the test cases for this function. The implementation of function `argsInRange` can be elegant by

Test 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 7: Test cases for testing function `argsInRange`

use of the standard array function `every`:

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

In stead of modifying function `triType`, we write a new function `triAngle` that first checks the arguments. If the arguments are not correct it throws an exception, else it calls function `triType`. The implementation is as follows:

```
/**
* @function     triAngle
* @description  Determines the type of triangle
* @param        a number  representing the length of side a
* @param        b number  representing the length of side b
* @param        c number  representing 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 <= 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("One of the arguments is not in the range 0, "
                         + MAX);
  }
  return triType(a,b,c);
}
```

Define white-box test. We will now examine the code and pay special attention for overflow and the comparison of floats. Because the arguments are between 0 and 200, overflow will not occur. The problem left-over is the comparison of floats. If we simply use operator `===` we can have a problem due to internal representation and rounding errors. For example, `0.1+0.2 === 0.3` evaluates to false in TypeScript! In stead of using `===` we can investigate whether the absolute value of `0.1+0.2-0.3` is less than some value `EPS`. For `EPS` we can take 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 into account rounding errors
* @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. The final complete code becomes:

```ts
export const MAX = 200;

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

/**
* @function     isTri
* @description  Investigates whether a,b and c form a triangle
* @param        a number >0 representing the length of side a
* @param        b number >0 representing the length of side b
* @param        c number >0 representing the length of side c
* @precondition true
* @returns      true if a,b,c form a triangle , otherwise false
*/
export function isTri(a: number, b: number, c: number): boolean {
  return a + b > c && a + c > b && b + c > a;
}

/**
 * @function     hasThreeEqual
 * @description  Determines whether all arguments are equal
 * @param        a number
 * @param        b number
 * @param        c number
 * @precondition true
 * @returns      true if a=b=c otherwise false
 */
export function hasThreeEqual(a: number, b: number, c: number): boolean {
  return isEqual(a,b) && isEqual(b,c);
}
/**
 * @function     hasTwoEqual
 * @description  Determines whether two arguments are equal
 * @param        a number
 * @param        b number
 * @param        c number
 * @precondition true
 * returns       true if a=b or a=c or b=c
 */
export function hasTwoEqual(a: number, b: number, c: number): boolean {
  return isEqual(a,b) || isEqual(b,c) || isEqual(a,c);
}

/**
* @function     triType
* @description  Determines the type of triangle
* @param        a number  representing the length of side a
* @param        b number  representing the length of side b
* @param        c number  representing the length of side c
* @precondition true
* @returns      the type of the triangle (one out of TriangleType )
*/
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;
}
```

```
/**
* @function     triAngle
* @description  Determines the type of triangle
* @param        a number  representing the length of side a
* @param        b number  representing the length of side b
* @param        c number  representing the length of side c
* @precondition true
* @throws       RangeError if not(0<a,b,c<=200)
* @returns      the type of the triangle (one out of TriangleType )
*/
export function triAngle(a: number, b: number, c: number): TriangleType {
  let ar = [a, b, c];

  /**
   * @function     argsInRange
   * @description  Indicates whether for all members 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: 0 < el <= high else
       false
   *
   */
   function argsInRange(ar: Array<number>, low: number, high: number):
       boolean {
     return ar.every((el) => { return el > low && (el < high) || (isEqual
         (el,high)) });
   }

   if (!argsInRange(ar, 0, MAX)) {
     throw new RangeError("One of the arguments is not in the range 0, "
         + MAX);
   }
   return triType(a, b, c);
}

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

6 Related work

As far as we know, literature about procedural guidance for test development does not exist, especially not for test development as integrated part of incremental and iterative software development approaches.

6.1 Current techniques

There are many books about testing. For example, in the book *Pragmatic Unit Testing in Java with JUnit* [17], attention is paid to *structuring* a unit test in pre test, test and post test activities and 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, as, for example, checking boundary conditions, checking inverse relationships, and forcing error conditions.

Another example is the book *Software Testing: A Craftsman's Approach* [18] describing a number of main test techniques for unit testing, as for example Boundary Value Testing, Equivalence Class Testing, and Decision TableBased Testing. Each of these main techniques contains several special techniques, as for example Boundary Value Testing encloses 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, all merely points of attention as opposed to describing, for example, which techniques should be used in which order especially in relation to the incremental and iterative development approaches.

6.2 Procedural guidance for testing

Punnekkat et al. mention that work on improving the test design phase is new [28].

Bertolino [2] describes a roadmap of relevant challenges to be addressed. One of the problems she mentions is 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 always 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 fault, and will suffer from a saturation effect. We think that our approach supports these requirements. Applicable in a context of a first year course about software construction and testing, 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.

Bertolino [2] suggests that the test process should be engineered using testing patterns. Research should strive for producing engineered effective solutions that are easily integrated into development and do not require deep technical expertise. Our procedure can be considered as a process pattern.

6.3 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% [24]. 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 of how 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. The purpose of this report may be expressed as the ambition to extend Felleisen's methodology [11] with a step-by-step recipe for setting up unit tests.

6.4 Test First and Test Last approaches

Test First approaches, with TDD as its most widely known member, are software development practices in which test cases are written before the functionality is developed [8]. In a first step, the interface of say 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 its ability to encourage 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. [30] 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 'white-box testing', as opposed to 'black-box testing' that uses only the specification. 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 others are yet to be realized. Obviously this tends to blur the distinction between early and last testing.

Our approach includes Test First as well as Test Last properties. Especially the application of white-box analysis techniques incites to analyze the code structure often results in better code quality.

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 black-box tests, but *somewhere* certain white-box test should be added too! It is exactly this absence of guidance where our approach jumps into and give certain scaffolding.

6.5 Test Driven Development

Test Driven Development (TDD) is the most widely known member of the Test First approaches [1, 13]. We think that our approach enriches the TDD approach. A typical iterative and incremental TDD cycle consists of the following six steps [22]: 1) take a functionality and write a corresponding unit test, 2) run all tests and see the new one fails, 3) implement just enough code to pass the new test, 4) run the new test and all other unit test cases written for the code, 5) repeat the process until all tests are passed, 6) refactor the code and re-run all tests. The process is repeated from step one for each next functionality.

The TDD procedure gives no clues about the aspects to consider and the order to do that. The procedure uses the term 'functionality', but this term can enclose, for example, core functionally as well as robustness.

It strikes that there is no specification at the start of the process. Instead, the unit test substitutes the specification and is extended in an iterative and incremental way. In our opinion, however, analyzing and formulating specifications is an important activity, because it is known that undetected faults have mostly to do with incomplete specifications and missing logic [16].

Furthermore, the procedure leaves aside when to specify black-box as well as white-box tests, i.e. the procedure talks only about 'a corresponding unit test'. What is more, it seems that white-box tests are never considered explicitly, because tests are only written in the first step of each cycle. It is exactly these issues that are treated in our approach.

6.6 Related issues

Several papers, e.g. the one by Falessi and Kruchten [9], discuss the concept of *technical debt*. This is defined [25] 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 [27] 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. [28] 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 are the bounderies more or less visible at different levels in the software, i.e. the specification and implementation. These boundaries should always be tested but are often missed in practice. A solution is not given to this problem. We think that our approach gives guidance in solving this problem, by distinguishing boundaries on the input domain and in the implementation concerning different aspects. Secondly, they observed that students are in general better in defining valid input than invalid input data, and as such use mostly obvious test cases and often do not vary in inputs resulting in low coverage and less robust systems. Our approach emphasizes the aspect robustness, and give guidance how to do that, namely by a step-by-step approach in which the function's precondition is relaxed concerning the specifications as well as the implementation of the function. Based on the analysis, Punnekkat et al. define a cyclic meta process to eliminate mistakes made in test case construction, consisting of the following steps: 1) select test cases and test specifications, 2) perform an expert review on test design, 3) define mistakes categories, 4) measure mistakes, 5) propose a new test template, 6) teach the template and common mistakes, and finally 7) measure the efficiency. Our approach is concrete process, i.e. a template, on the level of procedural guidance consisting of fine grained steps.

7 Conclusions and future work

As we explained, 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. Applicable in a context of a first year course about software construction and testing, we developed a stepwise procedure enclosing and ordering a selected number of known test methods integrated in a development process of analysis, specification, implementation, testing, and refactoring.

Our approach includes the benefits of Test First as well as Test Late appraches. Especially the application of white-box analysis techniques incites to analyze the code structure often results in better code quality.

We think that our approach solved the problem of how the overwhelming number of test methods and test criteria may be combined most efficiently in an integrated development process [2]. It is exactly this absence of guidance where our approach provides scaffolding. Our procedure can be considered as a process pattern [2].

We think that our approach gives guidance in solving this problem, by distinguishing boundaries on the input domain and in the implementation concerning different aspects. Our approach emphasizes the aspect robustness, and give guidance how to do that, namely by a step-by-step approach in which the function's precondition is relaxed concerning the specifications as well as the implementation of the function.

We think that the procedure is generic, i.e. that it will support other aspects as for example performance as well. This is planned as future work.

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

References

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

[2] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[5] Henrik Bærbak Christensen. Systematic testing should not be a topic in the computer science curriculum! *SIGCSE Bull.*, 35(3):7–10, June 2003.

[6] Arie van Deursen and Leon Moonen. The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76, 2002.

[7] Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bull.*, 36(1):26–30, March 2004.

[8] Hakan Erdogmus, Forrest Shull, Burak Turhan, Lucas Layman, Grigori Melnik, and Madeline Diep. What do we know about test-driven development? *IEEE Software*, 27:16–19, 2010.

[9] Davide Falessi and Philippe Kruchten. Five reasons for including technical debt in the software engineering curriculum. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 28:1–28:4, New York, NY, USA, 2015. ACM.

[10] Sheikh Umar Farooq and Smk Quadri. 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, pages 72–77, New York, NY, USA, 2013. ACM.

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

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

[13] Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. 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):597–614, 2017.

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

[15] Hisham Haddad. Post-graduate assessment of cs students: Experience and position paper. *J. Comput. Sci. Coll.*, 18(2):189–197, December 2002.

[16] Hadi Hemmati. How effective are code coverage criteria? In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015.

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

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

[19] Paul A Kirschner, John Sweller, and Richard E Clark. 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):75–86, 2006.

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

[21] C. Larman. *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, 2009.

[22] Roberto Latorre. Effects of developer experience on learning and applying unit test-driven development. *IEEE Transactions on Software Engineering*, 40(4):381–395, 2014.

[23] K. Rustan M. Leino. Constructing a program with exceptions. *Information Processing Letters*, 53(3):159 – 163, 1995.

[24] Otvio Augusto Lazzarini Lemos, Fbio Fagundes Silveira, Fabiano Cutigi Ferrari, and Alessandro Garcia. The impact of software testing education on code reliability: An empirical assessment. *Journal of Systems and Software*, 2017.

[25] Steve McConnell. Managing technical debt. Retrieved December 22, 2017 from `https://www.sei.cmu.edu/community/td2013/program/upload/technicaldebt-icse.pdf`, 2013.

[26] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR: Upper Saddle River, NJ, U.S.A., 1997.

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

[28] Sasikumar Punnekkat, Sigrid Eldh, and Hans Hansson. Analysis of mistakes as a method to improve test case design. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST 2011)*, 00:70–79, 2011.

[29] Alex Radermacher, Gursimran Walia, and Dean Knudson. Investigating the skill gap between graduating students and industry expectations. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 291–300, New York, NY, USA, 2014. ACM.

[30] Giuseppe Scanniello, Simone Romano, Davide Fucci, Burak Turhan, and Natalia Juristo. 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, pages 1422–1427, New York, NY, USA, 2016. ACM.

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

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

[33] Qian Yang, J. Jenny Li, and David M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.