Assignment of master's thesis

Title:	Extension of the Cpputest framework and its usage in the ETCS simulator testing
Student:	Bc. Kateřina Kasalická
Supervisor:	Ing. Jiří Chludil
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of winter semester 2022/2023

Instructions

The goal of this thesis is to extend the testing and mocking framework for C/C++ Cpputest with focus on automation of mocks creation and usage. Thereafter the implementation shall be used to test selected modules of the ETCS (European Train Control System) simulator project of the Faculty of Transport of CTU partly developed at FIT CTU.

1. Analyse and describe the test process and its objectives in the software development process.
2. Analyse existing libraries and frameworks for software testing in C++.
3. Design extension of the framework Cpputest. Focus on automation of mocks creation, their usage and data handling. Based on the previous analysis, suggest further possible improvements of the Cpputest.
4. Implement a prototype of the Cpputest framework extension designed in the previous step.
5. Use the implementation from the previous step to test selected modules (DMI, EVC, RBC and Braking curve) of the ETCS simulator project. Describe your approach and report the results.

Electronically approved by Ing. Michal Valenta, Ph.D. on 4 June 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Extension of the CppUTest framework and its usage in the ETCS simulator testing

Bc. Kateřina Kasalická

Department of Software Engineering
Supervisor: Ing. Jiří Chludil

January 5, 2022

Acknowledgements

I would like to thank my supervisor Jiří Chludil for his advice and guidance through the whole process of this thesis creation and enabling me to pursue the topic I wished. I also thank Martin Šmíd for giving me professional confidence and Martin Šmíd for moral and psychic support.

Declaration

In Prague on January 5, 2022 …………………

Citation of this thesis

Kasalická, Kateřina. *Extension of the CppUTest framework and its usage in the ETCS simulator testing.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce se předně zabývá návrhem a implementací rozšíření existujícího testovacího C++ frameworku CppUTest. Framework CppUTest již obsahuje střední podporu pro práci s mocky, nicméně tato práce navrhuje rozšíření již existující funkcionality zaměřenou na rozšíření a automatizaci využití mocků.

Práce se nejdříve zabývá analýzou a shrnutím teorie softwarového testování a testovacích technik. Následně se věnuje analýze existujících C++ testovacích frameworků, získaná data jsou pak využita pro formulaci a návrh nových funckcionalit pro framework CppUTest.

Posledním úkolem této práce je vyzkoušet využití vyvinutého rozšíření implementováním sady testů zaměřené na projekt ETCS simulátoru.

Klíčová slova CppUtest, testování, ETCS, mock

Abstract

This thesis is mainly concerned with design and implementation of extension of the existing C++ testing framework CppUTest. The CppUTest already includes moderate support for work with mocks, however this work suggests extension of this support focused on s wider and more automatic usage of the mocks.

First task of this thesis is to analyze and summarize software testing principles and techniques. The following task is to analyze existing C++ testing frameworks, the collected data from this analysis are then used to suggest and form new features for the CppUTest.

The last task of this thesis is to examine usage of the developed extension via implementing a set of tests for the ETCS simulator project.

Keywords CppUtest, testing, ETCS, mock

Contents

List of Figures

List of Tables

List of source codes

Introduction

Software development is a complex process, often accompanied by many challenges and drawbacks for software developers. One of the main aims of the software development theories and recommended methods is to ensure the final product quality. Especially for larger projects, it has been proved difficult to ensure the developed software complies with the expected behavior and the customer's requirements.

Software testing is, or at least should be, inseparable part of a software development process, since it is a useful tool to ensure the product quality. Many aspects of software testing still remain underrated by the software development participants, since there is more to software testing than just implementing test cases. One of the motivations of this work is to describe the role of software testing, its goals and procedures, and demonstrate usage of the provided theory in practice.

CppUTest is a C++ testing framework that provides many useful testing tools that guide the tester and save their time. However, not all its features are that practical and time saving as they could be. Especially the mock support is currently limited and leads to unnecessary and tedious test development.

Goals of the Thesis

The main goal of this thesis is to extend the existing framework CppUTest so it would provide better support for mocks, specifically by adding support for better mock desired behavior setting and automation of the mocks creation. One of the aims of this thesis is to create an analysis of the other existing competitive testing frameworks as inspiration for the extension requirements and design decisions.

Another objective of this work is to examine the usage of the extension via implementing tests focused on the ETCS simulator project. The aim is also to summarize a testing theory and used techniques and demonstrate their usage in practice.

Chapter 1

Software testing

The goal of this chapter is to describe the basics of software testing and to demonstrate to the reader the different aspects of testing that shall be considered by a tester. This chapter does not aim to describe every aspect of the testing process, since software testing is very extensive subject and this thesis does not aspire to cover every detail of this process, but rather explain the basic concepts used throughout the following chapters.

1.1 Goals of software testing

Software testing is a process that aims to verify and validate that the tested software works correctly and as expected according to the provided requirements. This process contains many different activities — planning of the tests, designing and implementing tests, analyzing, reporting test progress and results, and assessing quality of the test object. [1] [2]

The objectives of a software testing usually contain the following:

- Evaluate work products such as requirements, user stories, design, and code to prevent defects.
- Verify whether the developed software fulfills the provided requirements.
- Check whether the developed software is complete and works as expected by users or customers.
- Assure sufficient quality of the product.
- Detect defects and errors and prevent software malfunctioning.
- Verify the test object compliance with contractual, legal, or regulatory requirements or standards. [2] [3] [1]

In general it is a tool to answer questions about the tested product. The process of testing leads to answering questions concerning quality of the product and also questions of the customers and users — whether the product meets the customer's requirements, if the developed software works correctly and does not contain selected defects, to name but a few.

1.2 Basic concepts and terms

Software testing field uses its specific concepts and terminology, The purpose of this section is not only to explain the basic concepts, but to also show several aspects of software testing process:

Verification is an activity that aims to confirm whether the concerned object works correctly. In other words, this activity tries to answer the question "Are we building the system right?" [4]. The IEEE standard [5] defines verification as "Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled."

Validation is an activity that focuses on evaluating whether the concerned object meets its intended use. In other words, this activity tries to answer the question "Are we building the right system?" [4]. The IEEE standard [5] defines verification as "Confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled."

Test object / Subject under testing is the tested object. Test object can be a whole system, a module, or just a function.

Test case can be easily defined as a pair of some input and expected results. An input can include a set of inputs and executions conditions, expected results can represent an output or outcome (for example change of a state). Test case can be also described as a sequence of steps and a set of requirements that should be fulfilled by the test object. [6] [1]

Test driver is defined by the IEEE standard [6] as "a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results". In other words, it executes the test cases, handles all that is necessary for its execution and checks the test object behavior towards the expectations.

Error is a mistake that can be made by a person. The IEEE standard [6] defines error as "a human action that produces an incorrect result, such as software containing a fault".

Defect is a result of a human error, for example a bug or a fault in the software code. [2]

Failure is an external behavior of the software that does not conform to the expected behavior. Failure is usually a consequence of a defect in the software, but it can be also caused by a defect in the definition of expectations. [1]

Coverage is defined by the IEEE standard [6] as ”the degree to which a given test or set of tests addresses all specified requirements for a given system or component”. In other words, it measures how much the certain test object is actually covered by the executed tests. There exist several methods and definitions for measuring the coverage, some of them will be described in the following sections).

1.3 Functional and non-functional testing

The requirements that are put on the developed software can be divided in two categories – functional and non-functional. Based on which of the two categories is tested, the tests can be also split accordingly.

Functional testing aims to evaluate the software against the functional requirements. The functional requirements specify ”what” the software should do. Such requirements can be related to communication systems, modules, logging, security, features, user interface, to name but a few. In other words, functional testing involves tests that focus on functions the software should perform. For example, a functional test can verify output value of a function. [1] [2]

Non-functional testing evaluates the software against the non-functional requirements. The non-functional requirements specify ”how well” the software should perform. To name a few, such requirements can be related to usability, reliability, performance or maintainability. In other words, non-functional testing focuses on how the software behaves. For example, a non-functional test can measure the time the software takes to perform some specific action. [2]

1.4 Dynamic and static testing

When considering how to test some aspect (functional or non-functional) of the software, there are two possible approaches to choose from – dynamic or static testing approach. Each of the approaches has some pros and cons and it is up to the tester to choose the more suitable for the considered requirement or just an aspect of the requirement.

Static testing does not rely on actual execution of code, but rather on a variety of reviews. Static testing involves manual examination of the code or evaluation of the code driven by a tool. [1] [2]

The static testing or static analysis has many pros over the dynamic testing. The first one is that while dynamic testing detects failures that need further investigation to find the actual defects, the static testing finds directly the defects in the implementation. Another distinction is that in many cases, it might be easier to perform static analysis than to implement very complex dynamic tests. For example dynamic testing of memory handling or security vulnerabilities might be very complex to implement, while analogous static analysis can be performed much easier. [2]

The main disadvantage of static testing is that it has to be performed manually, especially in big projects that could be very time inefficient. For example if the development is carried through many cycles and requires regression testing in each cycle, the static tests should be reevaluated for each cycle or implementation change.

Dynamic testing is execution based, meaning the tested implementation is executed in controlled test environment (described in 1.10). This is achieved by implementing a test case that is then automatically executed and evaluated.

While it might be more complex to implement a dynamic test, the implemented test can be then executed and reevaluated automatically without any additional effort and is usually more time efficient in the longer perspective.

In practice, the dynamic testing is preferred for its better time efficiency, however in some cases the static approach might be more suitable. Usually both approaches are combined to achieve higher coverage and reliability.

1.5 Black-box and white-box testing

When designing a test case, several information sources, such as specification, source code etc. need to be considered. And based on the information that is considered, two main testing concepts are black-box and white-box testing. Black-box testing techniques are also often called functional testing techniques and white-box techniques can be also called structural testing techniques. [1]

Black-box testing is considered only with the externally visible behavior of the tested unit. In this case, the unit can be a method, class, module, or whole system. The inputs for black-box testing are just requirements and specifications, no internal knowledge of the tested unit shall be used (for example the source code). Black-box testing approach is used by the testers, since they have almost no knowledge of the internal implementation of the tested unit. [1] [7]

White-box testing uses knowledge of the internal aspects of the feature, to name a few, structure, internal paths and implementation. The white-box testing is usually performed by the programmers using source code reviews and unit tests implementation (explained in 1.9.1). [1] [7]

1.6 Black-box testing techniques

As already stated in 1.5, black-box testing is mainly based on the requirements, therefore the black-box testing techniques mainly focus on selection of the test data for test cases and coverage of possible usage scenarios.

1.6.1 Equivalence class partitioning

As explained in 1.2, test case can be defined as a pair of input(s) and output(s). When choosing the appropriate input, there are usually countless options to choose from and unfortunately it is not possible to test them all due to resource and time limitations. ”Equivalence class testing is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage” [7].

Equivalence class partitioning divides data into equivalence classes (also called partitions), where all the inputs of the equivalence class are expected to be processed alike. Typically, the input values can be divided into valid and invalid values, but such partitioning usually needs further distension to obtain the final equivalence classes. [2]

For example, in a programming class, only students who have obtained at least 30 points (from the maximum of 70) for programming assignments can pass, and students that overreach the maximum by gaining bonus points are suggested to enroll in a competitive programming class (the maximum of possible points including the bonuses is 99 points). If the test object was a software determining whether the students passed and what further actions should be made, the possible equivalence classes and their representative values could be the following (considering only integers for simplicity):

- <0 - 29 points> – students that shall not pass with representative = 25 points,
- <30 - 69 points> – students that shall pass with representative = 42 points,
- <70 - 99 points> – students that shall pass and obtain the competitive class recommendation with representative = 77 points,
- <$-\infty$ - (-1) points> – invalid values with representative = -13 points,
- <100 - $+\infty$ points> – invalid values with representative = 7581.

1.6.2 Boundary value analysis

Boundary value analysis is based on the equivalence class partitioning described in 1.6.1. Whereas equivalence class partitioning aims to select a candidate from each equivalence class, boundary value analysis focuses on boundaries of the equivalence classes.

This technique can be applied only to data sets where boundaries can be defined — numbers. When the boundary value is identified, two test data sets should be constructed — one just above the boundary and one just below the boundary. This testing technique is quite effective in finding defects, since programmers often tend not to treat the boundaries values correctly. [2] [1]

When applying this technique on the example of passing programming students from the section 1.6.1, the following test values would be selected:

- <0 - 29 points> – selected boundary values would be (-1) (just below the boundary), 0 (just above the boundary), and analogously 29 and 30; for the intervals <30 - 69 points> and <70 - 99 points> boundaries would be selected accordingly.
- <$-\infty$ - (-1) points> – values (-1) and 0 would be chosen for the second boundary (this values were already covered in the previous step), but the second boundary is little more tricky. In case the input would be of type *INT*, the best approach would be to select (-*MAXINT*) and (-*MAXINT* - 1).

1.6.3 Decision table testing

Both previously described techniques (Boundary value analysis and Equivalence class partitioning) considered each input separately. But in practice, the test object is far more complex and test cases have often various different inputs and outputs. In such case, all possible combinations of inputs and outputs shall be tested. To detect the individual possible inputs, the Equivalence class partitioning and Boundary value analysis techniques might be used. [1] [2] [7]

The decision table in 1.1 is then defined as follows:

- In the left column, individual inputs (called *conditions* in this case) are defined.
- Also in the left column, under the conditions, individual outputs (called *actions* in this case) are defined.
- On the right side of the first column with conditions, columns with individual test cases (called *rules* in this case) where each represents the possible combination of conditions and their corresponding actions, are defined. [7] [1]

	Rule 1	Rule 2	...	Rule N
Conditions				
Condition 1	$value_{11}$	$value_{12}$	...	$value_{1N}$
Condition 2	$value_{21}$	$value_{22}$	...	$value_{2N}$
...	...	...	...	...
Condition M	$value_{M1}$	$value_{M2}$	...	$value_{MN}$
Actions				
Action 1	$action_{11}$	$action_{12}$	...	$action_{1N}$
Action 2	$action_{21}$	$action_{22}$	...	$action_{2N}$
...	...	...	...	...
Action K	$action_{K1}$	$action_{K2}$	...	$action_{KN}$

Table 1.1: Decision table definition [7]

1.6.4 State transition testing

When considering a test case, the output may be dependent on the previous history of the test object. This behavior can be often defined by states of the test object. A state transition diagram is a tool to record the states of the test object, actions that induce transitions between the defined states, and possible corresponding actions. This technique allows the tester to construct the test cases to cover either all typical sequences of states, all possible transitions, all invalid transitions, or all states. [2] [7]

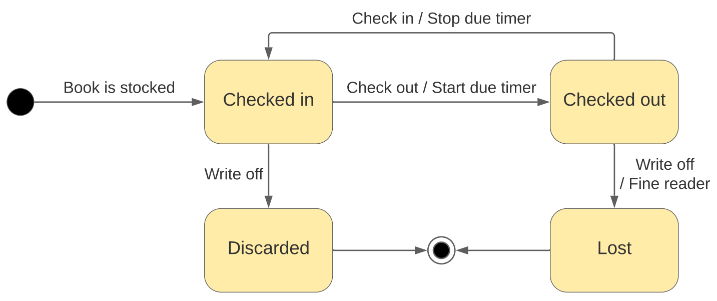

Figure 1.1: Example state transition diagram

To demonstrate the usage of the state transition diagram, example from 1.1 regarding states of a book from library will be considered. When a book is stocked into the library, its initial state is "Checked in". Then it can be either checked out by the reader or written off (after some time period) due to its bad condition or being outdated. When the book is "checked out" it can be checked in by the reader, in opposite case it is considered lost.

Instead of recording the states and transitions to a diagram, a table might be also used. This technique might be more convenient for complicated sys-

Current state	Event	Action	Next state
null	Stock in	–	Checked in
null	Check out	–	null
null	Check in	–	null
null	Write off (bad condition)	–	null
null	Write off (due date)	–	null
Checked in	Stock in	–	null
Checked in	Check out	Start due timer	Checked out
Checked in	Check in	–	null
Checked in	Write off (bad condition)	–	Discarded
Checked in	Write off (due date)	–	null
Checked out	Stock in	–	null
Checked out	Check out	–	null
Checked out	Check in	Stop due timer	Checked in
Checked out	Write off (bad condition)	–	null
Checked out	Write off (due date)	Fine reader	Lost
Discarded	Stock in	–	null
Discarded	Check out	–	null
Discarded	Check in	–	null
Discarded	Write off (bad condition)	–	null
Discarded	Write off (due date)	–	null
Lost	Stock in	–	null
Lost	Check out	–	null
Lost	Check in	–	null
Lost	Write off (bad condition)	–	null
Lost	Write off (due date)	–	null

Table 1.2: Example state transition table

tems with high number of states and transitions. The state transition table has columns *Current state*, *Event*, *Action* and *Next state*, as shown in 1.2 corresponding to the previous example in 1.1. [7]

1.6.5 Use Case testing

Use case is mainly a tool used by analysts and developers to describe usage of the system from the user's point of view. A use case is a scenario with individual steps (performed by actors and the system) leading to a goal.

The use cases can be also used by testers to derive test cases. Such test cases then follow the individual steps, therefore test each of the individual functionalities and simulate real life usage of the system in one complex test, while previously described techniques usually focused on individual separate functionalities only. [2] [7]

1.7 White-box testing techniques

As already suggested, white-box testing (also called structural testing) uses knowledge of internal implementation details of the test object. So instead of deriving the test cases from the requirements, the code itself, its constraints and data attributes are considered.

1.7.1 Control flow testing

As the name control flow suggests, the goal of this technique is to test all possible execution paths of the code base. This can be achieved by creating a control flow diagram (also often called graph), where each node represents a set of program statements. The nodes are connected by edges if the set of statements of the second node may be executed just after the statements of the first node. There are five types of nodes, including decision, merge, statement, entry, and exit node. [8]

Test cases are then derived from the constructed diagram, where a test case is a complete path in the diagram. The aim of this technique is to cover as many possible paths by the test cases as possible. For this technique, six different coverage levels are defined with corresponding criteria:

1. **Statement coverage criteria** (or node coverage) – ”Every statement in the program has been executed at least once.” [8]

2. **Decision coverage** (or edge coverage) – ”Every statement in the program has been executed at least once and every decision in the program has taken all possible outcomes at least once.” [8]

3. **Condition coverage** – ”Every statement in the program has been executed at least once, and every condition in each decision has taken all possible outcomes at least once.” [8]. On the first sight, this condition might look very much like the previous one. First it must be considered that decision in the program (i.e. an *if* statement) can consist of many different statements. Hence while the previous condition states that every decision outcome shall be considered, in this case, all outcomes of each individual condition of the statement must be evaluated. [7]

4. **Decision/Condition coverage** – ”Every statement in the program has been executed at least once, every decision in the program has taken all possible outcomes at least once, and every condition in each decision has taken all possible outcomes at least once.” [8]

5. **Multiple Condition coverage** – ”Every statement in the program has been executed at least once, all possible combination of condition outcomes in each decision has been invoked at least once.” [8]. Where for n conditions, there are 2^n combinations of condition outcomes. [8]

6. **Path coverage** – ”Every complete path in the program has been executed at least once.” [8]

1.7.2 Data flow testing

As explained in 1.7.1, control flow testing is a technique to derive test cases from the flow of the control in the code base. Analogously data flow testing focuses on flow of the data in the implementation. This is again achieved by creating a data flow graph that uses similar principles as the control flow graph. The graph tracks different occurrences of each variable, the key occurrences to identify anomalies are definition and reference. [7] [9]

The goal of the data flow testing technique is to detect anomalies. Based on the occurrence types of the investigated variable, several anomalies can be detected:

- Defined and then defined again.
- Undefined but referenced.
- Defined but not referenced. [1]

1.8 Experience based testing techniques

Previously described testing techniques were rather systematic and clearly prescribed, experience based testing is very different approach. It relies more on the skill, intuition and experience in the field of the tester. These techniques can be able to identify test cases and defects that the previous techniques did not due to their limitations. [2]

1.8.1 Exploratory testing

As the name suggests, exploratory testing is useful to explore the behavior of the test object, especially when there is not available sufficient specification or other description. This technique can be useful in following cases:

- When the test object is new for the tester, exploratory testing can be used to get familiar with its behavior.
- When there is no sufficient specification and the test must be performed as soon as possible. Such situation can occur in the early development of a feature, in which case there is not a specification at all, or it is in progress and does not define the behavior in enough detail.
- Once a defect is detected (using any testing technique), exploratory testing can be used to make further investigation of the defect traits and its outcomes, e.g. to provide more detailed feedback to the developer.

- In some (usually invalid) cases the behavior of the test object can be undefined, but it might be still useful to investigate such case. For example if the system crashes due to an invalid handling by the user, this technique can be useful to determine how exactly the system crashes. [2] [7]

Exploratory testing is sometimes performed through sessions within a defined time box. During this session, the tests are gradually derived from the previously executed tests and their results. This is usually conducted with the help of a test charter containing test objectives as a guidance for the tester. [2] [7]

1.8.2 Error guessing

Error guessing testing is a technique that heavily relies on the experience and knowledge of the tester. The aim of this technique is to create a list of possible defects, errors and failures and design test cases that will investigate these failures and their origin. [2]

The list of the possible defects can be assessed using the experience of the tester, however there are critical areas that tend to have more defects:

- Segments of code with high cyclomatic complexity.
- The code that has been recently added or modified.
- Code blocks that contained higher number of defects in the past.
- Parts of the test object using new technology that has not been used in the project before.
- Features with unusually vague specification.
- Portions of code developed by novice developers or programmers more prone to creating defects.
- Component that was developed by higher number of developers, since there is higher chance of misunderstanding among the developers. [1]

1.9 Levels of testing

The software development, design, and requirements specification can be divided into several levels (or layers). These levels could be defined as different views on the software project.

1. On the first level (business requirements), business requirements for the product are gathered.

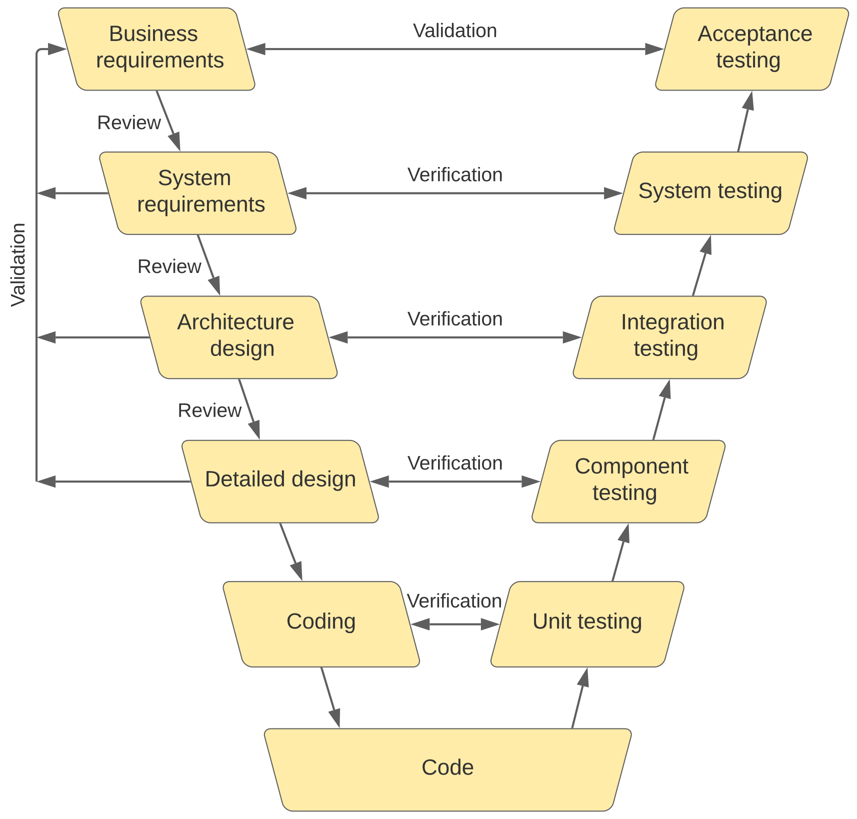

Figure 1.2: V-Model

2. On the second level (system requirements), requirements for the whole system that is being developed are specified, these requirements deal more with the technical details than the first level.

3. The third level (architecture design) focuses on the architecture of the software, on this level individual modules and their cooperation is described.

4. The following level (detailed design) specifies design of the software in more detail, typically describes complete functionality of the individual modules separately.

5. The last level (coding) is the process of writing the actual code base, it includes public APIs[1], methods definitions and declarations, to name but a few.

[1]Application programming interfaces

Mentioned levels are displayed in the V-Model 1.2. The V-Model is often depicted in different versions that usually include only four levels, unlike the V-Model shown here. That is caused by joining some of the levels, most often the *System requirements* and *Architecture design* level. This more detailed view enables to describe the software testing process in more detail and show all aspects of tester's and validator's responsibilities.

Each of the development levels has its corresponding testing level, as shown in the V-Model 1.2. The purpose of the individual testing levels is to verify (or validate) that the requirements specified on the corresponding development level have been fulfilled by the tested code. Each testing level has its characteristic attributes – specific objectives, test object, test base, test environment, characteristic defects and failures, specific approaches and responsibilities. The individual test levels will be discussed separately in the following subsections. [2] [1]

1.9.1 Unit testing

"Unit testing refers to testing program units in isolation" [1]. The definitions of the scope of a unit differ, but it is often understood as methods, functions, or procedures. In some cases, a class or even a whole module is considered a unit. The unit testing is also sometimes referred to as a module or component testing. [1]

In this thesis, the main considered differences between the unit testing and component testing are the purpose and the author of the testing. In practice, unit tests are often implemented and performed by the programmer of the tested functionality. The purpose of the unit testing is to detect defects in the early stage of the development. It is mainly a tool for the programmer to ensure that the code has satisfactory quality, since the programmer is responsible for the quality of the code. Therefore the unit tests and their results are not used in the validation process.

Since the unit tests are performed by the programmer of the tested code, the approach has a few different features from component testing that is performed by a tester. In contrast to the component testing, unit testing is a white box testing, since the programmer has knowledge of the internal structure of the tested code. Hence detection of the defects is usually faster than during component testing. [10]

1.9.2 Component testing

As already stated in 1.9.1, component and unit testing are often considered to be the same, since both levels refer to testing program units in isolation [1]. But the scope of the component testing slightly differs from the unit testing. While the goal of the unit testing is usually to test separate methods, functions, or procedures and focuses more on testing implementation details, the component testing aims to test, verify, and validate functionality of whole components or modules. [10]

A component ”refers to a separate module or programming object that works independently of the other components in a system while maintaining communication with the entire system” [11]. To give an example, component can be an API, a database, a process, or a daemon. The components are usually tested in isolation, in characteristic test environment often using mocks and stubs [2].

The component testing is performed by a tester, therefore it is a black box testing. Unlike the unit testing, results of component testing are used in the validation process to evaluate whether the test object fulfills its requirements.

1.9.3 Integration testing

”Integration testing is the level of test done to ensure that the various components of a system interact and pass data correctly among one another and function cohesively.” [12]. Unlike in unit and component testing, integration testing does not test just parts of the system in isolation, but tests that the components or modules have been integrated correctly. In other words, the purpose of integration testing is to put the developed components/modules together and find possible defects related to compatibility of the components.

Especially in moderate to large projects with tens to even hundreds of programmers cooperating together, different components are developed by various programmers and tested in isolation on component testing level by various testers. The coordination of alike project can be very complicated and defects such as incompatible interfaces can be easily made. The goal of the integration testing level is to reveal these defects and ensure the integrated components are compatible. [1]

1.9.4 System testing

”The objective of system-level testing, also called system testing, is to establish whether an implementation conforms to the requirements specified by the customers” [1]. While the integration testing focuses on testing compatibility of a system components and their collaboration, system testing focuses on the behavior of a system from the outside. It aims to verify that requirements for a system, functional and non-functional, have been met.

In practice, the system testing and integration testing are often implemented and evaluated together, since their test cases can be very similar. The test environment is often the same and on both levels already fully integrated system is tested. The only difference is that the integration testing level focuses on the internal collaboration of the components, while the system testing level goal is to verify behavior of a system from the outside.

1.9.5 Acceptance testing

The goal of acceptance testing is to validate that the developed product fulfills customer's requirements. It can be very similar to system testing, since it focuses on the same criteria, but it is performed by the customer or a third party entrusted by the customer. The main goal of the acceptance testing is not to detect defects, but rather build the customer's confidence in the product that it meets the required functionality, standards and other requirements. [2] [7]

1.10 Test environment

The IEEE standard [6] defines the test environment (also called *test bed*) as "hardware and software configuration necessary to conduct the test case". In other words, when considering a test case, certain conditions need to be met to execute the test case in such environment that the test object behaves the same as in the production environment. The test environment (depicted in 1.3) can include hardware configuration, operating system, third party software, network setting, test data, and database server to name but a few. [1] [13]

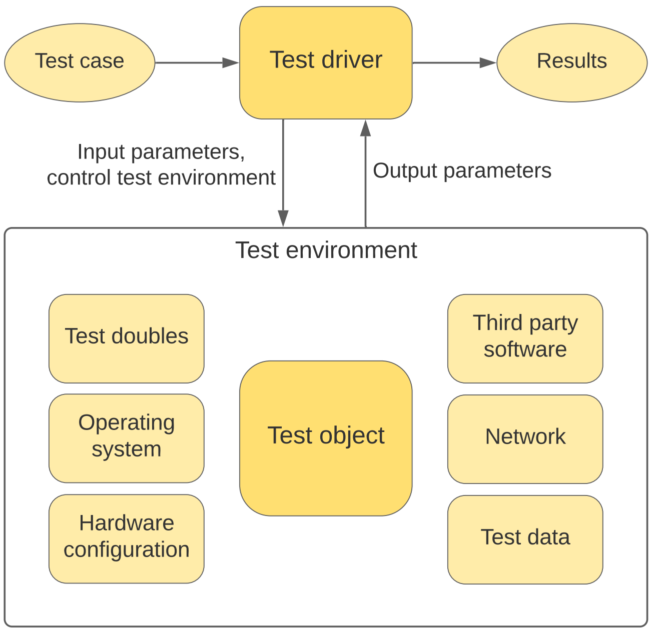

Figure 1.3: Test environment

The tester must ensure that the test environment complies with the production environment and does not influence behavior of the test object in any unexpected way. But at the same time, it enables the tester to have a full control of the environment through the test driver and can be used to the tester's advantage. This is especially utilized on the unit and component testing level by using test doubles (described in detail in 1.11). [14]

1.11 Test doubles

As suggested in 1.10, tester can simulate the production environment by creating test double(s). Test double replaces a real object (i.e. component, library, database etc.), simulates its behavior and at the same time checks the behavior of the test object. The test object calls are then redirected to the test double instead of the real object as depicted in 1.4. The test double has usually quite simple implementation and is far less complex than the real object. [15] [16] There are a few possible reasons for using test doubles:

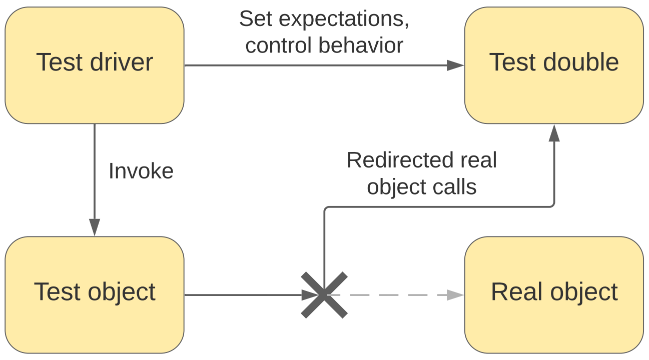
Test driver
Set expectations, control behavior
Test double
Invoke
Redirected real object calls
Test object
Real object

Figure 1.4: Test double usage

- Some components needed for executing the test object are not implemented yet. Using test doubles instead of the non-implemented components enables to execute the tests earlier, since there is no need to wait for the implementation.

- Test doubles allow the tester to check communication between the components that might not be accessible for the tester in production (or similar) environment.

- Using test doubles can reduce tests execution time. For example if a certain component takes long time to respond to certain calls, it can be replaced by faster responding test double.

- Some components might have side effects that are not desired in testing process. For example, an email shall be sent when an event is triggered, but it would be inconvenient to actually send the email every time the corresponding test is executed. So a test double is created, which registers the call requesting to send the email, but does not actually send the email.

- Some dependency of the test object has complex setup that cannot be easily satisfied, i.e. it needs special hardware or plenty of configuration data and other dependencies. [17] [18]

This approach is used mainly on the component and unit test level, since it enables the tester to have full control of the test environment. Otherwise it would be often impossible to investigate communication between individual components or systems. On the other hand, whenever a public API of the real object changes, the test double must be updated accordingly.

There are a few variations of the test double – test stub, test spy, mock object and fake object. The term test double is often incorrectly confused with just a subset term mock or stub. [16] These are the test double variations:

Test Stub – "is an object that holds predefined data and uses it to answer calls during tests." [15]. Stub is an object that includes the same API as the real object, therefore the test object calls can be redirected to it. The stub then only responds according to predefined data, hard-coded or configured from the test driver. [19]

Test Spy – returns predefined data the same way the stub does. But additionally it saves the calls from the test object and their parameters. The saved data are later verified by the test driver. [20]

Mock Object – has, on the first sight, very similar behavior as the test spy. It returns predefined data and registers calls from the test object. But unlike the test spy, mock object directly verifies the calls and their parameters right after they were made by the test object, therefore mock object does not return the data to the test driver. The call expectations (and return data) are configured from the test driver. [21]

Fake Object – is very different from the previous variations. The goal of the Fake object is to imitate the behavior of the real object, but in a simpler way. This is mainly used when the real object is not implemented yet, or its responses are too slow. Unlike previous variations, fake object is not controlled by the test driver at all, therefore its return values are not predefined and the test object calls are not verified in any way from this test double. [22]

Chapter 2

Analysis of C++ testing frameworks

When developing dynamic tests, the best practice is to use a testing framework. Of course, it is possible to implement automatic tests from scratch. But testing software is not just about implementing scenarios and simple checks, when managing bigger code base (both implementation and tests), the testing should have some structure and procedures, which is easier to achieve by using a framework. Testing frameworks usually already manage many aspects of software testing and make the process much easier for the tester, by functionalities such as verifying expected behavior, assembling tests into test suites, identifying test failures, providing test reports, cleaning up after test execution, finding and executing individual tests etc. [23]

2.1 xUnit

xUnit is a term used for collection of frameworks that derive their functionality and structure from Smalltalk SUnit framework [24]. The concepts first introduced in the SUnit framework and popularized by the JUnit framework [25] were then widely used in several other frameworks, usually called *x*Unit, where *x* is replaced by name of the programming language the framework was developed for, but this naming rule is not always applicable. When discussing testing frameworks, it is often desirable to identify whether it belongs to the xUnit testing family, i.e. it follows the xUnit concepts. [26] [27] That includes the following concepts:

Test runner – is an executable program that runs tests implemented using a xUnit framework and reports the test results. It provides testers with uniform way to execute the tests. The behavior of the test runner may differ in the way it registers the test cases – it either discovers them automatically based on predefined structure, or the tester must explicitly register them. [28]

Test method – each test (test case) is defined as a method, procedure, or function [29]. This test method implements four stages:

1. Setup – preparation of the test data, test doubles, test context etc.
2. Exercise – interactions with the test object.
3. Verify – verification of the outcomes to determine whether it met the expectations.
4. Teardown – setting the test environment to the previous set up (before the setup phase). [30]

Test case class / Test suite – all the test methods (since they are methods) must be collected into a class. The related test methods are put under a class that is called *Test case class*, so a *Test case object* is created for each test by instantiating the test case class once for each test method. [31]

Test case object – for each test method a test case object is instantiated, that allows the test runner to simply manipulate and execute the tests individually. The test case objects are then collected into *Test suite object*. [32]

Test suite object – is a composite[2] of test case objects, it holds the collection of the individual test case objects. The test suite object is called by the test runner to execute and evaluate the tests. [34]

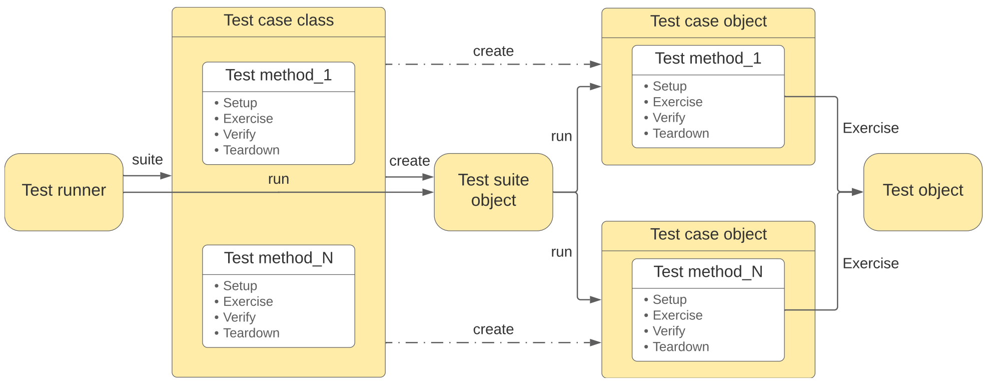

Figure 2.1: xUnit framework diagram (inspired by [28])

[2]Composite design pattern composes "objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." [33]

The overall structure of a xUnit framework and relationships between the just introduced concepts are depicted in diagram 2.1. Since the xUnit approach is used in several testing frameworks, for a tester that is already familiar with one of this frameworks, it is quite effortless to adapt to using other frameworks from the xUnit frameworks family.

2.2 Comparison criteria

This chapter aims to compare some of the C++ testing frameworks, for which a methodology needs to be defined. Different projects and testers might have different criteria, however the goal of this chapter is not a comparison based on project specific criteria, but rather on the most general criteria possible. The criteria will be scored either by [0 - 5 points] (where 5 is the best score) or [yes/no] option. The following criteria will be considered:

Basic usage simplicity [0 - 5 points] – how easy/difficult it is to implement a simple test case. Evaluation of this criterion will be based mainly on very simple test example showed in 2.1 – test with just simple assertion.

```
TEST (SimplestTest) {
    int someNum = 42;
    ASSERT(someNum, 42);
}
```

.

Source code 2.1: Simplest test example inspired by [35]

Fixtures [0 - 5 points] – whether the framework supports setup and teardown steps (for test case classes) and to what extent. In this step, their usage simplicity will be also evaluated, which will be again mainly performed on simple test example 2.2.

Assertion [0 - 5 points] – testing frameworks shall provide tools to actually verify the behavior of the test object, which is usually achieved by assertions. The provided support may vary, i.e. whether assertion of non-trivial data types are supported or whether the assertions provide output with compared values in case of failure.

Exceptions handling [yes/no] – the test object can sometimes throw unexpected exceptions, e.g. access some invalid memory location. In such case, the desired behavior is that the framework will not crash with its test object, but rather catch the exception and report it.

```
SETUP (FixtureTestCase) {
    int someNum = 42;
}

TEST (FixtureTestCase, Test0) {
    ASSERT(someNum, 42);
    someNum = 7581;
}

TEST (FixtureTestCase, Test1) {
    ASSERT(someNum, 42);
}
```

Source code 2.2: Simplest test example with fixtures inspired by [35]

Test doubles support [0 - 5 points] – as explained in 1.11, test doubles are a useful tool to simulate the production environment and test the test object behavior from different perspectives. The test frameworks have different levels of support – none, providing API especially for test doubles and their expectation assertion, or full support including the creation or generation of the test doubles.

Extensibility [0 - 5 points] – various projects might need very specific functionalities, in such case it is desired that the framework is easily extensible or modifiable.

xUnit [yes/no] – whether the framework is from the xUnit family. The frameworks from xUnit family might be easier to learn for testers familiar with the xUnit concepts.

License – under what license the framework is.

2.3 CppUnit

"CppUnit is the C++ port of the famous JUnit framework for unit testing." [36]. So CppUnit is purely derived from the JUnit framework, therefore it shall be the most representative C++ member of xUnit family. This analysis was performed on version 1.12.1.

Basic usage simplicity – implementation of the simplest unit test defined in 2.2 is shown in 2.3. Firstly a class for the test case must be created, it is a good practice to create a separate header file for this purpose. The test suite needs to be explicitly registered, for which three different macros (lines 2, 4 and 9) must be used overall. Only then follows the actual implementation of the test, which needs to be also registered using a macro. [37] [**2 points**]

```
class SimplestTestSuite : public CppUnit::TestFixture {
    CPPUNIT_TEST_SUITE(SimplestTestSuite);
    CPPUNIT_TEST(SimplestTest);
    CPPUNIT_TEST_SUITE_END();
protected:
    void SimplestTest();
};

CPPUNIT_TEST_SUITE_REGISTRATION(SimplestTestSuite);

void SimplestTestSuite::SimplestTest() {
    int someNum = 42;
    CPPUNIT_ASSERT_EQUAL(someNum, 42);
}
```

Source code 2.3: Simplest CppUnit test example

Fixtures – the support of fixtures is decent, as shown in 2.4, adding `setUp()` and `tearDown()` steps is quite straightforward and does not add additional complexity (does not require usage of any additional macros). For each test, an individual test case object is created, therefore it is ensured the tests do not affect each other (i.e. in case of 2.4 the `someNum` variable is initialized for each test separately with value 42). But objects need to be allocated dynamically in the `setUp()` step, if they need to be initialized for each individual test. [38] [**3 points**]

```
class FixtureTestSuite : public CppUnit::TestFixture {
    CPPUNIT_TEST_SUITE(FixtureTestSuite);
    CPPUNIT_TEST(Test0);
    CPPUNIT_TEST(Test1);
    CPPUNIT_TEST_SUITE_END();
public:
    void setUp(); // tearDown() fixture would be analogous
protected:
    int someNum;
    void Test0();
    void Test1();
};

CPPUNIT_TEST_SUITE_REGISTRATION(FixtureTestSuite);

void FixtureTestSuite::setUp() {
    someNum = 42;
}

void FixtureTestSuite::Test0() {
    CPPUNIT_ASSERT_EQUAL(someNum, 42);
    someNum = 7581;
}

void FixtureTestSuite::Test1() {
    CPPUNIT_ASSERT_EQUAL(someNum, 42);
}
```

Source code 2.4: Simplest CppUnit test example with fixtures

Assertion – CppUnit framework provides quite basic support of assertions – condition statements, equality of basic data types, and additionally comparison of floating-point numbers with delta precision. This supported assertions are provided both with and without message option. But there is absolute lack of assertions for ”less than”, ”greater than”, and inequality. Also, assertions evaluating whether the test object threw or did not throw an exception are included. [39] [**2 points**]

Exceptions handling – CppUnit uses a protector [40], which is a wrapper around the tests. It shall catch all thrown exceptions and prevent the tests from crashing. Additionally a macro `CPPUNIT_TEST_EXCEPTION` can be used for test registration to catch a specific exception. [**yes**]

Test doubles support – CppUnit does not include any test doubles support whatsoever. [**0 points**]

Extensibility – the source code obtained from the project official site [36] includes all needed sources that can be freely modified. The archive also includes descriptive coding guidelines. The code is reasonably split into modules, but unfortunately the code itself contains almost no comments or documentation whatsoever, but official documentation site [41] is quite verbose and sufficient for most purposes.

The older versions of CppUnit required the STL[3], RTTI[4], and exception handling. But these dependencies have been successfully eliminated in the newer versions and according to the code guidelines, it aims to avoid unnecessary dependencies in general. [**4 points**]

xUnit – [**yes**]

License – [**GNU LGPL**] [44]

Overall the CppUnit framework provides the basic functionalities that are desired from a xUnit framework, but does not provide any additional tools such as test doubles support and better assertion support. It might be suitable for smaller projects with no test doubles demand, but its usage is not as easy as it could be.

2.4 Boost Test Library

Boost Test Library [45] is part of the Boost library [46] that is widely used and introduces a few new approaches to C++ testing. This section will consider the 1.76.0 version.

[3]”The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.” [42]

[4]”Run-time type information is a mechanism that exposes information about an object’s data type at runtime” [43]

Basic usage simplicity – as shown in 2.5, the implementation of simple test is very close to ideal, it is almost as simple as the one defined in 2.2. [**5 points**]

```cpp
#define BOOST_TEST_MODULE Simplest test     // Defines the name of this program

BOOST_AUTO_TEST_CASE(SimplestTest) { // no additional registration of the test is needed
    int someNum = 42;
    BOOST_CHECK_EQUAL(someNum, 42);
}
```

Source code 2.5: Simplest Boost.Test test example

Fixtures – the Boost testing library has a very original approach to fixtures, as shown in 2.6. The fixture is defined through a struct, where the constructor is later used as `setUp()` and the desctructor as `tearDown()` for the associated tests. By using `BOOST_TEST_MESSAGE` macro a descriptive message can be added for more verbose output.

This different approach allows the fixtures to be associated with test suites, but also with individual tests with `BOOST_FIXTURE_TEST_CASE` macro instead of `BOOST_AUTO_TEST_CASE`. Therefore fixtures can be defined globally and used across different test suites. The struct is created for each test individually, therefore the tests do not affect each other through the fixtures. Overall it provides very good functionality, but its definition still demands more than minimal effort. [47] [**4 points**]

```cpp
#define BOOST_TEST_MODULE Simplest fixtures

struct CustomFixture {
    CustomFixture() : someNum(42) {BOOST_TEST_MESSAGE("setup fixture");}
    ~CustomFixture()              {BOOST_TEST_MESSAGE("teardown fixture");}

    int someNum;
};

BOOST_FIXTURE_TEST_SUITE(CustomTestSuite, CustomFixture) // associate the fixture with test suite

    BOOST_AUTO_TEST_CASE(Test0) {
        BOOST_CHECK_EQUAL(someNum, 42);
        someNum = 7581;
    }

    BOOST_AUTO_TEST_CASE(Test1) {
        BOOST_CHECK_EQUAL(someNum, 42);
    }

BOOST_AUTO_TEST_SUITE_END()
```

Source code 2.6: Simplest Boost.Test test example with fixtures

Assertion – the Boost test library provides wide variety of assertions. Firstly, three severity levels for assertions are available:

- Warning – in case of an unfulfilled expectation, a warning is printed out, but the deviation does not count towards the errors count and does not stop the test from continuing.
- Error – a corresponding error is printed out, the deviation counts toward the errors count, but it does not stop evaluation of the rest of the test.
- Fatal error – the corresponding deviation is printed out, counts toward the errors count and stops the test from further evaluation. [48]

The most universal assertion available is `BOOST_TEST()` macro that evaluates provided statement, which is just alone quite standard functionality for a test framework, but in case of failure, this particular macro prints out the whole statement with actual values, which is unusual for simple statement assertion. Additionally it automatically detects certain comparisons, e.g. comparison of C-strings is evaluated as if the std::string objects were used, for floating point values a tolerance might be provided and many more. It is actually a universal operator supporting equality, ”less than”, ”greater than” and bitwise comparisons for non-trivial data types. [49]

Also assertion for expectations of exceptions [50], timeout [51], and output streams [52] of the test object are provided. [**5 points**]

Exceptions handling – thrown exceptions are caught and do not affect evaluation of following tests, information about thrown exceptions are printed out. [**yes**]

Test doubles support – the Boost test library itself does not provide any test doubles support. But a compatible library Turtle [53] was developed to add the functionality. The Turtle library was especially developed to supplement the Boost test library, but it is independent from the original Boost project and its latest version was published in 2014, since then, no updates were made.

The Turtle library provides an API to set and check the test double calls expectations, i.e. what function shall be called, how many times it shall be called and with what arguments. It detects also unexpected calls and provides an option to ignore the input arguments or a given number of calls. Also a mechanism for generating actual test doubles is provided, therefore the test doubles do not need to be created manually from scratch. This is achieved again by a set of macros, for example to generate a test double class with method, as shown in 2.7. But the

```cpp
MOCK_CLASS(MockClass) { // mock of a class
    // mock of class method: int method(int i) {};
    MOCK_METHOD(method, 1, int(int));
};

BOOST_AUTO_TEST_CASE(mockUsage) {
    MockClass mockObject;
    // expect one method call with input parameter equal to 42 and instruct the mock to return 7581
    MOCK_EXPECT(mockObject.method).once().with(42).returns(7581);
    testObject.trigger();
    // verify all existing mock objects expectations
    mock::verify();
}
```

Source code 2.7: Simplest Boost.Test test example with test double

usage of the macros is quite complicated, since a special macro needs to be used for destructors, constructors, static methods, and operators, to name but a few. It has also many documented limitations – objects for which test doubles cannot be generated, such as non-virtual methods, template methods, methods with a *throws* specifier, and many others. So usability of the test doubles generating is rather very limited and unfortunately the documentation is very brief about the usage. [54] [**3 points**]

Extensibility – the source code is reasonably structured and very well documented. But the Boost test library is only available by downloading the whole Boost library and the test library implementation contains many dependencies from the whole library. The whole boost library source code (before compilation) has around 150 MB, which suggests how huge this library is, therefore expanding or modifying the framework might be quite difficult because of the amount of dependencies. [**2 points**]

xUnit – the usage of fixture is slightly unusual, but overall it fulfills the xUnit concepts. [**yes**]

License – [**Boost Software License**] [55]

Overall the Boost test library provides wide variety of functionalities, but it comes with a price of big library. Additionally the test doubles support is provided just by the external Turtle library, which unfortunately did not stand to the Boost library simplicity and universal usage, therefore makes quite poor extension.

2.5 Google Test

Google Test [56] is currently a widely popular C++ testing framework, since it offers plenty of useful functionalities and the development team still works on new releases. This section will be concerned with the v1.11.0 version.

Basic usage simplicity – the simplest test case displayed in 2.8 is basically ideal, same as 2.1 defined in comparison criteria. [**5 points**]

```
TEST(SimplestTestSuite, SimplestTest) {
    int someNum = 42;
    EXPECT_EQ(someNum, 42);
}
```

Source code 2.8: Simplest Google Test test example

Fixtures – the approach of Google Test framework is very similar to the Boost.Test approach, as shown in 2.9. The fixture is defined through a class, but unlike in Boost, it does not use a constructor and destructor as `setUp()` and `tearDown()` steps. It is rather achieved by inheritance from `testing::Test` class (or other for more parameterizable fixtures) and overriding `SetUp()` and `TearDown()` methods, the class is instantiated for each test individually, therefore the tests do not affect each other through the fixture (but that can be also achieved if demanded, by defining **`static`** methods and attributes). It demands even less lines than the Boost.Test library, the result code does not include any unnecessary lines, such as explicit registration of the fixtures. [57]

```
class CustomFixture : public ::testing::Test {
 protected:
    void SetUp() override {
        someNum = 42;
    }
    // void TearDown() override {}  // tearDown() is not needed in this case

    int someNum;
};

TEST_F(CustomFixture, Test0) {
    EXPECT_EQ(someNum, 42);
    someNum = 7581;
}

TEST_F(CustomFixture, Test1) {
    EXPECT_EQ(someNum, 42);
}
```

Source code 2.9: Simplest Google Test test example with fixtures

Additionally various global fixtures can be combined, which will be applied to all tests gradually in LIFO[5] approach. Also value-parameterized tests can be implemented through the fixtures – one test is run several times for different values defined as parameter of the test. Also *Typed tests* are introduced – by combining fixtures and templates, one test can be performed for several types, if they share the same expectations. [58] [**5 points**]

[5]Last In, First out principle

Assertion – Google Test provides two following severity levels for assertions:

- Nonfatal failure – a corresponding error is logged and allows the current test to continue running.
- Fatal failure – a corresponding error is logged and the current test is aborted.

A macro for equality, "less than", "less or equal", "greater than", and "greater or equal" are provided for each supported type. The assertions are supported for basic data types, strings, C-strings and floating point values, in other words all possibly supported data types, but for the non-trivial data types, different assertions must be used, i.e. `ASSERT_DOUBLE_EQ()` for equality of doubles, `ASSERT_STREQ()` for equality of C-strings etc., therefore there is not an universal operator as in the Boost.Test library. Also exceptions and timeout assertions are provided. [59]

Additionally, Google Test introduces built-in *matchers*, a set of methods for variety of comparisons. It supports generic comparisons, e.g. `IsNull()`, `Optional()` (checks whether argument of type `optional<>` contains a provided value), `VariantWith<T>()` (checks whether provided value is `variant<>` that holds the alternative of type T with a value matching the provided value), and more. Further it provides extended support for comparison of floating point values (approximate equality etc.), strings (support of regular expressions, suffix and prefix requirements, substrings searching, to name but a few), containers (i.e. whether a container contains provided value, or a container is a subset of another container), members, pointers, tuples, etc. [60] [**5 points**]

Exceptions handling – by default, Google Test catches thrown exceptions and reports them as failure. But this can be also disabled, for example for debugging purposes. [58] [**yes**]

Test doubles support – the Google Test aimed to create a test doubles support with JMock [61] as an inspiration, but with C++ in mind. It provides an API to set and check the test double calls expectations, i.e. what method shall be called, how many times, and with what arguments. The expectations are defined through the already mentioned *matchers*, which allows to check variety of expectations, rather than just equality of arguments and number of calls. The expectations are overall quite parameterizable, i.e. the sequence of calls can be either arbitrary or strictly defined, individual calls of one method can be (partially) ignored etc.

The Google Test also provides macro `MOCK_METHOD()` to generate a test double of a method. Then the test double can be controlled via macros for setting its actions – Google Test defines several *actions* [62] that allow several ways to return a value (one predefined value, iterate through provided list for each call, and more), perform side effects (modifying input/output arguments, throwing exceptions, etc.) or invoking predefined functions, functors, or lambdas. This is all implemented in a convenient way, where it can be used effectively in any case – default actions for each call might be defined, then additional actions can be added for individual call or predefined number of calls.

```cpp
class CustomTestDouble : public OriginalClass {
public:
    // mock of class method: int method(int i) {};
    MOCK_METHOD(int, method, (int));
};

TEST(SimplestTestSuite, SimplestTest) {
    CustomTestDouble mock;
    // expect one method call with input parameter equal to 42 and instruct the mock to return 7581
    EXPECT_CALL(mock, method()).With(Eq(42)).Times(1).WillOnce(Return(7581));
    testObject.trigger();
    // test double expectations will be checked automatically
}
```

Source code 2.10: Simplest Google Test test example with test double

The creation and usage of test double is shown in 2.10. The Google Test supports automatic creation of test doubles only for methods, test doubles of classes must be created manually as demonstrated in the example 2.10 (by deriving from the original class), the class methods are then mocked and controlled as explained. This approach is not fully automatic, however it provides the tester with reasonable amount of control over the test double. [63] [**5 points**]

Extensibility – the official documentation includes several notes describing how certain functionalities can be further extended. The source codes include plenty of comments explaining the implementation, the implementation is distributed into a few modules, but the distribution does not fully correspond with the individual functionalities and there seem to be no description about it (the official documentation [56] is rather a tutorial and does not provide standard reference style description of modules and classes). The implementation has also several dependencies to standard libraries including STL. [**4 points**]

xUnit – [**yes**]

License – [**BSD 3-Clause ”New” or ”Revised” License**] [64]

Overall Google Test provides the widest range of functionalities so far. The basic usage is simple and intuitive and is not influenced by additional more complex features. But at the same time, the framework is very parameterizable through the *matchers* and *actions*.

2.6 CppUTest

CppUTest [65] is a C++ and C testing framework, frequently used in embedded projects, but suitable for other projects too. CppUTest aimed for simple usability and design, portability for both new and old platforms and compliance with the xUnit concepts. This section will be concerned with the 3.8 version. [65]

Basic usage simplicity – the basic usage is very close to the reference 2.1, as demonstrated in 2.11, but a test suite must be defined and registered first in order to implement a test. [**4 points**]

```
TEST_GROUP(SimplestTestSuite) {};

TEST(SimplestTestSuite, SimplestTest) {
    int someNum = 42;
    CHECK_EQUAL(someNum, 42);
}
```

Source code 2.11: Simplest CppUTest test example

Fixtures – the fixtures usage shown in 2.12 complies with the xUnit approach. The fixtures are defined as a part of a test suite, the corresponding `setUp()` and `tearDown()` steps are then performed for each related test individually, therefore the individual tests do not influence each other. The fixture definition is simple and straightforward, no additional explicit registration of the test suite (as in Boost Test library) is needed.

```
TEST_GROUP(SimplestTestSuite) {
    void setup() {
        someNum = 42;
    }
    // void teardown() {}   // tearDown() is not needed in this case

    int someNum;
};

TEST(SimplestTestSuite, Test0) {
    CHECK_EQUAL(someNum, 42);
    someNum = 7581
}

TEST(SimplestTestSuite, Test1) {
    CHECK_EQUAL(someNum, 42);
}
```

Source code 2.12: Simplest CppUTest test example with fixtures

Otherwise the CppUTest framework does not provide any additional functionality, such as global fixtures in Google Test, but similar outcomes can be achieved by using inheritance in the test suites. [**4 points**]

Assertion – CppUTest provides intermediate support of assertions, it supports evaluation of equality for integer numbers, unsigned numbers, floating point values with tolerance, strings (case sensitive and insensitive, substring searching), pointers, whole memory areas, and more, the assertions allow to add output message in case of failure. The support of other relation than equality is a bit more modest, since just one macro `CHECK_COMPARE()` is available to compare two values with provided operation (which must be defined between the provided values), which has its limitations, but is easily extensible. Also an assertion for exception expectations is provided. Different severity levels are not supported, all assertions result in fatal errors. [66] [**3 points**]

Exceptions handling – the thrown exceptions are caught and reported, but there seem to be no way to switch off this functionality (e.g. for debugging). [**yes**]

Test doubles support – CppUTest includes a support API for testing with test doubles called CppUMock. CppUMock provides methods for setting and checking mock expectations, such as numbers of calls and passed input parameters, the calls tracing can be also switched to ignore any unnecessary data. Also a way to set mock to return value and output parameters is supported – this is achieved by simply passing data from the test driver to the mock, the framework does not put any restrictions on the data whatsoever. It does not provide any additional logic for the outcomes of the mock, beside an option to set default return value, but more complex logic can be implemented by the tester and easily ported with the provided API (e.g. iterating through a list of return values). The usage of the API is demonstrated in 2.13.

```
class CustomTestDouble : public OriginalClass {
public:
    virtual int method(int arg) {
        // register call of the call and its input parameter an return preset value
        return mock().actualCall("method").withParameter("arg", arg).returnIntValue();
    }
};

TEST(SimplestTestSuite, SimplestTest) {
    // expect one method call with input parameter equal to 42 and instruct the mock to return 7581
    mock().expectOneCall("method").withParameter("arg", 42).andReturnValue(7581);
    testObject.trigger();
    mock().checkExpectations();
}
```

Source code 2.13: Simplest CppUTest test example with test double

CppUMock provides a simple and intuitive API for controlling test doubles, but it does not support the actual creation of the mocks at all. As the official documentation states [67], its goal was mainly very simple use, the developer stays in control, and no code generation, which was fully achieved. However CppUTest is fully compatible with Google Test and Google Mock, therefore more complex functionalities can be easily added if needed. [67] [**3 points**]

Extensibility – as already mentioned, the official documentation is rather a tutorial than a standard documentation, and the implementation itself contains comments or any other explanation just rarely. The implementation is divided into a few modules, but the partitioning does not really correspond to individual functionalities. The framework provides a way to hook pre-test and post-test actions via plugin [68], but there seem to be no other support or guidelines to modify the framework or add additional functionality. On the other hand, the framework is still quite small, self-sustained and intuitively designed. [**3 points**]

xUnit – [**yes**]

License – [**BSD 3-Clause "New" or "Revised" License**] [69]

Overall the CppUTest framework provides basic functionality in an intuitive and simple way. It does not apply any unnecessary constraints, therefore the framework usage can by customized by the tester as needed, but that can also result in more failures if used incorrectly.

2.7 Results overview

	CppUnit	BoostTest	GoogleTest	CppUTest
Usage simplicity	2	5	5	4
Fixtures	3	4	5	4
Assertion	2	5	5	3
Exceptions handling	yes	yes	yes	yes
Test doubles	0	3	5	3
Extensibility	4	2	4	3
xUnit	yes	yes	yes	yes
License	GNU LGPL	Boost Software License	BSD 3-Clause License	BSD 3-Clause License

Table 2.1: Overview of C++ framework comparison results

Four frameworks were analysed and evaluated, the examined frameworks where selected based on their users count and historical influence. Eight basic criteria were tracked, overview of the result is enlisted in the table 2.1.

The importance of the considered criteria might differ for various projects, i.e. Google Test provides wide set of functionalities, but some smaller projects may not even use them, therefore using smaller framework such as CppUTest can be more convenient.

Chapter 3

Design of the CppUTest extension

One of the main goals of this thesis is to design and implement extension of the C++ testing framework CppUTest analysed in section 2.6, with focus on support of test doubles. As already mentioned, CppUTest provides API for setting test doubles expectations and their checking, but does not support creation of the test doubles itself. The developers of the CppUTest framework suggest that they did not create the test doubles creation on purpose, since it leaves the tester in charge and allow them to implement any behavior of the test double.

But in practice, majority of the used test doubles share just a few common features that could be automatized, instead of their manual and tedious reproduction in the individual test doubles. The aim of this chapter is to identify these features and design an extension that will automate their implementation, with respect to the CppUTest philosophy and goals.

3.1 Functional requirements

This section aims to identify the common features of majority of the test doubles and formulate functional requirements that will lead to automation of implementation of such test doubles with regard to the CppUTest framework. The following requirements are applicable for the designed extension.

[**F_1** | **Automatic actual call**] – all test doubles created with the extension shall automatically call the CppUTest method `actualCall()` with the mocked method or function name to register the call.

Note: The CppUTest framework demands every test double of method or function to register the call, but currently it needs to be done manually for all individual mocks. Therefore this requirement aims to fulfill this automatically.

[**F_2 | Constant return value**] – test doubles created with the extension shall provide an option to set their return and output[6] value(s) for parameterizable number (including one) of calls using the CppUTest API.

Note: The CppUTest already provides API for setting the test double return or output value, but this desired behavior needs to be currently manually processed in the test double. This requirement states that the extension shall support creation of test doubles that will automatically process this desired behavior.

[**F_3 | Key-value return value**] – test doubles created with the extension shall provide an option to set their return and output value(s) based on a provided key-value map(s), where the key corresponds to one of the input parameters and the assigned value is then the desired return or output value.

Note: The CppUTest API does not explicitly provide such functionality, therefore this requirement also requires to provide some way to control this behavior through the test driver. In other words, this requirement states that the test double shall allow the tester to define a map for its return value or individual output value(s), where one of the input parameters is used as a key.

[**F_4 | Default return value**] – test doubles created with the extension shall provide an option to set their default return and output value(s) for undefined number of calls using the CppUTest API.

Note: The CppUTest provides similar feature, but the default value can be actually set only from the test double, not the test driver as demanded in this requirement. Therefore the support shall be be added to the test driver API as well.

[**F_5 | Return values priority**] – if the test double created with the extension has set return and output value(s) through more than one of the provided options when its call is processed, the value(s) shall be used according to the following priority:

1. If the value(s) is/are set as constant return value defined in F_2, this/these value(s) shall be used with the highest priority.
2. If the value(s) is/are not set as constant return value defined in F_2 and is/are set as key-value return value defined in F_3, the key-value return value shall be used.

[6]Output value is an output parameter of a function or method.

3. If the value(s) is/are not set as constant return value defined in F_2 and is/are not set as key-value return value defined in F_3, but is/are set as default return value defined in F_4, the key-value return value shall be used.

[**F_6 | Return value not set**] – if the return value of test double created with the extension is not set when its call is processed, the corresponding test shall fail.

[**F_7 | Original function call**] – test doubles created with the extension shall provide an option to call the original function/method (the production function/method the test double replaces) and propagate its behavior (return and output value(s)) without changing. The created test double shall support this functionality in three following ways:

1. The original function/method will be called for defined number of the test double calls (analogous to the constant return value defined in F_2). If this condition is met, the counter for defined number of calls shall be decremented regardless of the following conditions results.
2. The original function/method will be called based on a list of values which correspond to one of the input parameters value, i.e. if value of the provided parameter in the current call is equal to one of the values in the provided list, the original function/method shall be called.
3. The original function/method will be called by default for undefined number of calls.

If at least one of the conditions above is met, the original function shall be called.

Note: The CppUTest does not explicitly provide an option to set the test double to call the original function, therefore this functionality shall be added to the test driver API as well.

[**F_8 | Throw an exception**] – test doubles created with the extension shall provide an option to set them to throw a provided exception. The created test double shall support this functionality in three following ways:

1. The exception will be thrown for defined number of the test double calls (analogous to the constant return value defined in F_2). This setting of exception shall override other possibly set exception in the following two steps.
2. The exception will be thrown based on a key-value map, where the key corresponds to one of the input parameters (analogous to

the key-value return value defined in F_3). The exception type may vary for different key values. This setting of exception shall override other possibly set exception in the following one step.

3. The exception will be thrown by default for undefined number of calls (analogous to the default return value defined in F_4). This setting does not override other possibly set exception.

Note: The CppUTest does not explicitly provide an option to set the test double to throw an exception, therefore this functionality shall be added to the test driver API as well.

[**F_9 | Parameters automatic registration**] – the test doubles created with the extension shall register all its calls parameters values using the CppUTest method `withParameter()` with the individual parameter name and value.

Note: The parameters values actual expectations can be set and checked in the test driver using the CppUTest API.

[**F_10 | Invoke side effects**] – the test doubles created with the extension shall provide an option to invoke a passed function, functor or lambda function. The test double is not responsible for any other evaluation of such invocation, for instance passing its return value. The created test double shall support this functionality in three following ways (analogous to throwing an exception):

1. The side effect will be invoked for defined number of the test double calls. This setting of side effect shall override other possibly set side effect in the following two steps.
2. The side effect will be invoked based on a key-value map, where the key corresponds to one of the input parameters. The side effect type may vary for different key values. This setting of side effect shall override other possibly set side effect in the following one step.
3. The side effect will be invoked by default for undefined number of calls. This setting does not override other possibly set side effect.

Different desired behavior types, such as invoking side effects, setting output parameters values, throwing an exception, and more, can be freely combined, even though it might not be obvious from the requirements. Priority of some disjunct combinations, such as combination of desired return value and throwing an exception, is not yet determined and will be considered in the following sections.

3.2 Non-functional requirements

This section aims to identify non-functional requirements relevant for the extension. The most important concern is to design the extension so it will comply with the CppUTest core design principles. The following requirements are derived mainly from these principles.

[**N_1 | Simple design and usage**] – the extension shall be simple in design and simple in usage. Meaning no new constraints (e.g. Google Test *Actions* or *Matchers* alternatives) shall be introduced in the test doubles created with the extension.

Note: This requirement is clearly very vague, imprecisely defined and immeasurable. But it is important to explicitly formulate that the extension shall follow the CppUTest simplicity that brings intuitive usage and no unnecessary restrictions.

[**N_2 | No code generation**] – the extension shall not use code generation for the creation of the test doubles.

Note: The motivation behind this requirement is to keep the tester in charge and the code base understandable. The functional requirements do not grant to cover all test double possible needed behavior, therefore it can be assumed that test doubles created with the extension may be combined with manually implemented test doubles. The code generation prohibition ensures that using of both manually implemented and created with the extension test doubles will not introduce additional complexity.

[**N_3 | No complicated macros**] – the extension shall not use non-trivial macros for creation of the test doubles.

Note: This is again one of the CppUTest mock support principles, which enhances its simple usage and design.

[**N_4 | C++ language**] – the extension shall be implemented in C++ programming language. Support of a C API is out of scope of this work.

[**N_5 | CppUTest compatibility**] – the extension shall be compatible with the CppUTest framework and it shall use its existing test doubles support as much as possible without changing the current behavior principles.

The requirements derived from the CppUTest core design principles are rather vague and immeasurable, however these constraints are still needed to ensure as ideal compatibility as possible with the CppUTest framework. These requirements need to be defined and respected in the design to satisfy the testers that chose the CppUTest framework for its characteristic simple usage and design.

3.3 Test double design

The defined requirements imply what use cases shall be supported, especially by the created test double. The characteristics of design of this extension differ from a regular software development process, since it focuses more on improving an existing test double creation and modifying an existing library, therefore the use cases will not be discussed in a regular manner. This chapter will rather focus on determining the key dilemmas of the designed extension, defining its expected behavior in more detail and determining the core design decisions.

Firstly, the requirements are relevant mainly for the created test double itself, the rest of the design decisions will need to be made mostly based on the design of the test double. Since the test double implementation shall require minimal manual effort from the tester, but at the same time satisfy all the functional requirements in any case, the aim is to design a test double that will determine its desired behavior dynamically based on the data provided from the test driver.

The designed behavior of the test double is presented in the activity diagram 3.1. The test double gradually fetches its desired behavior and based on the accepted data performs the desired actions. The desired behavior is set and fetched from the test driver, which will be discussed in detail in the following sections. The test double activities can be briefly separated into the following steps.

1. Register the actual call.
2. Register values of the provided parameters. This must be ensured before any modification of the output parameters, either by original function call or by filling the values based on desired behavior.
3. (Optional) Call the original function and save its result(s).
4. Fill values of the output parameters according to the desired behavior. Since this is performed after the optional original function call, it ensures the tester can set the test double to either call the original function and preserve its original output parameters values, or to overwrite them with desired values.
5. (Optional) Invoke side effect, which can include throwing an exception.
6. Determine the return value, applicable only for test doubles with return value. Again, if no desired return value is set, but the original function was called, its return value will be propagated.

Since the goal of the CppUTest is to leave the tester in charge and do not create any additional restrictions on its usage, the desired behavior options

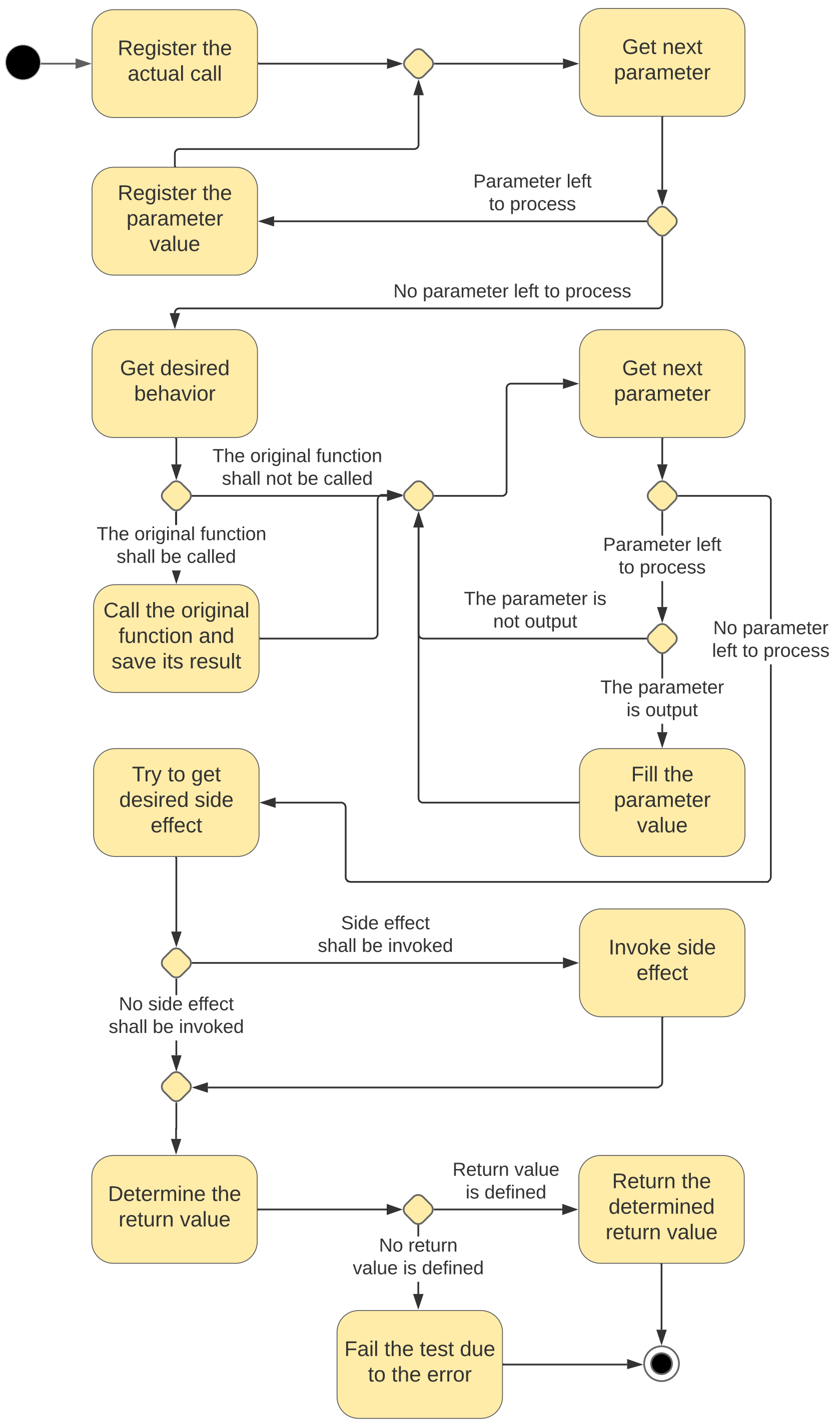
Register the actual call
Get next parameter
Register the parameter value
Parameter left to process
No parameter left to process
Get desired behavior
Get next parameter
The original function shall not be called
The original function shall be called
Call the original function and save its result
Parameter left to process
The parameter is not output
No parameter left to process
The parameter is output
Try to get desired side effect
Fill the parameter value
Side effect shall be invoked
Invoke side effect
No side effect shall be invoked
Determine the return value
Return value is defined
Return the determined return value
No return value is defined
Fail the test due to the error

Figure 3.1: Activity diagram of the test double

can be freely combined and it is the tester's responsibility to use only logically valid combinations. These possible combinations can include the following options.

- Use each test double parameter as purely input parameter in one test and as output parameter in another test.
- Call the original function/method in the test double, but set the test double return value to a custom value (different from the original function/method call return value).
- Call the original function/method with successful result, but throw an exception from the test double.

In order to perform the actions identified in this section, the test double needs to have the following data provided.

- Values of parameters provided by the test object.
- Names of the parameters, since they are necessary to register the parameters values via the CppUTest API. The names will need to be explicitly handled to the test double in the implementation by the tester.
- The desired behavior, which will be fetched from the test driver via the CppUTest API that will be discussed in the following sections.

3.4 Desired behavior determination

Based on the requirements, several options to set desired behavior of the test double shall be supported and there are no explicit restrictions on their combinations. In the test double design section 3.3, behavior for most of the combinations was already defined in the activity diagram 3.1, but priorities of different types of return and output values settings, original function call settings and side effects settings were not designed in detail yet. All of this four types of parameterizable desired behavior (output value, return value, original function call and side effect) actually shall share three following different approaches with corresponding priorities to be set through the test driver.

1. The desired behavior is set as a *constant* option for a defined number of calls. This means the desired behavior will be applied for the provided number of calls, after the number is reached, the desired behavior will no longer make an effect on behavior of the test double.
2. The desired behavior is set as a *key-value* map option, where the desired behavior is selected based on the value received from the test object of one of the input parameters, this received value is used as a key to determine the desired behavior. One key-value entry can be set for undefined

number of calls (until the test double is cleared). If the received value is not found in the provided map, this option cannot be applied.

3. The desired behavior is set as the *default* option, which is applied if none of the above was set for infinite number of calls. If the desired behavior is not set even in this option, it is handled differently for the following cases.

    - For the return value, this indicates that it is unknown what value shall be returned. If neither original function was called, therefore its return value is also unknown, which shall lead to failure of the evaluated test.
    - This is not applicable for output parameters – if a desired value of the output parameter is not set, the parameter is then treated as a purely input parameter.
    - If it is not desired to call the original function, it is simply not called and its behavior is not propagated anyhow.
    - If no desired side effect is set, it indicates no side effect shall be invoked.

3.5 Extension of the existing CppUMock API

This section aims to answer a question how to integrate the extension with the existing CppUMock library, which is part of the CppUTest framework. The CppUMock consists of two parts – test driver API and test double API. The relationships between the APIs and their usage is demonstrated in domain diagram 3.2.

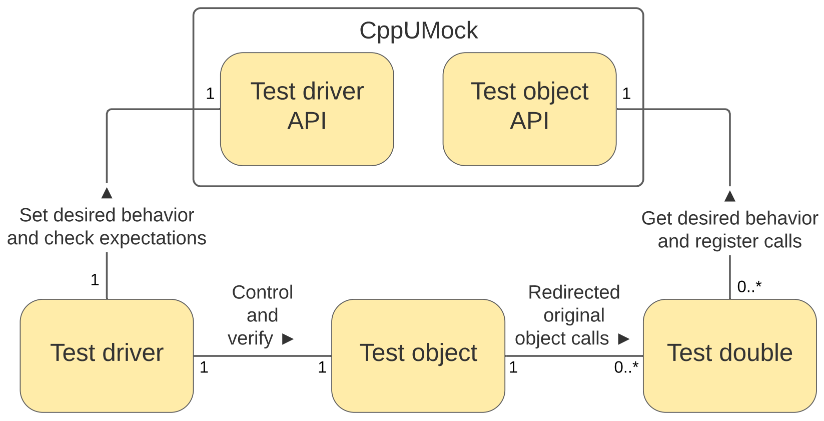
CppUMock
Test driver API
Test object API
1
1
Set desired behavior and check expectations
Get desired behavior and register calls
1
0..*
Test driver
Control and verify ▶
Test object
Redirected original object calls ▶
Test double
1
1
1
0..*

Figure 3.2: Domain diagram of CppUMock usage

The test driver API provides methods through which the tester can control the test double and check the expectations. This includes setting the test double return value, output parameters, expectations of calls, input parameters values etc. The test double API then provides the second side – getting the behavior expectations and registering the actual calls and their input parameters.

The functionalities the extension shall support, such as combining of constant, key-value, and default return value are, unfortunately highly linked to the functionalities already supported by the CppUMock. Therefore it will be more convenient to directly integrate the functionalities to the existing library and their corresponding code files instead of creating a separate library. However compatibility with the original framework shall be preserved, so the current behavior of the CppUTest is not changed.

For further design, a crucial information regarding the CppUTest library is that it does not use templates in neither of its functionalities, meaning that desired behavior settings and expectations, such as desired return value, are explicitly supported only for the following (C++) data types – `bool`, `int`, `unsigned int`, `long int`, `unsigned long int`, `long long`, `unsigned long long`, `double` (with optional accuracy for input parameters expectations), `const char*`, `void*`, `const void*` and `void (*)()`. Thus for its public API, it defines one method for each data type if needed, for example for setting test double desired return value. Internally, a class `MockNamedValue` is used for storing the values uniformly, which is implemented using the union construct. The `MockNamedValue` class is used to store all the relevant test double desired behavior and CppUMock highly relies on its usage, therefore the new features shall be also implemented with usage of this class, instead of introducing templates to the existing constructs, which would otherwise corrupt the compatibility.

3.6 Test double creation API

The aim of this section is to consider the test double API possibilities. Since the one of the CppUTest design principles is to leave the tester in charge and the test double functionalities designed in the section 3.3 do not fulfill all possible test double requirements and scenarios desired in a testing process (but it shall fulfill most of them), the designed test double shall be easy to combine with manually defined test doubles. Therefore the test double will be created for each function or method individually, as it is also designed in other testing frameworks analysed in chapter 2.

Then there are two possible options to implement a test double function or method that allow easy creation for the tester:

1. Using macro – a macro for defining and creating test double, similar to `MOCK_METHOD()` in Google Test.

2. Function call – the functionality of the function or method will be delegated by calling a designed function.

Both options would be sufficient to provide the required functionality. But the second option, function call, complies more with the CppUTest design principles, since one of the principles is to avoid complex macros. Another advantages of the second option are the following.

- The tester can combine functionality provided through the extension with their custom additional implementation.

- If combined with fully manually implemented test doubles, the code base will be more consistent.

- It does not bring any additional restrictions, such as test doubles of non-virtual methods would not be allowed (as in case of Google Test and Turtle extension of the Boost Test).

Therefore the creation of the test double in the extension will be supported through a function. Such function shall be able to accept undefined number of arguments, which can be achieved through variadic function template [70]. However it shall be considered that one of the CppUTest principles is to avoid templates, nevertheless the test double creation API will be separated from the existing CppUTest implementation, therefore it will not affect the existing implementation. Also there is no other reasonable solution for defining function with arbitrary number of arguments. Thus the function shall be defined as presented in 3.1, where the option with return value will be defined for individual data types supported by CppUTest, to ensure compatibility with the CppUMock API.

```
// test double without return value
template <typename... Types>
void mockWithoutReturnValue(char* method_name, char** args_names, char** args_types,
    Types&&... arg);

// test double with int return value
template <typename... Types>
int mockReturningInt(char* method_name, char** args_names, char** args_types, Types&&... arg);
```

Source code 3.1: Designed Test double creation API definition

The test double function defined in 3.1 shall accept the following parameters:

- `const char* method_name` – name of the method/function the test double replaces.
- `const char** args_names` – array of C-strings of the parameters. The names are used for registering values of the parameters provided by test object in the call.
- `const char** args_types` – array of C-strings of names of types of the parameters. The names shall be provided only for data types unsupported by CppUTest (typically user defined classes). For CppUTest supported data types parameters, the values shall be empty C-strings.
- `Types... arg` – the passed parameters of the test double function.

However, the presented definition does not provide all possibly required data to the test double function, since a reference to the original function shall be provided in cases the original function shall be called. Also the original function reference shall not be a mandatory parameter in all cases, since test doubles are in testing process sometimes used to substitute for a unit that is not implemented yet, therefore in some cases the reference to the original function is not yet known. Thus two options shall be always provided – one with no original function reference (already defined in 3.1) and one with the original function reference, as defined in 3.2.

```cpp
// test double with original function reference and without return value
template<typename F, typename... Types>
void mockWithoutReturnValueWithOriginalFunction(char* method_name, char** args_names,
    char** args_types, F original_function, Types&&... arg);

// test double with original function reference and with int return value
template<typename F, typename... Types>
int mockReturningIntWithOriginalFunction(char* method_name, char** args_names, char** args_types,
    F original_function, Types&&... arg)
```

Source code 3.2: Designed Test double creation API definition with original function/method reference parameter

The second alternative defined in 3.2 accepts one additional parameter, `F original_function`, which shall include reference to the original function, and if desired, shall be called with the provided parameters. Again, the function option with return value shall be defined for individual data types supported by CppUTest.

3.7 Test driver API

As already mentioned, the CppUMock has test driver API and test double API, this section will be dedicated to the design of extensions and modifications of the test driver API. These extensions and modifications will be derived based on the functional requirements defined in 3.1. The API is designed based on existing CppUMock API, uses same or similar method and parameter names, which are not described in detail in this section, therefore for more details see the CppUMock documentation [67].

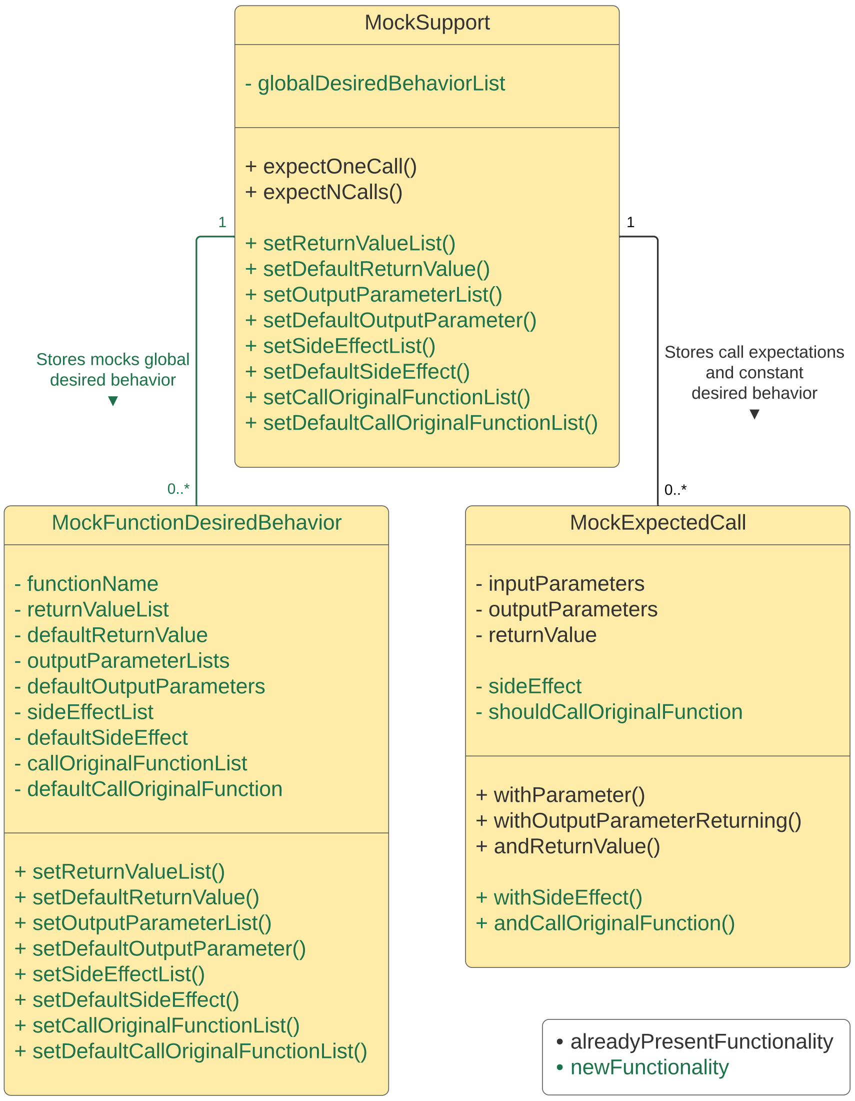
MockSupport
- globalDesiredBehaviorList
+ expectOneCall()
+ expectNCalls()
+ setReturnValueList()
+ setDefaultReturnValue()
+ setOutputParameterList()
+ setDefaultOutputParameter()
+ setSideEffectList()
+ setDefaultSideEffect()
+ setCallOriginalFunctionList()
+ setDefaultCallOriginalFunctionList()
1
Stores mocks global desired behavior ▼
0..*
1
Stores call expectations and constant desired behavior ▼
0..*
MockFunctionDesiredBehavior
- functionName
- returnValueList
- defaultReturnValue
- outputParameterLists
- defaultOutputParameters
- sideEffectList
- defaultSideEffect
- callOriginalFunctionList
- defaultCallOriginalFunction
+ setReturnValueList()
+ setDefaultReturnValue()
+ setOutputParameterList()
+ setDefaultOutputParameter()
+ setSideEffectList()
+ setDefaultSideEffect()
+ setCallOriginalFunctionList()
+ setDefaultCallOriginalFunctionList()
MockExpectedCall
- inputParameters
- outputParameters
- returnValue
- sideEffect
- shouldCallOriginalFunction
+ withParameter()
+ withOutputParameterReturning()
+ andReturnValue()
+ withSideEffect()
+ andCallOriginalFunction()
• alreadyPresentFunctionality
• newFunctionality

Figure 3.3: Test driver API domain diagram

The overall test driver API focused additions and their integration into the existing API are demonstrated in the domain diagram 3.3. In general, the `MockSupport` class provides an entry point for all CppUMock functionalities and stores all coherent data. `MockExpectedCall` stores expectations and desired behavior of individual test doubles calls, i.e. constant desired behavior, such as return value and parameters value, therefore it will be extended to support constant desired behavior of side effect calls and original function calls. Then a new class `MockFunctionDesiredBehavior` will be created, for storing global desired behavior, i.e. behavior for all calls of one function test double, such as key-value and default values. The classes demonstrated in 3.3 are just the most significant classes that will need to be extended and created, additional helper classes and functions will need to be also implemented.

Based on the requirement F_3 *key-value return value*, new methods shall be created to set desired key-value return or output value as defined in 3.3, where returned `ReturnValueDictionary` and `OutputParameterDictionary` will be initialized and empty, but will provide methods to fill them with key-value data. The `parameterName` attribute is the name of the parameter which values will be used as the key.

```
ReturnValueDictionary& MockSupport::setReturnValueList(const SimpleString& functionName,
    const SimpleString& parameterName);

OutputParameterDictionary& MockSupport::setOutputParameterList(const SimpleString& functionName,
    const SimpleString& outputParameterName, const SimpleString& keyParameterName);
```

Source code 3.3: Setting key-value return or output value API definition

Based on the requirement F_4 *default return value*, new methods shall be implemented to set desired default return or output value defined in 3.4, where `T` will be replaced with data type corresponding to the return value type.

```
void MockSupport::setDefaultReturnValue(const SimpleString& functionName, <T> value);

void MockSupport::setDefaultOutputParameter(const SimpleString &functionName,
    const SimpleString &outputParameterName, const void *value, size_t size);
```

Source code 3.4: Setting default return or output value API definition

Based on the requirement F_7 *original function call*, methods to set the test double to call the original function shall be created as defined in 3.5, where returned `ShallCallOrigFunctionDictionary` will be initialized and empty, but will provide methods to fill it with key-value data.

Based on the requirement F_10 *invoke side effects*, methods to pass a desired side effect invocation shall be provided as defined in 3.6, where returned

```
// call the original function for constant number of calls
MockExpectedCall& MockExpectedCall::andCallOriginalFunction(bool value);

// call the original function based on the provided list
ShallCallOrigFunctionDictionary& MockSupport::setCallOriginalFunctionList(
    const SimpleString& functionName, const SimpleString& parameterName);

// call the original function by default
void MockSupport::setDefaultCallOriginalFunction(const SimpleString& functionName, bool value);
```

Source code 3.5: Setting original function call API definition

`SideEffectDictionary` will be initialized and empty, but will provide methods to fill it with key-value data.

```
// invoke side effect for constant number of calls
MockExpectedCall& withSideEffect(void (*value)());

// invoke side effect based on a provided key-value list
SideEffectDictionary& setSideEffectList(const SimpleString& functionName,
    const SimpleString& parameterName);

// invoke default side effect
void setDefaultSideEffect(const SimpleString& functionName, void (*value)());
```

Source code 3.6: Setting test double to invoke a side effect API definition

Based on the requirement F_8 *throw an exception*, methods to set the test double to throw provided exception shall be implemented. However, passing an exception of unknown type to the framework is impossible without usage of templates or RTTI, which would violate the CppUTest framework principles and current design. Instead, the tester will have a possibility to wrap throwing of an exception into a function and pass this function as a side effect (defined in previous step) of the test double method. Therefore no additional methods need to be implemented for this purpose.

3.8 Test double API

As already mentioned, the CppUMock has the test double API, which is responsible mainly for determining test double desired behavior, while the purpose of the test driver API (designed in section 3.7) was to set the desired behavior. The design in this section will be derived mainly from the existing test double API, therefore for more details about the existing API see the CppUMock documentation [67].

An overview of the test double API new functionalities and modifications of the existing API are demonstrated in the domain diagram 3.4. Brief descriptions of classes present in the diagram `MockSupport`, `MockExpectedCall`, and `MockFunctionDesiredBehavior` were already provided in section 3.7. Class

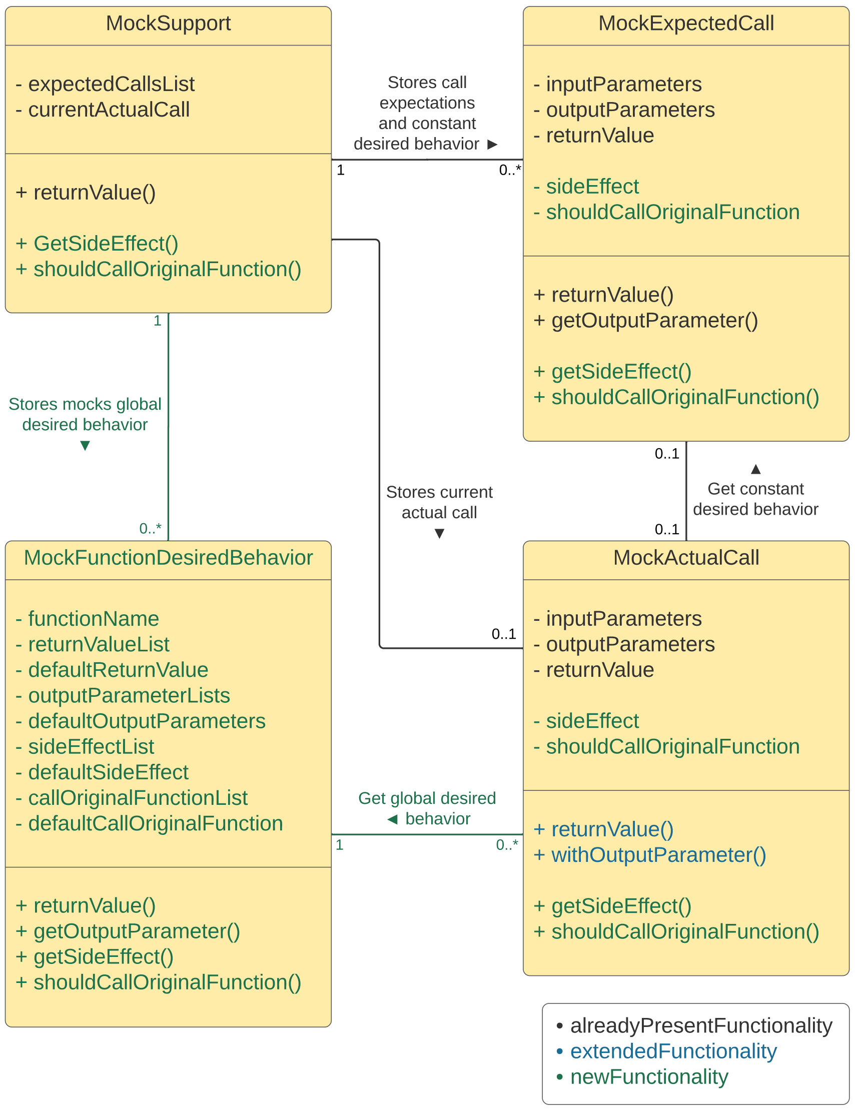

Figure 3.4: Test Double API domain diagram

`MockActualCall` provides methods to record the actual input parameters values passed to the test double and determine the desired behavior. Internally, the actual call is matched with a `MockExpectedCall` object based on expected sequence of the calls. Currently, the desired behavior is then determined based on the matched expected call. As part of the extension, the `MockActualCall` class will need to consider also the global desired behavior stored in `MockFunctionDesiredBehavior`, therefore modification in some methods will be implemented. And also methods for determining newly added desired behavior will be created – side effect call and original function call.

The CppUMock test double API already provides a set of methods to determine desired return value and output parameters values, but it does not consider the newly added global desired behavior, such as key-value list determined values and default values, therefore the existing methods shall be modified to reflect the new functionalities in their behavior as already defined in the section dedicated to desired behavior determination 3.4. The changes shall be applied to the methods listed in 3.7 (see CppUMock documentation [67] for more detail), but their original features shall be preserved. Methods to get the return value of specific type, such as `MockActualCall::returnIntValue()`, internally use the `MockActualCall::returnValue()` method, therefore their modification is not needed.

```
MockNamedValue MockActualCall::returnValue();
MockActualCall& MockActualCall::withOutputParameter(const SimpleString& name, void* output);
MockActualCall& MockActualCall::withOutputParameterOfType(const SimpleString& typeName,
    const SimpleString& name, void* output);
```

Source code 3.7: Determining test double return value API

Currently the CppUMock assumes that the test double already has information which parameters are input and which are output, since they were until now implemented manually by the tester. But in the test double created with the extension, it shall be determined dynamically, therefore a new method defined in 3.8 will be added, which shall only check whether the passed parameter name has registered desired output value.

```
bool MockActualCall::isParameterOutput(const SimpleString& name);
```

Source code 3.8: Determining test double parameter type (input/output) API definition

Corresponding to the requirement F_7 *original function call*, the test double API shall implement a method to determine whether original function shall be called, as defined in 3.9. The original function reference shall be already passed to the test double through the test double creation API defined in 3.6.

```
bool MockActualCall::shouldCallOriginalFunction();
bool MockSupport::shouldCallOriginalFunction();
```

Source code 3.9: Determining original function call desired behavior API definition

Corresponding to the requirement F_10 *invoke side effects*, the test double API shall implement a method to determine whether a side effect (a function, lamba function or functor) should be invoked from the test double as defined in 3.10, which shall return the set side effect to the test object. If no side effect was set, the defined method shall return a null reference and the test double shall invoke no side effects.

```
void (*MockCheckedActualCall::getSideEffect())();
void (*MockSupport::getSideEffect())();
```

Source code 3.10: Determining desired behavior regarding invoking a side effect API definition

Chapter 4

Implementation of the CppUTest extension

This chapter provides additional information about the implementation. The extension was implemented based on the design in chapter 3 and no design changes were made. This chapter provides a manual, a requirements fulfillment overview, and a brief description of unit tests. For more technical detail regarding the implementation, see the doxygen[7] documentation in the source codes delivered with this thesis.

4.1 Manual

The extended framework delivered with this thesis supports all functionalities provided in the original CppUTest framework and their usage is unchanged, therefore see the official CppUTest manual at [66] and CppUMock manual at [67] for its usage. The aim of this section is to demonstrate usage of the new functionalities in the original CppUTest manual style.

4.1.1 Return value

The extension provides two new ways of setting a return value. The return value can be selected based on a key-value list, where the key corresponds to one of the parameters, i.e. if the value of the selected parameter of the current actual call is equal to one of the key-value entries of the list, the corresponding value is selected as the return value. The key-value return value list can be set as demonstrated in 4.1.

[7]"Doxygen is the de facto standard tool for generating documentation from annotated C++ sources" [71]

```
mock().setReturnValueList("function", "key_parameter")
    .key(42).value(7581)
    .key(12).value(123);
```

Source code 4.1: Setting test double return value list

Alternatively a default return value can be set from the test driver API as shown in 4.2. This value is then returned by the test double if no other return value is set.

```
mock().setDefaultReturnValue("function", 7581);
```

Source code 4.2: Setting test double default return value

The test double can still get the return value as before, i.e. int return value can be obtained via `mock().returnIntValue()` call or directly via returned actual call reference `mock().actualCall("function").returnIntValue()`. If more desired return value options are set and applicatble for the current call, the original (constant) return value has the highest priority, then the key-value list return value, and lastly the default return value has the lowest priority.

4.1.2 Output parameters

Analogously to the return value, the extension provides methods to set either key-value list or default output parameter value, the only difference is that the name of the output parameter, reference of the value, and its size must be provided, as demonstrated in 4.3.

```
int desired_value = 7581;
// set key-value list
mock().setOutputParameterList("function", "output_parameter", "key_parameter")
    .key(42).value(&desired_value, sizeof(desired_value))
    .key(12).value(&desired_value, sizeof(desired_value));
// set default value
mock().setDefaultOutputParameter("function", "output_parameter", &desired_value,
    sizeof(desired_value));
```

Source code 4.3: Setting test double output parameter

The priority of desired behavior is also analogous to the return value, i.e. the constant value has the highest priority, then the key-value list value, and lastly the default value has the lowest priority. Also objects are supported, simply by using alternatives `valueOfType("ObjectType", &desired_value)` and `setDefaultOutputParameterOfType()`.

4.1.3 Side effects invocation

The test double might be also set to invoke a side effect, where the side effect can be either a function, method, lambda function, or functor, the only condition is that it can be invoked and passed as `void (*)()` type value, hence, the side effect cannot accept any parameters and should not return any value.

The setting of desired side effect invocations is demonstrated in 4.4, where reference to a function is passed as the side effect in all cases. The side effect desired behavior can be again set for constant number of calls, as key-value list, or default value, with priority corresponding to the usage order in the example, i.e. same as in the previous cases.

```
// the side effect definition
void sideEffect() {
    std::cout << "Side effect ivoked!" << std::endl;
}

TEST(Example, SideEffect)
{
    // expect one call with constantly set side effect
    mock().expectOneCall("function").withSideEffect(&sideEffect);
    // set key-value side effect list
    mock().setSideEffectList("function", "key_parameter")
        .key(7581).value(&sideEffect);
    // set default side effect
    mock().setDefaultSideEffect("function", &sideEffect);
    // set more call expectations and invoke test object here
}
```

Source code 4.4: Setting desired side effect usage

The test double can then get the desired side effect and invoke it as shown in 4.5. If no side effect is desired, **`nullptr`** is returned. The desired side effect can be also obtained via the `mock().getSideEffect()` call (after registering the actual call).

```
void function() {
    // register the actual call and get the side effect
    void (*side_effect)() = mock().actualCall("function").getSideEffect();
    // invoke the side effect
    sideEffect();
}
```

Source code 4.5: Getting the desired side effect usage

4.1.4 Original function call

The test double can be set to call the original function (or method). This might be useful in tests designed to just observe the passed parameters or when the test object is able to detect whether the original function has been really called.

Example in 4.6 demonstrates how the test double can be set to call the original function, which can be performed either for constant number of calls, as key-value list, or as default value. Test double can get the desired behavior by calling `shouldCallOriginalFunction()` (returning `bool`) on the actual call reference or after the `mock()` call. If no corresponding desired behavior was set, `false` is returned by default. Otherwise the constant setting has the highest priority, then the key-value list, and lastly the default value.

```
// expect one call with desired behavior to not call the original function
mock().expectOneCall("function").andCallOriginalFunction(false);
// set key-value list
mock().setCallOriginalFunctionList("function", "key_parameter")
    .key(7581).value(false);
// set the test double to call the original function by default
mock().setDefaultCallOriginalFunction("function", true);
```

Source code 4.6: Setting original function call usage

The APIs provide explicit methods only for determining whether the original function shall be called by the test double, but no explicit method for passing and storing the original function reference is provided. The reference can be either manually stored in the test double implementation or passed via the `mock().setData()` call (see the CppUTest manual [67] for usage details).

4.1.5 Test double creation

The extension provides a tool for test doubles creation. It automates usage of CppUMock supported desired behavior processing with minimal effort from the tester. This is achieved by creating a set of universal methods, which the tester can call from the test double to delegate its functionality. This delegated function gradually automatically performs the following steps:

1. Registers the test double actual call via the `mock().actualCall()` call.

2. Registers values of the passed parameters via the `withParameter()` method call. It registers all passed parameters, including potentially output parameters, therefore either expected values for all parameters shall be set via the test driver API, or `ignoreOtherCalls()` shall be called for the corresponding expected call, in case no expected values of output parameters are provided.

3. Via `shouldCallOriginalFunction()` call, it determines whether the original function is desired to be called. If so, the original function

is called, the parameters are passed without changing, and its output parameters values and return value are stored.

4. Fetches desired behavior regarding the output parameters. That means if a desired value is set, e.g. via test driver API `withOutputParameter()` method call for a parameter, it is considered an output parameter and the corresponding desired value is applied to it. This can override the output parameter value set in the previous step via the original function call.

5. Determines whether a side effect should be invoked via the test double API `getSideEffect()` method call. If so, the returned side effect is invoked.

   Hint: The side effect feature can be also used to throw an exception from the test double by wrapping the throwing of the exception into a function, for example.

6. If the test double has a return value, its return value is fetched via the corresponding test double API call (e.g. `returnIntValue()`). If no desired return value was set via the test driver API, but the original function was called, its returned value is returned. If none desired return value is found, a corresponding test fail is invoked due to the undetermined return value.

```
int function(int input, CustomClass &output) {
    const char* parameter_names[] = {"input", "output"};
    const char* parameter_types[] = {"", "CustomClass"};
    return mockReturningInt("function", parameter_names, parameter_types, &input, &output);
}
```

Source code 4.7: Test double creation support usage

An example usage of test double returning `int` is demonstrated in 4.7. The `mockReturningInt()` accepts the following parameters:

- `method_name` – name of the method/function the test double replaces.
- `args_names` – array of C-strings of the parameter names. The names are used for registering values of the parameters provided by the test object in the call and determining desired behavior.
- `args_types` – array of C-strings of names of types of the parameters. The names shall be provided only for data types unsupported by CppUTest (typically user defined classes). For CppUTest supported data types parameters, the values shall be empty C-strings, where the supported data types are `bool`, `int`, `unsigned int`, `long int`, `unsigned`

`long int`, `long long`, `unsigned long long`, `double` (its checks precision can be controlled from the test driver API), `const char*`, `void*`, `const void*` and `void (*)()`.

- `Types... arg` – references to the passed parameters of the test double function/method (e.g. `&input` and `&output` in the example 4.7).

The demonstrated example shows usage of test double returning `int`, however other functions with other return values are also provided as listed below:

- `void mockWithoutReturnValue()`
- `bool mockReturningBool()`
- `int mockReturningInt()`
- `unsigned int mockReturningUnsignedInt()`
- `long int mockReturningLongInt()`
- `unsigned long int mockReturningUnsignedLongInt()`
- `long long mockReturningLongLongInt()`
- `unsigned long long mockReturningUnsignedLongLongInt()`
- `double mockReturningDouble()`
- `const char* mockReturningStringValue()`
- `void* mockReturningPointerValue()`
- `const void* mockReturningConstPointerValue()`
- `void (*mockReturningFunctionPointerValue())()`

The extension provides a set of methods to set the test double to call the original function, however, the original function reference needs to be provided directly to the test double. The functions above do not support it, another set of functions is therefore supported for this purpose, that additionally accept reference to the original function as a parameter, as demonstrated in 4.8, where the passed `orig_function` contains the reference to the original function.

```
mockReturningInt("function", parameter_names, parameter_types, orig_function, &input, &output);
```

Source code 4.8: Test double with original function reference creation support usage

Again, functions with different return values accepting original function reference are provided as listed below:

- `void mockWithoutReturnValueWithOriginalFunction()`
- `bool mockReturningBoolWithOriginalFunction()`

- `int mockReturningIntWithOriginalFunction()`
- `unsigned int mockReturningUnsignedIntWithOriginalFunction()`
- `long int mockReturningLongIntWithOriginalFunction()`
- `unsigned long int mockReturningUnsignedLongIntWithOriginalFunction()`
- `long long mockReturningLongLongIntWithOriginalFunction()`
- `unsigned long long mockReturningUnsignedLongLongIntWithOriginalFunction()`
- `double mockReturningDoubleWithOriginalFunction()`
- `const char* mockReturningStringValueWithOriginalFunction()`
- `void* mockReturningPointerValueWithOriginalFunction()`
- `const void* mockReturningConstPointerValueWithOriginalFunction()`
- `void (*mockReturningFunctionPointerValueWithOriginalFunction())()`

4.2 Unit tests

The implementation includes a set of unit tests focused on the new features. Since testing of the extension with 100 % coverage is out of scope of this work, the added tests cover just the basic usages of the new features, e.g. the key-value list return value is tested, but not for all return value and parameters supported data types. Overall, 31 unit tests were added.

The implemented tests have one very specific characteristic, since they actually test a testing framework. They are implemented using the CppUTest itself, using its test macros, test driver API, and test double API. But the roles of the involved actors are exchanged. The test implementation itself remains in the role of the test driver, but the testing framework is in this case in the role of the test object. And for simplicity, the CppUTest test double API is also accessed via the test driver.

```
TEST(MockReturnValueTest, DefaulReturnValue)
{
    const char* function_name = "TestFunction";
    mock().setDefaultReturnValue(function_name, 7581);
    mock().expectOneCall(function_name);

    LONGS_EQUAL(7581, mock().actualCall(function_name).returnIntValue());
    LONGS_EQUAL(7581, mock().intReturnValue());
}
```

Source code 4.9: Example default return value unit test

One of the unit tests is demonstrated in 4.9. This test checks that when a default return value is set, the framework propagates this setting correctly, which is achieved by setting it via the test driver API and checking the return value obtained via the test double API.

4.3 Requirements fulfillment

Requirement	Corresponding implementation
F_1 Automatic actual call	All test double creation API functions automatically register the actual call.
F_2 Constant return value	• Return value via `andReturnValue()` method call on expected call instance(s). • Output parameters via `withOutputParameter()` and `withOutputParameterOfType()` method call on expected call instance(s).
F_3 Key-value return value	`mock().setReturnValueList()`
F_4 Default return value	`mock().setDefaultReturnValue()`
F_5 Return values priority	`returnValue()` and `withOutputParameter()` methods of the actual call consider the values priority as required.
F_6 Return value not set	All test double creation API functions with return value fail in case the desired return value is not set.
F_7 Original function call	• `andCallOriginalFunction()` method call on expected call instance(s) to set constant desired behavior. • `mock().setCallOriginalFunctionList()` call to set key-value list desired behavior. • `mock().setDefaultCallOriginalFunction()` call to set default desired behavior. • `shouldCallOriginalFunction()` call on the actual call instance retrieves the desired behavior and considers the priorities as required.
F_8 Throw an exception	Test double can be set to throw an exception via the side effect feature (see below).
F_9 Parameters automatic registration	All test double creation API functions automatically register values of all provided parameters.
F_10 Invoke side effects	• `withSideEffect()` method call on expected call instance(s) to set constant desired behavior. • `mock().setSideEffectList()` call to set key-value list desired behavior. • `mock().setDefaultSideEffect()` call to set default desired behavior. • `getSideEffect()` call on the actual call instance retrieves the desired behavior and considers the priorities as required.

Table 4.1: Functional requirements fulfillment

An overview of the functional requirements and their corresponding fulfillment is demonstrated in the table 4.1. All the demonstrated requirements are fulfilled with the listed implementation methods.

Requirement	Fulfillment
N_1 Simple design and usage	This requirement is not well measurable, however the extension does not introduce any new restrictions into the CppUTest framework usage and does not use any new constraints.
N_2 No code generation	The extension does not use code generation, the test double creation API functions are easily combinable with manual test doubles.
N_3 No complicated macros	No new macros were introduced.
N_4 C++ language	The extension is implemented exclusively in C++.
N_5 CppUTest compatibility	The original behavior of CppUTest was not changed, which is supported by passing unit tests (part of the original implementation).

Table 4.2: Non-functional requirements fulfillment

An overview of the non-functional requirements and their corresponding fulfillment is demonstrated in table 4.2. Since some of these requirements are quite vague and hardly measurable, their fulfillment determination is not trivial. However, no violations of the stated requirements were identified and are overall considered fulfilled.

Chapter **5**

ETCS simulator

This chapter analyses the project of the ETCS[8] simulator of the Faculty of Transportion of CTU partly developed at FIT CTU. The selected modules of this chapter shall be tested in this thesis. But before analysing the project itself and its modules, basic concepts and terms will be explained.

5.1 ERTMS

The European Rail Traffic Management System (further just ERTMS) is an EU project to unify the European railways. The initial motivation for this project was the incompatibility of railway infrastructure in Europe (in the past and partially still in the present). The aim of ERTMS is to create a standard for the whole European railway infrastructure and therefore also enhance competitiveness of the European rail sector.

ERMTS consists of two components – Global System for Mobiles - Railway (further referred to as GSM-R) and European Train Control System (further referred to as ETCS).

GSM-R – the radio communication element for both data and voice communication between the train and the track. It is based on the public standard GSM, extended with specific functions, and using frequencies specifically reserved for rail applications.

ETCS – the signalling element for the train. It is a standard for signalling into the driver cabin via displaying information in less complex manner on the onboard display. Simultaneously it monitors the train movement and together ensures the train operates safely. [72] [73]

[8]European Train Control System

The ERTMS tracks can be operated in three different signalling levels. The individual levels are described below:

ERTMS Level 1 – on this level, the train communicates with eurobalises that are placed on the track. The eurobalises (also called balises) are electronic devices placed on the track that track the train position and transmit control information to the train, such as information about the incoming passages with speed limit.

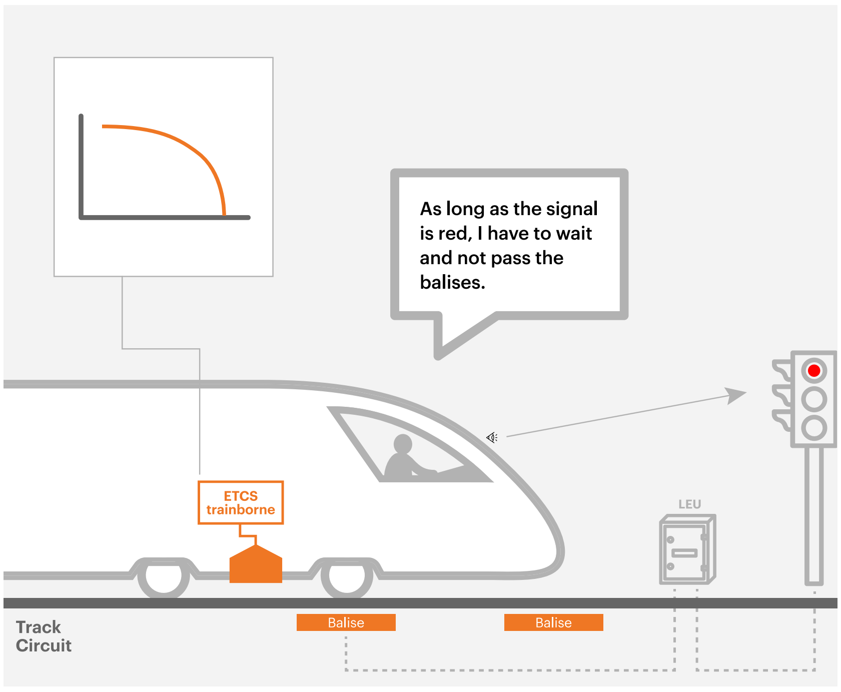
As long as the signal is red, I have to wait and not pass the balises.
ETCS trainborne
LEU
Track Circuit
Balise
Balise

Figure 5.1: ERTMS Level 1 with balise without infill [74]

This level can be provided either with just eurobalises, or eurobalises with infill. The alternative without the infill is demonstrated in the diagram 5.1. The version with the eurobalises radio infill in diagram 5.2 transmits the data corresponding to the individual eurobalises in advance with respect to its location. Therefore the train can obtain the message to pass the signal before reaching the balises once the signal is green. [74] [75]

ERTMS Level 2 – on this level, eurobalises are used only to communicate fixed messages, such as location reference and length of the approaching section. The rest of the communication is performed via the GSM-R between the Centralized Block Centre and the trains. The diagram for the level 2 is demonstrated in 5.3. [74] [75]

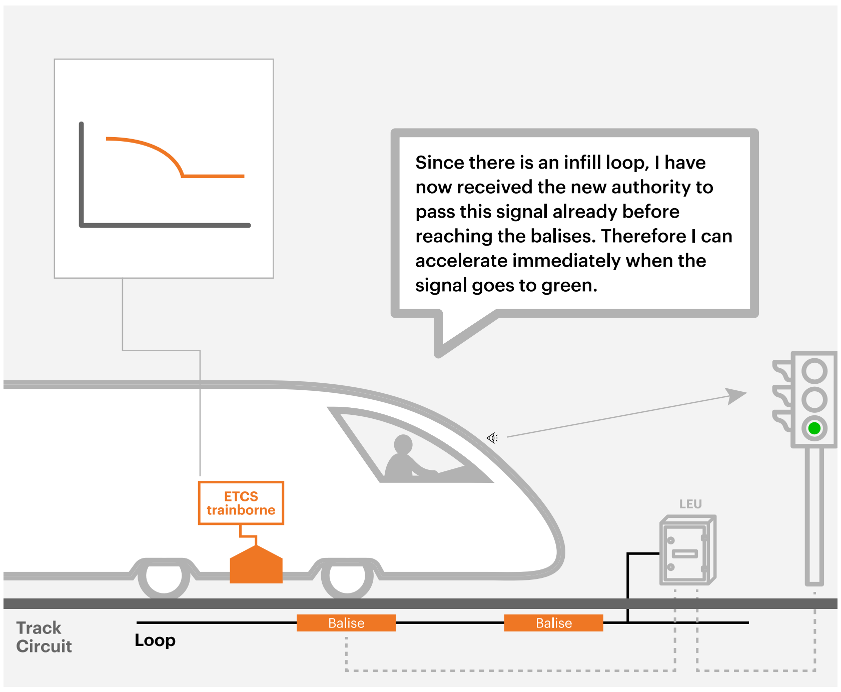
Since there is an infill loop, I have now received the new authority to pass this signal already before reaching the balises. Therefore I can accelerate immediately when the signal goes to green.
ETCS trainborne
LEU
Track Circuit
Loop
Balise
Balise

Figure 5.2: ERTMS Level 1 with balise with infill [74]

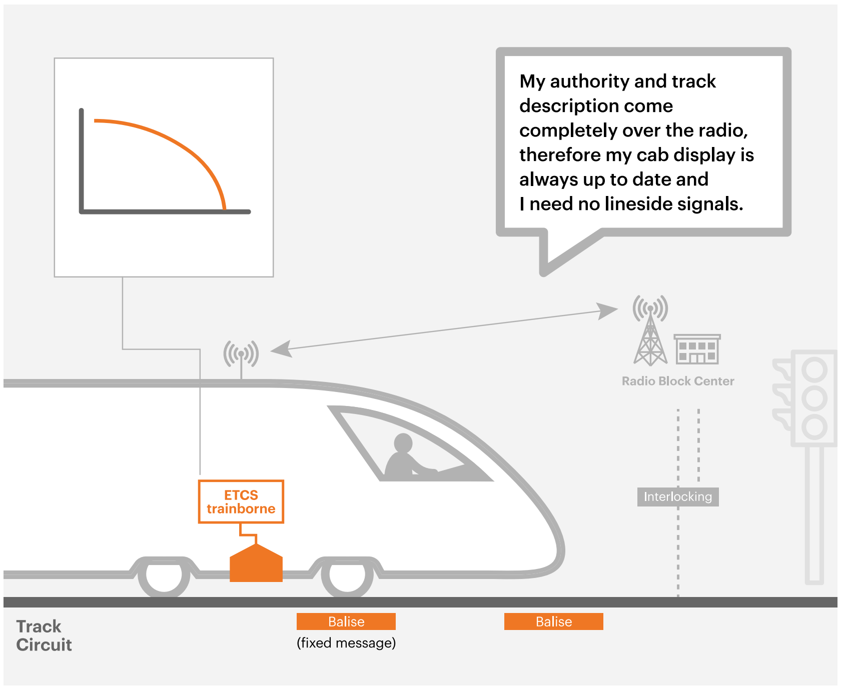
My authority and track description come completely over the radio, therefore my cab display is always up to date and I need no lineside signals.
Radio Block Center
Interlocking
ETCS trainborne
Track Circuit
Balise
(fixed message)
Balise

Figure 5.3: ERTMS Level 2 [74]

ERTMS Level 3 – the tracks satisfying the ERTMS level 2 and not using any other signalling system, therefore being controlled solely through the ERTMS, are considered ERTMS level 3. On this level, the trains additionally have to report the trains integrity on itself. Diagram for this level is demonstrated in 5.4. [74] [75]

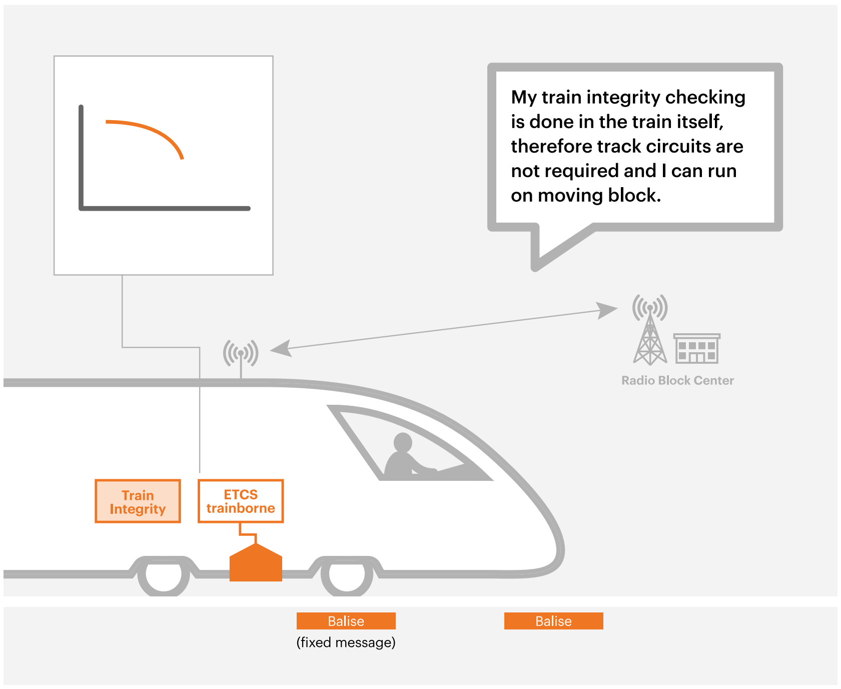
My train integrity checking is done in the train itself, therefore track circuits are not required and I can run on moving block.
Radio Block Center
Train Integrity
ETCS trainborne
Balise
(fixed message)
Balise

Figure 5.4: ERTMS Level 3 [74]

5.2 ETCS simulator

The ETCS simulator that is developed at FIT CTU and FTS CTU aims to simulate the ETCS system, e.g. for training the train drivers. The simulator shall be implemented in compliance with the ETCS specification for the level 1 and level 2. The simulator includes several modules, but just a few will be subject of this thesis. The following modules will be tested in this thesis:

DMI (Driver Machine Interface) is a module that ensures communication between the train driver and the ETCS system. The DMI receives data from the other ETCS modules and displays them in standard way (according to the unified ETCS specification). For more details, see [76].

EVC (European Vital Computer) module is the core of the ETCS simulator. This module receives data from other modules (e.g. RBC), communicates them to the DMI module and writes data to the JRU. For more details see [77].

RBC (Radio Block Centre) module mainly calculates and communicates information regarding permission for movement to the train based on the information about the train route. It ensures safe movement of the train on the track. For more details, see [78].

Braking curve module shall calculate the breaking curve based on data received from the EVC. For more details, see [79].

JRU (Juridical Recording Unit) module responsibility is to receive data from the EVC, their preservation and visualisation. The preserved data shall be downloadable to an external device. For more details, see [80].

Chapter 6

Testing of the ETCS simulator

This chapter is dedicated to testing of the ETCS simulator introduced in chapter 5.2. Only some of the ETCS simulator modules are developed at FIT CTU and subject of this work, therefore only component level tests will be implemented in this thesis, since higher level tests would focus on the whole system, in this case the ETCS simulator, and the lower level, unit tests, is responsibility of the developers and do not bring any validation value.

6.1 Testing utilities

Tests focusing on one test object typically share the same approach to invoking the test object and verifying its behavior. Therefore before implementing the tests themselves, a helper library was introduced, called testing utilities. This library includes set of functions and classes useful for testing all of the modules. This provides the following functionalities:

Process – is a class that enables running a new separate process and then terminating it. It provides sets of methods to invoke an executable in a separate process based on the provided path (either relative or absolute) in selected working directory, check if the corresponding process is still running, and terminate the process. This is useful for executing the test object and restarting it for each test.

ETCS Messages – defines a set of messages used by the ETCS modules. The messages are defined according to the official ETCS set of specifications, specifically SUBSET-026-7 [81] and SUBSET-026-8 [82].

Constants – a set of constants used by the tested modules and their corresponding tests. This includes special values defined in SUBSET-026-7 [81], special values used in the ETCS simulator demo, and message attributes names.

6.2 MQTT broker

The individual ETCS simulator modules communicate via a MQTT[9] broker Eclipse Mosquitto [83], which is a third party software that is generally considered reliable and already tested. Also there were no requirements that would require detailed testing of the communication between the individual modules and the MQTT broker. These facts were used for setting up the test environment, where this very broker was used for communication between the test driver and the test object as displayed in diagram 6.1.

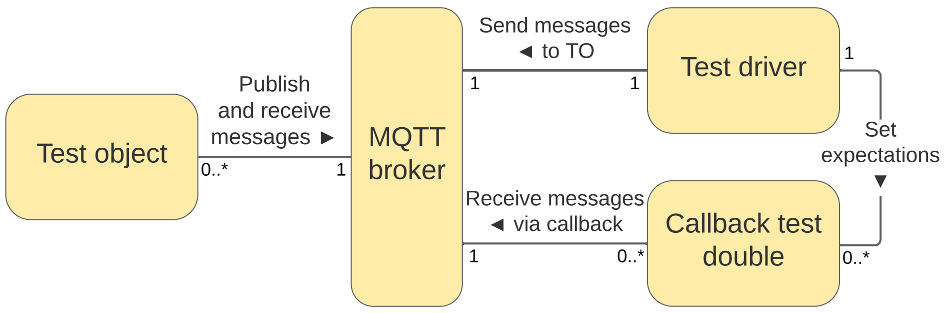

Figure 6.1: Tests using MQTT broker domain diagram

6.3 General tests structure

The implemented tests are divided into individual test groups, where test group is a set of tests focused on one feature, e.g. communication session initiation. The test group typically defines setup and teardown actions that are same for all tests in the same test group.

Each test group and test have a corresponding doxygen documentation describing their purpose and individual steps. The provided doxygen documentation uses a set of tags to ensure better readability and maintainable structure. The following tags are used:

- **@brief** – denotes brief description of the test or test group. Corresponding text explains the general idea and verification purpose, but does not describe the individual steps of how it is achieved.
- **@action** – denotes individual steps performed by the test driver, e.g. description of the message sent to the test object.
- **@expected** – describes a specific expectation and how it is verified by the test driver, e.g. that a specific message from test object is received via the MQTT client callback.

[9]MQ Telemetry Transport is publish-subscribe network protocol

- **@note** – does not correspond to any action or expectation of the test, but just explains some information or context that is vital for understanding of the test.

6.4 RBC tests

This section is dedicated to tests of module RBC, which is responsible for communicating track information to the train to ensure its safe movement. At first, the test environment where the tests are evaluated will be described, then the individual dynamic tests will be listed.

Usually, tests are designed based on the requirements using techniques described in chapter 1. Although in this case, only requirements on rather a system level were provided, which did not provide enough details for most of the component level tests, however, the official ETCS specification provides detailed description of the messages exchanged between the RBC and EVC module in SUBSET-026-7 [81], SUBSET-026-8 [82], and SUBSET-026-3 [84], which were partially used in the ETCS simulator. Therefore the tests were designed mainly using the exploratory testing technique based on the ETCS specification, stated system level requirements, and brief documentation of the RBC module, which might result in incompleteness of the overall test set.

6.4.1 Test Environment

As already mentioned, MQTT broker is used for communication with the test object, in this case the RBC module. RBC is an independently running process, therefore the *Process* class from testing utilities is used for its invocation and termination for each test.

Also testing tools useful specifically for the RBC focused test groups were implemented. To handle the messages coming from the test object, test doubles corresponding to the individual messages were implemented, e.g. if message *Configuration Determination* (number 32 in SUBSET-026-8 [82]) was received, it would be passed to the corresponding message 32 callback function.

This is achieved by implementing a universal callback function that redistributes the messages to their individual message callback functions. The callback function is part of another helper class *BaseCase* which provides functionalities for setting up the test environment and test object.

6.4.2 Session initiation test group

This test group focuses on communication initiation between the EVC and RBC, specifically exchange of messages *Initiation of a communication session* number 155 (sent from EVC) and *Configuration Determination* number 32 (sent from RBC).

The list of following tests was implemented and evaluated. Only their *brief* description is provided, for more detailed description and individual test steps, see the doxygen documentation.

- **Title: init_success**

  Description: Checks that RBC successfully accepts message from EVC for ”Initiation of a communication session” (155) and responds with message ”Configuration Determination” (32) with correctly set data.

  Result: OK

- **Title: multiple_inits_same_train**

  Description: Checks that RBC successfully accepts the first message from EVC for ”Initiation of a communication session” (155) and responds with message ”Configuration Determination” (32) with correctly set data, but ignores all subsequent ”Initiation of a communication session” messages from EVC for the same train.

  Result: OK

- **Title: init_multiple_sessions**

  Description: Checks that RBC successfully accepts multiple messages ”Initiation of a communication session” (155) from EVC for different trains and responds with message ”Configuration Determination” (32) with correctly set data for each.

  Result: OK

- **Title: init_time_incrementation**

  Description: Checks that the RBC always responds with train time ”T_TRAIN” attribute in the message ”RBC/RIU System Version” (32) higher than in the received message ”Initiation of a communication session” (155) from EVC.

  Result: OK

6.4.3 Train data acknowledgement test group

This test group focuses on train data acknowledgement by the RBC, corresponding to the train data sent by the EVC. The list of following tests was implemented and evaluated. Only their *brief* description is provided, for more detailed description and individual test steps, see the doxygen documentation.

- **Title: acknowledgement_success**

  Description: Checks that RBC successfully accepts messages ”Session established” (159), ”SoM Position Report” (157), and ”Validated Train

Data" (129) from EVC and responds with message "Acknowledgement of Train Data" (8) with correctly set data.

Result: OK

- **Title: acknowledgement_unknown_train**

  Description: Checks that RBC does not respond to messages "Session established" (159), "SoM Position Report" (157), and "Validated Train Data" (129) from EVC, when the train (NID_ENGINE) provided in this messages is unknown to RBC, i.e. the session was not initiated with this train.

  Result: OK

6.4.4 Session termination test group

This test group focuses on termination of communication session between the EVC and RBC. The list of following tests was implemented and evaluated. Only their *brief* description is provided, for more detailed description and individual test steps, see the doxygen documentation.

- **Title: termination_success**

  Description: Checks that RBC successfully accepts message from EVC for "Termination of a communication session" (156) for an already established session and responds with message "Acknowledgement of termination of a communication session" (39) with correctly set data.

  Result: OK

- **Title: termination_unknown_train**

  Description: Checks that RBC ignores message "Termination of a communication session" (156) for a session that is not established, i.e. the NID_ENGINE sent in the message does not have a corresponding initiated and confirmed session.

  Result: OK

- **Title: termination_on_mission**

  Description: Checks that RBC ignores message "Termination of a communication session" (156) for a train that is in state ON MISSION and does not respond with message "Acknowledgement of termination of a communication session" (39).

  Result: OK

6.4.5 Movement authority request test group

This test group focuses on MA[10] requests between RBC and EVC. The list of following tests was implemented and evaluated. Only their *brief* description is provided, for more detailed description and individual test steps, see the doxygen documentation.

- **Title: MA__request__success**

  Description: Checks that RBC successfully accepts message ”MA Request” (132) from EVC and responds with message ”Movement Authority” (3) with correctly set data, including Packets ”Level 2/3 Movement Authority” (15), ”Gradient Profile” (21) and ”Position Report Parameters” (58).

  Result: OK

- **Title: MA__request__unknown__train**

  Description: Checks that RBC does not respond to message ”MA Request” (132) from EVC, when the train (NID_ENGINE) provided in this message is unknown to RBC, i.e. the session was not initiated with this train.

  Result: OK

- **Title: position__report__success**

  Description: Checks that RBC successfully accepts message ”Train Position Report” (136) from EVC and responds with message ”Movement Authority” (3) with correctly set data, including Packet ”Level 2/3 Movement Authority” (15), when valid message ”MA Request” (132) has been previously sent to the RBC.

  Result: OK

- **Title: position__report__without__MA**

  Description: Checks that RBC does not accept message ”Train Position Report” (136) from EVC and does not respond with message ”Movement Authority” (3) when no message ”MA Request” (132) has been previously sent to the RBC.

  Result: OK

6.4.6 End of mission test group

This test group focuses on end of mission communicated via session between the EVC and RBC. The list of following tests was implemented and evaluated.

[10] Movement authority

Only their *brief* description is provided, for more detailed description and individual test steps, see the doxygen documentation.

- **Title: eom_success**

  Description: Checks that RBC successfully accepts message from EVC for ”End of Mission” (150) for an already established session for train in state ”ON MISSION” and responds with message ”Acknowledgement of termination of a communication session” (39) with correctly set data.

  Result: OK

- **Title: eom_not_on_mission**

  Description: Checks that RBC does not accept message from EVC for ”End of Mission” (150) for an already established session for train that is not in state ”ON MISSION” and does not respond with message ”Acknowledgement of termination of a communication session” (39).

  Result: OK

- **Title: eom_message_duplicate**

  Description: Checks that RBC does not accept a duplicate message from EVC for ”End of Mission” (150) for train that was on mission, but the mission was already ended by corresponding message ”End of Mission” (150).

  Result: OK

6.5 Future work

In conclusion, only set of tests focused on the RBC component was implemented. The implemented test scenarios were picked mainly using the exploratory testing technique, therefore if possible future development will bring new requirements, new tests will be required as well. However the developed tests and testing utilities were designed to be easily extensible and reusable in the future, especially the testing utilities will hopefully prove very useful in the future development, since it should resolve most of the issues of developing new test environments for the ETCS project components.

The source codes delivered with this thesis contain documentation of the testing utilities and other provided test environment tools. Also brief notes for developing new tests are provided.

6.6 Evaluation of the CppUTest extension usage

The CppUTest extension was used to implement the individual messages callbacks, by passing the received values to the *testDouble()* function. This function then registered the received parameters values and checked the desired behavior, as described in previous chapters.

Unfortunately test doubles demanded by the tests in this chapter were quite simple, and their usage of test doubles was very straight forward, since the main responsibility of the test doubles was to just register the provided values for their verification and no more complex functionalities were necessary. Therefore there was not an opportunity to use much of the newly introduced features.

Although what proved to be practical was the functionality delegation to the *testDouble()* functions instead of replacing the called functions entirely. This was useful since the data provided by the MQTT broker on callback were wrapped in MQTT structures whose values would be hard to verify if they were passed to the CppUTest framework in their original form. In this case the partially manually implemented test doubles firstly unwrapped the provided data, and then delegated them to the *testDouble()* function in a verifiable form.

Chapter 7

Conclusions

The main goal of this thesis was to extend the existing framework CppUTest so it would provide better support for testing with mocks. Before design of the extension itself, analysis of testing approaches and techniques was performed and summary of the basics was created. The next step was examination of other existing C++ testing frameworks where different design approaches where compared.

The information collected in the performed analysis were used to design new features for the CppUTest extension. This included adding an option to set a desired side effect or original function call and introducing new approach to setting desired behavior in general, either by setting default values from test driver or creating a key-value map, which determines the corresponding desired behavior based on the provided parameters values. Another new designed feature was a universal test double that would automatically register the parameters values and evaluate the preset desired behavior.

The designed features were successfully implemented and integrated into the CppUTest framework. The unit tests already included in the framework are passing, proving the original functionality was preserved, and new unit tests focused on the new features were added.

The last step of this thesis was to create a set of tests for the ETCS simulator modules. Due to big scope of the other tasks in this work and spare documentation of some of the ETCS simulator modules, the tests were not developed for all the modules. Instead, however, it was decided to focus on the modules that proved to be more failure-prone in the past. Thus set of tests focused on the RBC module was created together with testing utilities, providing support for possible development of other tests and promising better extensibility in the future.

Bibliography

1. NAIK, Kshirasagar. *Software testing and quality assurance.* Hoboken, New Jersey: John Wiley & Sons, Inc., 2008. No. 2008008331.
2. OLSEN, Klaus; POSTHUMA, Meile; ULRICH, Stephanie. *ISTQB CTFL Syllabus 2018 V3.1.* 2018. Available also from: `https://www.istqb.org/downloads/send/2-foundation-level-documents/281-istqb-ctfl-syllabus-2018-v3-1.html`. [cit. 2021-4-3].
3. Goals Of Software Testing. *Software testing tutorials and automation* [online]. [N.d.]. Available also from: `https://www.software-testing-tutorials-automation.com/2018/02/goals-of-software-testing.html`. [cit. 2021-4-3].
4. *Difference between Verification and Validation* [online]. 2013. Available also from: `https://www.softwaretestingclass.com/difference-between-verification-and-validation/`. [cit. 2021-4-4].
5. IEEE Standard for System and Software Verification and Validation. *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004).* 2012, pp. 1–223. Available from DOI: `10.1109/IEEESTD.2012.6204026`.
6. ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E).* 2010, pp. 1–418. Available from DOI: `10.1109/IEEESTD.2010.5733835`.
7. COPELAND, Lee. *A Practitioner's Guide to Software Test Design.* Artech House, 2004. No. 158053791x.
8. LIN, Nai-Wei. *Control Flow Testing* [`https://www.cs.ccu.edu.tw/~naiwei/cs5812/st4.pdf`]. [N.d.]. [cit 2021-07-06].
9. LIN, Nai-Wei. *Data Flow Testing* [`https://www.cs.ccu.edu.tw/~naiwei/cs5812/st5.pdf`]. [N.d.]. [cit 2021-07-09].

10. *Difference between Component and Unit Testing.* 2019. Available also from: `https://www.geeksforgeeks.org/difference-between-component-and-unit-testing/`. [cit. 2021-4-11].

11. PALMER, Larry Ray. *What is Component Software?* 2021. Available also from: `https://www.easytechjunkie.com/what-is-component-software.htm`. [cit. 2021-4-11].

12. CRAIG, Rick D.; JASKIEL, Stefan P. *Systematic Software Testing.* Artech House, 2002. No. 1580535089.

13. *Test Environment for Software Testing.* 2021. Available also from: `https://www.guru99.com/test-environment-software-testing.html`. [cit. 2021-7-17].

14. *Test Environment for Software Testing.* 2019. Available also from: `https://www.professionalqa.com/test-environment`. [cit. 2021-7-17].

15. *Test Doubles — Fakes, Mocks and Stubs.* 2017. Available also from: `https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da`. [cit. 2021-7-18].

16. *Test Double.* 2011. Available also from: `http://xunitpatterns.com/Test%20Double.html`. [cit. 2021-7-18].

17. *What is meant by Stubs and Drivers?* 2020. Available also from: `https://www.professionalqa.com/stubs-and-drivers`. [cit. 2021-7-18].

18. *Mock Testing.* 2021. Available also from: `https://devopedia.org/mock-testing`. [cit. 2021-7-18].

19. *Test Stub.* 2011. Available also from: `http://xunitpatterns.com/Test%20Stub.html`. [cit. 2021-7-18].

20. *Test Spy.* 2011. Available also from: `http://xunitpatterns.com/Test%20Spy.html`. [cit. 2021-7-18].

21. *Mock Object.* 2011. Available also from: `http://xunitpatterns.com/Mock%20Object.html`. [cit. 2021-7-18].

22. *Fake Object.* 2011. Available also from: `http://xunitpatterns.com/Fake%20Object.html`. [cit. 2021-7-18].

23. *Test Automation Framework.* 2011. Available also from: `http://xunitpatterns.com/Test%20Automation%20Framework.html`. [cit. 2021-7-24].

24. *SUnit.* [N.d.]. Available also from: `http://sunit.sourceforge.net/`.

25. *JUnit.* [N.d.]. Available also from: `https://junit.org/`.

26. FOWLER, Martin. *XUnit.* 2006. Available also from: `https://martinfowler.com/bliki/Xunit.html`. [cit. 2021-7-24].

27. *XUnit.* [N.d.]. Available also from: `https://dbpedia.org/page/XUnit`. [cit. 2021-7-24].

28. MESZAROS, Gerard. *xUnit Test Patterns*. Pearson Education, Inc., 2007. No. 0131495054.
29. *Test Method*. 2011. Available also from: `http://xunitpatterns.com/Test%20Method.html`. [cit. 2021-7-25].
30. *Four-Phase Test*. 2011. Available also from: `http://xunitpatterns.com/Four%20Phase%20Test.html`. [cit. 2021-7-25].
31. *Testcase Class*. 2011. Available also from: `http://xunitpatterns.com/Testcase%20Class.html`. [cit. 2021-7-25].
32. *Testcase Object*. 2011. Available also from: `http://xunitpatterns.com/Testcase%20Object.html`. [cit. 2021-7-25].
33. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns*. Addison-Wesley, 1995. No. 0201633612.
34. *Test Suite Object*. 2011. Available also from: `http://xunitpatterns.com/Test%20Suite%20Object.html`. [cit. 2021-7-25].
35. LLOPIS, Noel. *Exploring the C++ Unit Testing Framework Jungle*. 2004. Available also from: `https://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle#cppunit`. [cit. 2021-8-1].
36. *CppUnit*. [N.d.]. Available also from: `https://sourceforge.net/projects/cppunit/`.
37. *CppUnit Cookbook*. [N.d.]. Available also from: `http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html`. [cit. 2021-7-31].
38. *CppUnit::TestFixture Class Reference*. [N.d.]. Available also from: `http://cppunit.sourceforge.net/doc/cvs/class_test_fixture.html`. [cit. 2021-7-31].
39. *Making assertions*. [N.d.]. Available also from: `http://cppunit.sourceforge.net/doc/cvs/group___assertions.html`. [cit. 2021-7-31].
40. *Protector Class Reference*. [N.d.]. Available also from: `http://cppunit.sourceforge.net/doc/cvs/class_protector.html`. [cit. 2021-8-1].
41. *CppUnit Documentation*. [N.d.]. Available also from: `http://cppunit.sourceforge.net/doc/cvs/index.html`. [cit. 2021-7-31].
42. *The C++ Standard Template Library (STL)*. 2021. Available also from: `https://www.geeksforgeeks.org/the-c-standard-template-library-stl/`. [cit. 2021-8-1].
43. *RTTI (Run-time type Information) in C++*. 2017. Available also from: `https://www.geeksforgeeks.org/g-fact-33/`. [cit. 2021-8-1].
44. *GNU LGPL*. 2007. Available also from: `https://www.gnu.org/licenses/lgpl-3.0.en.html`.
45. *Boost.Test*. [N.d.]. Available also from: `https://www.boost.org/doc/libs/1_76_0/libs/test/doc/html/index.html`.

46. *Boost C++ Libraries*. [N.d.]. Available also from: `https://www.boost.org/`.
47. *Test case fixture*. [N.d.]. Available also from: `https://www.boost.org/doc/libs/1_76_0/libs/test/doc/html/boost_test/tests_organization/fixtures/case.html`. [cit. 2021-8-1].
48. *Assertion severity level*. [N.d.]. Available also from: `https://www.boost.org/doc/libs/1_76_0/libs/test/doc/html/boost_test/testing_tools/tools_assertion_severity_level.html`. [cit. 2021-8-6].
49. *BOOST_TEST: universal and general purpose assertions*. [N.d.]. Available also from: `https://www.boost.org/doc/libs/1_76_0/libs/test/doc/html/boost_test/testing_tools/boost_test_universal_macro.html`. [cit. 2021-8-6].
50. *Exception correctness*. [N.d.]. Available also from: `https://www.boost.org/doc/libs/1_76_0/libs/test/doc/html/boost_test/testing_tools/exception_correctness.html`. [cit. 2021-8-6].
51. *Time-out for test cases*. [N.d.]. Available also from: `https://www.boost.org/doc/libs/1_76_0/libs/test/doc/html/boost_test/testing_tools/timeout.html`. [cit. 2021-8-6].
52. *Output streams testing tool*. [N.d.]. Available also from: `https://www.boost.org/doc/libs/1_76_0/libs/test/doc/html/boost_test/testing_tools/output_stream_testing.html`. [cit. 2021-8-6].
53. *Turtle*. [N.d.]. Available also from: `https://sourceforge.net/projects/turtle/`.
54. *Turtle Reference*. [N.d.]. Available also from: `http://turtle.sourceforge.net/turtle/reference.html`. [cit. 2021-8-7].
55. *Boost Software License*. [N.d.]. Available also from: `https://www.boost.org/users/license.html`.
56. *Google Test*. [N.d.]. Available also from: `http://google.github.io/googletest/`.
57. *Googletest Primer*. 2021. Available also from: `http://google.github.io/googletest/primer.html`. [cit. 2021-8-13].
58. *Advanced googletest Topics*. 2021. Available also from: `http://google.github.io/googletest/advanced.html`. [cit. 2021-8-13].
59. *Assertions Reference*. 2021. Available also from: `http://google.github.io/googletest/reference/assertions.html`. [cit. 2021-8-14].
60. *Matchers Reference*. 2021. Available also from: `http://google.github.io/googletest/reference/matchers.html`. [cit. 2021-8-14].
61. *jMock*. [N.d.]. Available also from: `http://jmock.org/`.

62. *Actions Reference*. 2021. Available also from: `http://google.github.io/googletest/reference/actions.html`. [cit. 2021-8-14].
63. *Mocking Reference*. 2021. Available also from: `http://google.github.io/googletest/reference/mocking.html`. [cit. 2021-8-14].
64. *BSD 3-Clause "New" or "Revised" License*. [N.d.]. Available also from: `https://github.com/google/googletest/blob/master/LICENSE`.
65. *Cpputest*. [N.d.]. Available also from: `https://cpputest.github.io/`.
66. *Core Manual*. [N.d.]. Available also from: `https://cpputest.github.io/manual.html`. [cit. 2021-8-15].
67. *CppUMock Manual*. [N.d.]. Available also from: `https://cpputest.github.io/mocking_manual.html`. [cit. 2021-8-16].
68. *Plugin Manual*. [N.d.]. Available also from: `https://cpputest.github.io/plugin_manual.html`. [cit. 2021-8-21].
69. *BSD 3-Clause "New" or "Revised" License*. [N.d.]. Available also from: `https://github.com/cpputest/cpputest/blob/master/COPYING`.
70. KORPELA, Henri. *C++11 - New features - Variadic templates*. 2012. Available also from: `http://www.cplusplus.com/articles/EhvU7k9E/`. [cit. 2021-9-5].
71. *Doxygen*. [N.d.]. Available also from: `https://www.doxygen.nl/index.html`. [cit. 2021-11-26].
72. *ERTMS in brief*. [N.d.]. Available also from: `https://www.ertms.net/about-ertms/ertms-signaling-levels/`. [cit. 2021-11-26].
73. *What is ERTMS?* [N.d.]. Available also from: `https://uic.org/rail-system/ertms/`. [cit. 2021-11-26].
74. *ERTMS Signaling levels*. [N.d.]. Available also from: `https://www.ertms.net/about-ertms/ertms-signaling-levels/`. [cit. 2021-11-26].
75. *ETCS B3 R2 GSM-R B1 – System Requirements Specification*. [N.d.]. Available also from: `https://www.era.europa.eu/content/set-specifications-3-etcs-b3-r2-gsm-r-b1_en`. [cit. 2021-11-27].
76. KADLČEK, David; STEJSKAL, Jan; JAHODA, Petr; UDAVICHENKA, Yury; MACHÁČEK, Jiří; VEJVODA, Štěpán. *Analýza projektu DMI displej pro simulátor ETCS*. 2021. Tech. rep. Faculty of Information Technology, CTU in Prague. [cit. 2021-11-28].
77. BARTOS, Jan; MENSHIKOV, Ivan; ONDRUSEK, David; PHAM, Xuan Trung; SAFAR, Jan; VOLOSIN, Alex Jan. *EVC pro ETCS simulátor, Analytická dokumentace*. 2021. Tech. rep. Faculty of Information Technology, CTU in Prague. [cit. 2021-11-28].

78. SKIPALA, Michal; BENK, Patrik; GORGOL, Matěj; KRASNENKOVA, Alina; ROSHCHUPKINA, Daria; STERNWALD, Jiří. *ETCS simulátor – RBC, Dokumentace*. 2021. Tech. rep. Faculty of Information Technology, CTU in Prague. [cit. 2021-11-28].

79. GOLMGREN, Nikita; BÍLEK, Matouš; KRAVTSOV, Aleksei; LANCA, Matěj; SLANINOVÁ, Dominika; UMPRECHT, Jan; WICHTERLE, David. *Výpočet brzdné křivky pro ETCS simulátor, Analytická dokumentace*. 2021. Tech. rep. Faculty of Information Technology, CTU in Prague. [cit. 2021-11-28].

80. GOLMGREN, Nikita; BÍLEK, Matouš; KRAVTSOV, Aleksei; LANCA, Matěj; SLANINOVÁ, Dominika; UMPRECHT, Jan; WICHTERLE, David. *JRU a JRU DL Tool pro ETCS simulátor, Analytická dokumentace*. 2021. Tech. rep. Faculty of Information Technology, CTU in Prague. [cit. 2021-11-28].

81. *System Requirements Specification, Chapter 7, ERTMS/ETCS language*. 2016. Tech. rep. European Union Agency For Railways. Available also from: `https://www.era.europa.eu/content/set-specifications-3-etcs-b3-r2-gsm-r-b1_en`. [cit. 2021-12-28].

82. *System Requirements Specification, Chapter 8, Messages*. 2016. Tech. rep. European Union Agency For Railways. Available also from: `https://www.era.europa.eu/content/set-specifications-3-etcs-b3-r2-gsm-r-b1_en`. [cit. 2021-12-28].

83. *Eclipse Mosquitto*. [N.d.]. Available also from: `https://mosquitto.org/`.

84. *System Requirements Specification, Chapter 3, Principles*. 2016. Tech. rep. European Union Agency For Railways. Available also from: `https://www.era.europa.eu/content/set-specifications-3-etcs-b3-r2-gsm-r-b1_en`. [cit. 2021-12-28].

Appendix A
Acronyms

API	Application programming interface
CTU	Czech Technical University
DMI	Driver Machine Interface
ERTMS	European Rail Traffic Management System
ETCS	European Train Control System
EU	European Union
EVC	European Vital Computer
FIT	Faculty of Information Technology
FTS	Faculty of Transportation
GSM-R	Global System for Mobiles - Railway
JRU	Juridical Recording Unit
LIFO	Last In, First out
MA	Movement authority
MQTT	MQ Telemetry Transport
RBC	Radio Block Centre
RTTI	Run-time type information
STL	The Standard Template Library

Appendix B

Contents of enclosed storage medium

```
 readme.txt ........................ file with medium contents description
 src
   cpputest_extension................. CppUTest extension source files
   rbc_component_tests ......................... RBC tests source files
   thesis ............................... LaTeX source files of the thesis
 text
   DP_Kasalicka_Katerina_2022.pdf ........ thesis text in PDF format
```